GNU manuals (and a few others)

This (the Directory node) gives a menu of major topics.

Some of these are available in printed form. The local installation status of most of it.

A2ps

A text (and other formats) to PostScript converter, with pretty printing.

Auc-TeX

A much enhanced LaTeX mode for gnuemacs.

Automake

Create GNU Standards-compliant Makefiles from template file

Bash

Bash (Bourne Again SHell)

Bison

Bison parser generator (= GNU yacc).

Calc

An advanced calculator and mathematical tool which runs under the emacs environment.

CL

Partial Common Lisp support for Emacs Lisp.

Cpp

GNU C compiler pre-processor.

Cvs

GNU version control system

Dvips

A DVI-to-PostScript translator.

Elisp

GNU Emacs Lisp.

Emacs

The extensible self-documenting text editor.

Eplain

Expanding on plain TeX.

Forms

Forms mode is an Emacs package that lets the user edit a data structure by filling in a form.

Flex

Fast lexical analyzer generator (= GNU lex).

Gas

GNU assembler `gas'.

Gawk

Gnu version of awk.

Gcc

GNU C compiler.

G++

GNU C++. As of version 2.0 incorporated into gcc.

Gdb

The source-level C debugger.

Gdbm

GNU alternative to ndbm library

Gdiff

GNU version of diff.

Gmp

GNU C multi-precision library.

Gnumake

GNU make.

Gnus

GNU news reader.

Mh-e

Emacs interface to MH

Gzip

GNU zip - compression utilities.

Info

Documentation browsing system.

Ispell

Interactive spelling checker.

Kpathsea

The path searching library used by TeX,Metafont, dvips and xdvi.

LibG++

The GNU c++ library.

Libtool

Script to allow package developers to provide generic shared library support.

Perl

A language for easy manipulation of text, files and processes. Pages specific to Version 5.

Python

An interpreted, extensible, embeddable, interactive, object-oriented programming language. Library info only.

Readline

The GNU Readline library.

Texinfo

With one source file, make either a printed manual (through TeX) or an Info file (through texinfo).

Textutils

GNU versions of many standard UNIX text manipulation tools

Vera

V.E.R.A., a list dealing with computational acronyms.

VIP

A VI-emulation for Emacs.

Note1: most of these documents are distributed in texinfo form with the associated software. These documents are updated whenever the associated software is updated **at this site**, thus these are not necessarily the most up-to-date versions available. As such they may not be relevant at sites outside cl.cam. The texinfo documents are converted to html format using the *texi2html* tool available from those splendid people at CERN.

Note2: to overseas visitors - many other sites on the Web have similar collections to this one, it may be worth the effort looking for a collection closer to home.

Note3: These documents are not available for FTP access at this site so don't even bother to ask!

ckh@cl.cam.ac.uk

a2ps, version 4.10

General Purpose PostScript Generating Utility

Edition 4.10, 10 March 1998

Akim Demaille Miguel Santana

a2ps, version 4.10

TOC

Introduction

This document describes `a2ps` version 4.10. The latest versions may be found on the a2ps home page.

We tried to make this document informative and pleasant. It tries to be more than a plain reference guide, and intends to offer information about the concepts or tools etc. that are related to printing PostScript. This is why it is now that big: to offer you all the information you might want, **not** because using `a2ps` would be difficult.

Please, send us emailcards `:)`. Whatever the comment is, or if you just like `a2ps`, write to Miguel Santana and Akim Demaille.

Description

`a2ps` formats files for printing on a PostScript printer.

The format used is nice and compact: normally two pages on each physical page, borders surrounding pages, headers with useful information (page number, printing date, file name or supplied header), line numbering, pretty-printing, symbol substitution etc. This is very useful for making archive listings of programs.

Compared to the previous versions, `a2ps` version 4.10 provides:

1. various configuration files (see section Configuration Files)
2. some powerful meta-sequences to define the headers the way you want (see section Meta Sequences);
3. a totally open approach of fonts and encodings;
4. a rather strong support of Latin 2, 3, 4, 5 and 6 encodings, thanks to `Ogonkify` (see section

'Overview' in Ogonkify manual), written by Juliusz Chroboczek.

5. fully customizable output style: fonts, background and foreground colors, line numbering style etc. (see section Designing PostScript Prologues).
6. the ability to delegate the processing of some files to other filters (see section Your Delegations).
7. and of course, the ability to pretty-print sources written in quite a few various languages (see section Pretty Printing).

Version Numbering Scheme

`a2ps` is alive. To let you know its status, the version numbers are structured *à la* Linux.

What is to be understood in `a2ps version x.y.z.t'?

- if y is even, then this is an official release, fully tested and checked on many platforms, and widely distributed.
- if y is odd, then
  - if there is no t, then this is a distributed beta, with brand new features, and maybe some new bugs, or lacks in the documentation. It is only available on a2ps home page.
  - if there is t, then this is an alpha, only available to members of the `a2ps` mailing list, so that problems are explored and tracked down.

Reporting Bugs

We try hard to make `a2ps` portable on any Unix platform, and bug free. But sometimes there can still be bad surprises, even after having compiled and checked `a2ps` on several very different platforms.

You may encounter some of these problems yourself. In any case, please never abandon without giving us a chance. We need information from everybody so that mistakes get fixed as fast as possible.

So, if you have a problem (configuration error, compilation error, runtime error, documentation error or unclear), first check on a2ps home page and in the FAQ (see section Frequently asked questions) if the issue has not been addressed yet. If it is not the case, but it appears that the version of `a2ps` you have is old, consider upgrading.

If the problem persists, send us a mail (a2ps-bugs@inf.enst.fr) which title is `a2ps version: short-description' in which you mention the name of your machine and OS, the version of `a2ps`, every detail you have on your compiler, and as much traces as possible (the error messages you get on the screen, or the output of `make` when it fails etc.).

Be sure to get a quick answer.

a2ps Mailing List

There is a mailing list in which are discussed various topics around `a2ps`. There are also announcements about the version in alpha testing, requests for comments, new sheets, etc.

To subscribe to the list, send a mail to a2ps-request@inf.enst.fr, with `subscribe' in the body.

Please, note that the mailing list is by no means a bug reporting address: use a2ps-bugs@inf.enst.fr instead.

Helping the Development

If you like `a2ps` and if you feel like helping, there are several things you can do.

Translation

The interface of `a2ps` is under `GNU gettext` which means that all the messages can be translated, without having to look at the code of `a2ps`: you don't need to be a programmer at all. All the details are available on the a2ps translation page.

Style Sheets

Since `a2ps` is evolving and getting more powerful, the style sheets should be checked and improved. There are too many so that the authors work on them. Therefore if you feel your favorite language is not honored as it should be, improve the style sheet! (see section Pretty Printing for details.)

Encodings

`a2ps` is wide open to any 8-bit encoding. If your language is not covered today by `a2ps`, you can easily provide the support yourself. Honestly, the trickiest part is to find correct **free** fonts that support your mother tongue (see section Encoding Files, to know more).

Fonts

There are still some characters missing in Ogonkify. See the list of missing characters and the Ogonkify home page for details.

Documentation

If you feel something is missing or is unclear, send us your contributions.

Porting

Porting a program to special architectures (MS-DOS, OS/2 etc.), or building special packages (e.g., RPM) requires having an access to these architectures. If you feel like maintaining such a port, tell us.

Features

Well, if you feel like doing something else, go ahead!

User's Guide

This chapter is devoted to people who don't know `a2ps` yet: we try to give a soft and smooth introduction to the most useful features. For a reference manual, see section Invoking `a2ps`. For the definition of some words, see section Glossary, for questions you have, see section Frequently asked questions.

Purpose

`a2ps` is a program that takes a text file (i.e., human readable), and makes a PostScript file out of it.

How to print

To print a file `` `doc.txt' ``, just give it to `a2ps`: the default setting should be the one you'd like:

```
gargantua ~ $ a2ps doc.txt
[doc.txt (plain): 9 pages on 5 sheets]
[Total: 9 pages on 5 sheets] sent to the default printer
```

`a2ps` sent the file `` `doc.txt' `` to the default printer, writing two columns of text on a single face of the sheet. Indeed, by default `a2ps` uses the option `-2', standing for two virtual pages.

Basics for Printing

Say you want to print the C file `` `bar.c' ``, and its header `` `foo.h' ``, on 4 virtual pages, and save it into the file `` `foobar.ps' ``. Just hit:

```
gargantua $ a2ps foo.h bar.c -4 -o foobar.ps
[foo.h (C): 1 page on 1 sheet]
[bar.c (C): 3 pages on 1 sheets]
[Total: 4 pages on 2 sheets] saved into the file `foobar.ps'
```

The option `-4' tells `a2ps` to make four virtual pages: two rows by two columns. The option `-o' (which is the short version of `--output=foobar.ps') specifies the output file. The long options must always be separated by spaces, though short options with no arguments may be grouped.

Note too that the options may be before or after the files, it does not matter.

If you send `` `foobar.ps' `` to a printer, you'll discover that the keywords were highlighted, that the strings and comments have a different face. Indeed, `a2ps` is a pretty-printer: if it knows the (programming) language in which your file is written, it will try to make it look nice on the paper.

But too bad: `` `foo.h' `` is only one virtual page long, and `` `bar.c' `` takes three. Moreover, the comments are essential in those files. And even worse: the system's default printer is out of ink. Thanks god, precious options may help you:


```
gargantua $ a2ps -A4 foo.h bar.c --prologue=gray -P lw
[foo.h (C): 1 page on 1 sheet]
[bar.c (C): 3 pages on 1 sheet]
[Total: 4 pages on 1 sheet] sent to the printer `lw'
```

Here `-A' is used to Allow several files on the same sheet, `-P lw' means to print on the printer named `lw', and finally, the long option `--prologue' asked to use one of the alternative printing styles of `a2ps'. There are other available prologues (see option `--prologue' in section Input Options), and you can even design yours (see section Designing PostScript Prologues).

Special Printers

There are three special printers pre-defined.

The first one, `void', sends the output to the trash. Its main use is to see how many pages would have been used.

```
gargantua ~ $ a2ps -P void parsessh.c
[parsessh.c (C): 33 pages on 17 sheets]
[Total: 33 pages on 17 sheets] sent to the printer `void'
```

The second, `display' sends the output to `ghostview', so that you can check the output without printing. Of course if you don't have `ghostview', it won't work... And it is up to you to configure another displaying application (see section Your Printers).

And the last, `file' saves the output into a file named after the file you printed (e.g., saves into `foo.ps' when you print `foo.c').

Using Delegations

`a2ps' is able to decide that it is not the right tool to do what you want, in which case it delegates the task to other programs. What you should retain from this, is, *forget that there are delegations*. Indeed, the interface with the delegations has been designed so that you don't need to be aware that they exist to use them. Do as usual.

As an example, if you need to print a PostScript file, just hit:

```
gargantua ~ $ a2ps article.ps -d
[article.ps (ps, delegated to PsNup): 7 pages on 4 sheets]
[Total: 8 pages on 4 sheets] sent to the default printer
```

Using your defaults, `a2ps' delegated the task to put two virtual pages per physical page to `psnup', a powerful filter part of the famous `psutils' by Angus Duggan.

Suppose now that you want to display a Texinfo file. Then, provided you have all the programs `a2ps' needs, just hit


```
gargantua ~ $ a2ps a2ps.texi -P display
[a2ps.texi (texinfo, delegated to texi2dvi): 75 pages on 38 sheets]
[Total: 76 pages on 38 sheets] sent to the printer `display'
```

Once the documentation read, you know you want to print just the pages 10 to 20, and the cover. Just hit:

```
gargantua ~ $ a2ps a2ps.texi --pages=1,10-20 -d
[a2ps.texi (texinfo, delegated to texi2dvi): 13 pages on 7 sheets]
[Total: 14 pages on 7 sheets] sent to the default printer
```

A final word: compressed files can be treated in the very same way:

```
gargantua ~ $ a2ps a2ps.texi.gz -a1,10-20 -d
[a2ps.texi (compressed, delegated to Gzip-a2ps): 13 pages on 7 sheets]
[Total: 14 pages on 7 sheets] sent to the default printer
```

You should be aware that:

- the option `-Z' enables the delegations if they are not (see `--list=defaults' for your settings);
- the set of delegations is customizable, to know the delegations your `a2ps` knows, consult `a2ps --list=delegations'.

Checking the Defaults

If `a2ps` did not have the behavior expected, this may be because of the default settings given by your system administrator. Checking those default values is easy:

```
gargantua ~ $ a2ps --list=defaults
Configuration status of a2ps 4.10
Sheets:
  default paper         = A4
  layout per page       = 2 x 1 (landscape, rows first)
  borders               = yes
  compact mode          = yes
  inside margin         = 0
More stuff deleted here
Internals:
  verbosity level     = 2
  file command        = /usr/ucb/file -L
  temporary directory =
  library path        =
        /inf/soft/infthes/demaille/.a2ps
        /usr/local/share/a2ps/sheets
        /usr/local/share/a2ps/ps
        /usr/local/share/a2ps/encoding
        /usr/local/share/a2ps/afm
        /usr/local/share/a2ps/printers
```


```
/usr/local/share/a2ps
```

Remember that the on-line help is always available. Moreover, if your screen is small, you may *pipe* it into `more`. Just trust this:

```
a2ps --help | more
```

Important parameters

Many things are parameterizable in `a2ps`, but two things are just essential to make sure everything goes right:

The paper

Make sure that the paper `a2ps` uses is the same as your printer (See section Sheets Options, option `--medium').

The encoding

Make sure that the *encoding* `a2ps` uses is the same as the standard alphabet in your country (See section Input Options, option `--encoding').

Both values may be checked with `a2ps --list=defaults'.

Localizing

`a2ps` provides some Native Language Support, that is speaking your mother tongue. It uses three special features for non-English languages:

the tongue

i.e., the language used by the interface,

the date

i.e., the format and the words used in the language to specify a date,

the ordering

i.e., the alphabetic order that corresponds to your tongue (for instance the French letter `é' is to be considered has an `e', which is not what says its ASCII code).

To use these features, you may set your environment variable `LANG` to the standard code representing your language:

`ca`

Catalan

`cs`

Czech

`de`

German

`es`

`es`
: Spanish

`fr`
: French

`it`
: Italian

`ko`
: Korean

`nl`
: Dutch

`pl`
: Polish

`tr`
: Turkish

The problem with this approach is that a lot more than just messages and time information is affected: especially the way numbers are written changes, what may cause problems which `awk` and such.

So if you just want messages, time format and sorting order to be localized, then define:

```
set LC_MESSAGES=fr
set LC_TIME=fr
set LC_COLLATE=fr
export LC_MESSAGES
export LC_TIME
export LC_COLLATE
```

Note that the `make install' *must* have been done in order to have NLS support.

Interfacing with other programs

Here are some tips on how to use `a2ps` with other programs.

elm

Once you are in elm, hit o to enter in the options edition menu, hit p to edit the printer command, and enter `a2ps -2gEmail --strip=3 %s -d'. The option `-g' is given because there are some special escapes that `a2ps` understands, and replaces with the correct character.

pine

Jan Chrillesen suggests us how to use `a2ps` with the Pine mail-reader. Add the following to `` `.pinerc' `` (of course you can put it in `` `pine.conf' `` as well):


```
# Your printer selection
printer=a2ps -2gEmail --strip=3 -d

# Special print command if it isn't one of the standard printers
personal-print-command=a2ps -2gEmail --strip=3 -d
```

Netscape

This is actually valid for any program that generates PostScript that you want to post-process with `a2ps`.

Since the data is sent through the standard input, you need to specify that it is PostScript data (see section Style Sheet Files). And just to ensure the delegations are set, force `-Z'. Therefore, in the `Print' windows of Netscape, use the following command:

```
a2ps -ZEps -
```

Do not forget to tell Netscape whether your printer supports colors, and the type of paper it uses.

Invoking a2ps

Calling `a2ps` is fairly simple:

```
a2ps Options... Files...
```

If no Files... are given, `a2ps` prints its standard input. If `-' appears in the Files..., it designates the standard input too.

Command line options

To read the options and arguments that you give it, `a2ps` uses GNU `getopt`, hence:

- the options (short with arguments or long) must be separated by spaces.
- the order between options and files does not matter: `a2ps -4m main.c' and `a2ps main.c -4m' are identical.
- the order between options **does matter**, especially between options that influence the same parameters. For instance `a2ps -1 -l132' is not the same as `a2ps -l132 -1' (the latter being equivalent to `a2ps -1'.
- short options may be grouped together: `a2ps -4mg main.c -P printer'
- when there are no ambiguities, long options can be abbreviated, e.g., `--pro' will be understood as `--prologue',
- `--' ends the options. Anything behind `--' is considered to be a file: `a2ps -- -2' prints the file `-2' (2).

Here after a boolean is considered as true (i.e. setting the option on), if boolean is `yes', or `1'; as false if it

equals `no' or `0'; and raise an error otherwise. The corresponding short option takes no arguments, but corresponds to a positive answer.

When an argument is presented by square brackets, it means that it is optional. Optional arguments must never be separated from the short option.

Tasks Options

When these options are used, `a2ps` does not print. It executes the task and exits successfully.

Option: **-V**

Option: **--version**

print version and exit successfully.

Option: **-h**

Option: **--help**

Print a short help, and exit successfully.

Option: **--copyright**

Display Copyright and copying conditions, and exit successfully.

Option: **--guess**

Act like `file` does: display the (key of the) type of the Files.

For instance, on a `C` file, you expect it to answer `c', and upon a PostScript file, `ps'.

This can be very useful on broken systems to understand why a file is printed with a bad style sheet (see section Style Sheet Files).

Option: **--list=topic**

Display a report on `a2ps`' status with respect to topic, and exit successfully. topic can be any non-ambiguous abbreviation of:

`defaults'

`options'

Give an extensive report on `a2ps` configuration and installation.

`features'

Known media, encodings, languages, prologues, printers, macro meta sequences, delegations and user options are reported. In a word, anything that you may define.

`delegations'

Detailed list of the delegations. See section Your Delegations.

`encodings'

Detailed list of known encodings. See section Some Encodings.

`macro-meta-sequences'
: Detailed list of the macro meta sequences. See section Your Meta Sequences.

`media'
: Detailed list of known media. See section Your Media.

`prologues'
: Detailed list of PostScript prologues. See section Designing PostScript Prologues.

`printers'
: Detailed list of printers and named outputs. See section Your Printers.

`style-sheets'
: Detailed list of the known style sheets. See section Known Style Sheets.

`user-options'
: Detailed list of the user options. See section Your Shortcuts

There are also options meant for the maintainers only, presented for sake of completeness.

`texinfo-style-sheets'

`ssh-texi'
: Detailed list of known style sheets in Texinfo format. If the `sheet` verbosity is set, report version numbers, requirements and ancestors.

`html-style-sheets'

`ssh-html'
: Detailed list of the style sheets in `HTML` format.

`texinfo-encodings'

`edf-texi'
: Detailed list of encodings, in Texinfo format.

`texinfo-prologues'

`pro-texi'
: Detailed list of prologues, in Texinfo format.

Global Options

These options are related to the interface between you and `a2ps`.

Option: **-q**

Option: **--quiet**

Option: **--silent**

be really quiet

Option: **-v[level]**

Option: **--verbose[=level]**

tell what we are doing. At

- level = 0, report nothing,
- level = 1, `a2ps` just prints the total number of pages printed,
- level = 2 (default), it reports it for each file,
- above, it gives internal details.

There is also an interface made for the maintainer with finer grained selection of the verbosity level. level is a list of tokens (non ambiguous abbreviation are valid) separated by either `,' or `+'. The tokens may be:

`sheets'
: the style sheets,

`pathwalk'
`pw'
: the search for files,

`encodings'
: the encodings,

`configuration'
`options'
: reading the configurations files and the options,

`files'
: inputs and outputs,

`tools'
: launched programs or shell commands ; triggers the escape `?V' on (see section Available Meta Sequences),

`fonts'
: the font,s

`ppd'
: PPD processing,

`meta-sequences'
: the expansion of meta sequences,

`parsers'
: any parsing process (style sheets, PPD files etc.),

`all'
: all the messages.

Option: **-=shortcut**

Option: **--user-option=shortcut**

use the shortcut defined by the user. See section Your Shortcuts.

Note that contrary to the other options, this special construct must not be after the files to print. This is actually a bug. If somebody sees how to fix this, please send us a patch.

Option: **--debug**

enable debugging features. They are:

- print the overall BoundingBox in PostScript;
- down load a PostScript debugger which helps understanding why a printer may reject a file.

Option: **--macro-meta-sequence=key[:value]**

Without value, unset the macro meta sequence key. Otherwise, set it to value. See section Your Meta Sequences for more details.

Sheets Options

This options specify the general layout, how the sheet should be used.

Option: **-M** *medium*

Option: **--medium=medium**

use output medium medium. See the output of `a2ps --list=media' for the list of supported media. Typical values are `A3', `A4', `A5', `B4', `B5', `Letter', `Legal'. `A4dj', `Letterdj' are also defined for Desk Jet owners, since that printer needs bigger margins.

Option: **-r**

Option: **--landscape**

print in landscape mode

Option: **-R**

Option: **--portrait**

print in portrait mode

Option: **--columns=num**

specify the number of columns of virtual pages per physical page.

Option: **--rows=num**

specify the number of rows of virtual pages per physical page.

Option: **--major=direction**

specify whether the virtual pages should be first filled in rows (direction = `rows') or in columns (direction = `columns').

Option: **-1**

TOC

1 x 1 portrait, 80 chars/line, major rows (i.e. alias for `--columns=1 --rows=1 --portrait --chars-per-line --major=rows').

Option: **-2**

2 x 1 landscape, 80 chars/line, major rows.

Option: **-3**

3 x 1 landscape, 80 chars/line, major rows.

Option: **-4**

2 x 2 portrait, 80 chars/line, major rows.

Option: **-5**

5 x 1 landscape, 80 chars/line, major rows.

Option: **-6**

3 x 2 landscape, 80 chars/line, major rows.

Option: **-7**

7 x 1 landscape, 80 chars/line, major rows.

Option: **-8**

4 x 2 landscape, 80 chars/line, major rows.

Option: **-9**

3 x 3 portrait, 80 chars/line, major rows.

Option: **-j**

Option: **--borders=boolean**

print borders around virtual pages.

Option: **-A**

Option: **--compact=boolean**

Compact mode for a sequence of files. This option allows the printing of more than one file on the same page. Otherwise, the beginning of each file will be printed in a new sheet.

Option: **--margin[=num]**

Specify the size of the margin (num PostScript points, or 12 points without arguments) to leave in the inside (i.e. left for the front side page, and right for the back side). This is intended to ease the binding.

Pages Options

This options are related to the content of the virtual pages.

Option: **--line-numbers=number**

print the line numbers from number lines to number lines.

Option: **-C**

Alias for `--line-numbers=5'.

Option: **-f** *size[unit]*

Option: **--font-size=size[unit]**

scale font to size for body text. size is a float number, and unit can be `cm' for centimeters, `points' for PostScript points, and `in' for inches. Default unit in `inches'.

To change the fonts used, change the current prologue (see section Designing PostScript Prologues.

Option: **-l** *num*

Option: **--chars-per-line=num**

Set the number of columns per page. num is the real number of columns devoted to the body of the text, i.e., no matter whether lines are numbered or not.

Option: **-L** *num*

Option: **--lines-per-page=num**

Set the lines per virtual page for printing. The font size is automatically scaled up to fill in the whole page. This is useful for printing preformatted documents which have a fixed number of lines per page. The minimum number of lines per page is set at 40 and maximum is at 160. If a number less than 40 is supplied, scaling will be turned off.

Option: **-m**

Option: **--catman**

Understand UNIX manual **output** ie: 66 lines per page and possible bolding and underlining sequences. The understanding of bolding and underlining is there by default even if `--catman' is not specified.

If your file is actually a UNIX manual *input*, i.e., a roff file, then depending whether you left `a2ps` delegate or not, you will get a readable version of the text described, or a pretty-printed version of the describing file (see section Your Delegations).

Option: **-T** *num*

Option: **--tabsize=num**

set tabulator size to num. This option is ignored if `--interpret=no` is given.

Option: **--non-printable-format=format**

specify how non-printable chars are printed. format can be
`caret'
Use classical Unix representation: `^A', `M-^B' etc.
`space'
A space is written instead of the non-printable character.
`question-mark'
A `?' is written instead of the non-printable character.
`octal'
For instance `\001', `177' etc.
`hexa'
For instance `\x01', `\xfe' etc.
`emacs'
For instance `C-h', `M-C-c' etc.

Headings Options

These are the options through which you may define the information you want to see all around the pages.

All these options support text as an argument, which is composed of plain strings and meta sequences. See section Meta Sequences for details.

Option: **-B**

Option: **--no-header**

no page headers at all.

Option: **-b[text]**

Option: **--header[=text]**

set the page header

Option: **--center-title[=text]**

Option: **--left-title[=text]**

Option: **--right-title[=text]**

Set virtual page center, left and right titles to text.

Option: **-u[text]**

Option: **--underlay[=text]**

use text as under lay (or water mark), i.e., in a light gray, and under every page.

Option: **--left-footer[=text]**

Option: **--footer[=text]**

Option: **--right-footer[=text]**

Set sheet footers to text.

Input Options

Option: **-a[Page range]**

Option: **--pages[=Page range]**

With no argument, print all the page, otherwise select the pages to print. Page range is a list of interval, such as `-a1': print only the first page, `-a-3,4,6,10-': print the first 3 pages, page 4 and 6, and all the page after 10 (included). Giving `toc' prints the table of content whatever its page number is.

The pages referred to are the *input* pages, not the output pages, that is, in `-2', printing with `-a1' will print the first virtual page, i.e., you will get half the page filled.

Note that page selection does work with the delegations (see section Your Delegations).

Option: **-c**

Option: **--truncate-lines=boolean**

Cut lines too large to be printed inside the borders. The maximum line size depends on format and font size used and whether line numbering is enabled.

Option: **-i**

Option: **--interpret=boolean**

interpret tab, bs and ff chars

Option: **--end-of-line=type**

Specify what sequence of characters denotes the end of line. type can be:

`n`
`unix`
: `\n'.

`r`
`mac`
: `\r'.

`nr`
: `\n\r'. As far as we know, this type of end-of-line is not used.

`pc`
`rn`

`\r\n'. This is the type of end-of-line on MS-DOS.

`any`
`auto`

Any of the previous cases. This last case prevents the bad surprises with files from PC (trailing `^M').

Option: **-X** *key*

Option: **--encoding=key**

Use the input encoding identified by key. See section Some Encodings, and the result of `a2ps --list=encodings' to know what encodings are supported. Typical values are `ASCII', `latin1'... `latin6', `ison' etc.

Option: **--stdin=filename**

Give the name filename to the files read through the standard input.

Option: **-t** *name*

Option: **--title=name**

Give the name name to the document. Meta Sequences can be used (see section Meta Sequences).

This is used for instance in the name given to the document from within the PostScript code (so that GhostView and others can display a file with its real title, instead of just the PostScript file name).

It is **not** the name of the output. It is just a logical title.

Option: **--prologue=prologue**

Use prologue as the PostScript prologue for `a2ps`. prologue must be in a file named `` `prologue.pro' ``, which must be in a directory of your library path (see section Library Files). Currently the available prologues are:

`bw'

Style is plain: pure black and white, with standard fonts.

`color'

Colors are used to highlight the keywords.

`gray'

Gray foreground is used for comments and labels.

`gray2'

Black foreground is used for comments and labels.

`matrix'

The layout is the same as `bw', but alternating gray and white lines. There are two macros defining the behavior: `pro.matrix.cycle' defines the length of the cycle (number of white and gray lines). It defaults to 6. `pro.matrix.gray' defines the number of gray lines. Default is 3.

Option: **--print-anyway=boolean**

force binary printing. By default, the whole print job is stopped as soon as a binary file is detected. To detect such a file we make use of a very simple heuristic: if the first sheet of the file contains more than 40% of non-printing characters, it's a binary file.

Option: **-Z**

Option: **--delegate=boolean**

Enable delegation of some files to delegated applications. If delegating is on, then `a2ps` will *not* process the file by itself, but will call an application which handles the file in another way. If delegation is off, then `a2ps` will process *every* file itself.

Typically most people don't want to pretty-print a PostScript source file, but want to print what describes that file. Then set the delegations on.

See section Your Delegations for information on delegating, and option `--list=delegations' for the applications your `a2ps` knows.

Option: **--toc[=format]**

Generate a Table of Content, which format is a Meta Sequence (see section Meta Sequences) processed as a PreScript file (see section PreScript). If no format is given, don't generate the Table of Content.

Note that it is mostly useful to define a macro (see section Your Meta Sequences), for instance, in your `a2psrc' file:

```
MacroMetaSequence: toc.mine \
\\Keyword{Table of Content}\n\
#-1!f\
|$2# \\keyword{$-.20n} sheets $3s< to $3s> ($2s#) \
pages $3p<-$3p> $4l# lines\n||\
\\Keyword{End of toc}\n
```

and to give that macro name as argument to `--toc': `a2ps *.c --toc=#{toc.mine}'.

Note too that you can generate only the table of content using `--pages':

```
a2ps *.c --toc=#{toc.mine} -atoc
```

Pretty Printing Options

These options are related to the pretty printing features of `a2ps`.

Option: **--highlight-level=level**

Specify the level of highlighting. level can be

`none'

no highlighting

`normal'

regular highlighting

`heavy'

even more highlighting.

See the documentation of the style sheets (`--list=style-sheets') for a description of `heavy' highlighting.

Option: **-g**

Alias for `--highlight-level=heavy'.

Option: **-E** *[language]*

Option: **--pretty-print[=language]**

With no arguments, set automatic style selection on. Otherwise, set style to language. Note that setting language to `plain' turns off pretty-printing. See section Known Style Sheets, and the output of `--list=style-sheets' for the available style sheets.

If language is `key.ssh', then don't look in the library path, but use the file `key.ssh'. This is to ease debugging non installed style sheets.

Option: **--strip-level=num**

Depending on the value of num:

`0'

everything is printed;

`1'

regular comments are not printed

`2'

strong comments are not printed

`3'

no comment is printed.

This option is valuable for instance in `java` in which case strong comments are the so called documentation comments, or in `SDL` for which some graphical editors pollutes the specification with internal data as comments.

Note that the current implementation is not satisfactory: some undesired blank lines remain. This is planed to be fixed.

Output Options

These are the options to specify what you want to do out of what `a2ps` produces. Only a single destination is possible at a time, i.e., if ever there are several options `-o', `-P' or `-d', the last one is honored.

Option: **-o** *file*

Option: **--output=file**

leave output to file file. If file is `-', leave output to the standard output.

Option: **--version-control=type**

to avoid loosing a file, `a2ps` offers backup services. This is enabled when the output file already exists.

The type of backups made can be set with the `VERSION_CONTROL` environment variable, which can be overridden by this option. If `VERSION_CONTROL` is not set and this option is not given, the default backup type is `existing'. The value of the `VERSION_CONTROL` environment variable and the argument to this option are like the GNU `Emacs` `version-control' variable; they also recognize synonyms that are more descriptive. The valid values are (unique abbreviations are accepted):

`none'

`off'

Never make backups (override existing files).

`t'

`numbered'

Always make numbered backups.

`nil'

`existing'

Make numbered backups of files that already have them, simple backups of the others.

`never'

`simple'

Always make simple backups.

Option: **--suffix=suffix**

The suffix used for making simple backup files can be set with the `SIMPLE_BACKUP_SUFFIX` environment variable, which can be overridden by this option. If neither of those is given, the default is `~', as it is in `Emacs`.

Option: **-P** *name*

Option: **--printer=name**

send output to printer name. See item `Printer:' and `Unknown printer:' in section Your Printers and results of option `--list=defaults' to see the bindings between printer names and commands.

Option: **-d**

send output to the default printer. See item `DefaultPrinter:' in section Your Printers.

PostScript Options

The following options are related only to variations you want to produce onto a PostScript output.

Option: **--ppd[=key]**

With no argument, set automatic `PPD` selection, otherwise set the `PPD` to key. FIXME: what to read.

Option: **-n** *num*

Option: **--copies=num**

print num copies of each page

Option: **-s** *num*

Option: **--side=num**

number of sheet sides. num is of course 1 or 2.

Option: **-D** *key[:value]*

Option: **--setpagedevice=key[:value]**

Pass a page device definition to the generated PostScript output. If no value is given, key is removed from the definitions. Note that several `-D' can be accumulated.

For example, command

```
a2ps -DDuplex:true -DManualFeed:true report.pre
```

prints file `report.pre' in duplex (two sides) mode manually fed.

Page device operators are implementation dependent but they are standardized. See section Page Device Options for details.

Option: **-S** *key[:value]*

Option: **-S** *key[::value]*

Option: **--statusdict=key[:value]**

Option: **--statusdict=key[::value]**

Pass a statusdict definition to the generated PostScript output. `statusdict` operators and variables are implementation dependent; see the documentation of your printer for details. See section Statusdict Options for details. Several `-S' options can be accumulated.

If no value is given, key is removed from the definitions.

With a single colon, pass a call to an operator, for instance:

```
a2ps -Ssetpapertray:1 quicksort.c
```

prints file `quicksort.c' by using paper from the paper tray 1 (assuming that printer supports paper tray selection).

With two colons, define variable key to equal value. For instance:

```
a2ps -Spapertray::1 quicksort.c
```

produces


```
/papertray 1 def
```

in the PostScript.

Option: **-k**

Option: **--page-prefeed**

enable page prefeeding. It consists in positioning the sheet in the printing area while the PostScript is interpreted (instead of waiting the end of the interpretation of the page before pushing the sheet). It can lead to an significant speed up of the printing.

`a2ps` quotes the access to that feature, so that non supporting printers won't fail.

Option: **-K**

Option: **--no-page-prefeed**

disable page prefeeding.

Meta Sequences

The meta sequences (sometimes called escapes in other terminologies), are some sequences of characters that will be replaced by their values. They are very much like variables.

Use of Meta Sequences

They are used in several places in `a2ps`:

Page markers

Headers, footers, titles and the water mark (see section Headings Options), in general to print the name of file, page number etc. On a new sheet `a2ps` first draws the water mark, then the content of the first page, then the frame of the first page, (ditto with the others), and finally the sheet header and footers. This order must be taken into account for some meta sequences (e.g., `$l.', `$l^').

Named output

To specify the generic name of the file to produce, or how to access a printer (see section Your Printers).

Delegation

To specify the command associated to a delegation (see section Your Delegations).

Table of Content

To specify an index/table of content printed at the end of the job.

Variables in PostScript prologue

To allow the user to change some parameters to your prologues (see section Designing PostScript Prologues).

General Structure of the Meta Sequences

All format directives can also be given in format

escape width directive

where

escape

In general

`%'

meta sequences are related to general information (e.g., the current date, the user's name etc.),

`#'

meta sequences are related to the output (e.g., the output file name) or to the options you gave (e.g., the number of virtual pages etc.), or to special constructions (e.g., enumerations of the files, or tests etc.),

`$'

meta sequences are related to the current input file (e.g., its name, its current page number etc.),

`\'

introduces classical escaping sequences (e.g., `\n', `\f' etc.).

- width Specifies the width of the column to which the escape is printed. There are three forms for width

`+paddinginteger'

the result of the expansion is prefixed by the character padding so that the whole result is as long as integer. For instance `$+.10n' with a file name `$n'=`foo.c' gives `.....foo.c'.

If no padding is given, ` ' (white space) is used.

`-paddinginteger'

Idem as above, except that completion is done on the left: `$+.10n' gives `foo.c.....'.

`integer'

which is a short cut for `+integer'. For example, escape `$5P' will expand to something like ` 12'.

- directive See section Available Meta Sequences.

Available Meta Sequences

Supported meta-sequences are:

`\\'

character `\'

`\%'

character `%'

`\$'

character `$'

`\#'

character `#'

`#?cond|if_true|if_false|'

this may be used for conditional assignment. The separator (presented here as `|') may be any character. if_true and if_false may be defined exactly the same way as regular headers, included meta sequences and the `#?' construct.

The available tests are:

`#?1'

`#?2'

`#?3'

true if tag 1, 2 or 3 is not empty. See item `$t1' for explanation.

`#?d'

true if Duplex printing is requested (`-s2').

`#?j'

true if bordering is asked (`-j').

`#?l'

true if printing in landscape mode.

`#?o'

true if only one virtual page per page (i.e., `#v' is 1).

`#?p'

a page range has been specified (i.e., `#p' is not empty).

`#?q'

true if `a2ps` is in quiet mode.

`#?r'

true if major is rows (`--major=rows').

`#?v'

true if printing on the back side of the sheet (verso).

`#?V'

true if verbosity level includes the `tools' flag (See section Global Options option `--verbosity'.

- #!key|in|between| Used for enumerations. The separator (presented here as `|') may be any character. in and between are meta sequences.

The enumerations may be:

`#!$'

enumeration of the command line options. In this case in in never used, but is replaced by the arguments.

`#!f'

enumeration of the input files in the other they were given.

`#!F'

enumeration of the input files in the alphabetical order of their names.

`#!s'

enumeration of the files appearing in the current sheet.

For instance, the meta sequences `The files printed were: #!f|$n|, |.' evaluated with input `a2ps NEWS main.c -o foo.ps', gives `The files printed were: NEWS, main.c.'.

As an exception, `#!' meta sequences use the width as the maximum number of objects to enumerate if it is positive, e.g., `#10!f|$n|, |' lists only the ten first file names. If width is negative, then it does not enumerate the -width last objects (e.g., `#-1!f|$n|, |' lists all the files but the last).

- ${var} value of the environment variable var if defined, nothing otherwise.
- ${var:-word} if the environment variable var is defined, then its value, otherwise word.
- ${var:+word} if the environment variable var is defined, then word, otherwise nothing.
- $[num] value of the numth argument given on the command line. Note that $[0] is the name under which `a2ps` has been called.
- #{key} expansion of the macro meta sequence key if defined, nothing otherwise (see section Your Meta Sequences)
- #{key:-word} if the macro meta sequence var is defined, then its expansion, otherwise word.
- #{key:+word} if the macro meta sequence var is defined, then word, otherwise nothing.
- #. the extension corresponding to the current output language (e.g. `ps').
- %* current time in 24-hour format with seconds `hh:mm:ss'
- $* file modification time in 24-hour format with seconds `hh:mm:ss'
- $# the sequence number of the current input file
- %# the total number of files
- %a the localized equivalent for `Printed by User Name'
- %A the localized equivalent for `Printed by User Name from Host Name'
- %a{username} the localized equivalent for `Printed by username'
- %A{username@hostname} the localized equivalent for `Printed by username from hostname'.

These two are provided in the case `a2ps` would be used by the print service: since neither of the user name nor the host name can be known at the time the files reach `a2ps`, these options should be used.

- %c trailing component of the current working directory
- %C current time in `hh:mm:ss' format
- $C file modification time in `hh:mm:ss' format
- %d current working directory
- $d directory part of the current file (`.' if the directory part is empty).
- %D current date in `yy-mm-dd' format
- $D file modification date in `yy-mm-dd' format
- %D{string} format current date according to string with the `strftime(3)` function.

- $D{string} format file's last modification date according to string with the `strftime(3)` function
- %e current date in localized short format (e.g., `Jul 4, 76' in English, or `14 Juil 89' in French).
- $e file modification date in localized short format.
- %E current date in localized long format (e.g., `July 4, 76' in English, or `Samedi 14 Juillet 89' in French).
- $E file modification date in localized long format.
- $f full file name (with directory and suffix).
- \f character `\f' (`form feed`).
- #f0
- #f9 ten temporary file names. You can do anything you want with them, `a2ps` removes them at the end of the job. It is useful for the delegations (see section Your Delegations) and for the printer commands (see section Your Printers).
- %F current date in `dd.mm.yyyy' format.
- $F file modification date in `dd.mm.yyyy' format.
- #h medium height in PostScript points
- $l^ top most line number of the current page
- $l. current line number. To print the page number and the line interval in the right title, use `--right-title="$q:$l^-$l."'.
- $l# number of lines in the current file.
- %m the host name up to the first `.' character
- %M the full host name
- \n the character `\n' (`new line`).
- %n the user login name
- $n input file name without the directory part.
- %N the user's pw_gecos field up to the first `,' character (typically his/her full name).
- $N input file name without the directory, and without its suffix (e.g., on `` `foo.c' ``, it will produce `foo').
- #o name of the output, before substitution (i.e., argument of `-P', or of `-o').
- #O name of the output, after substitution. If output goes to a file, then the name of the file. If the output is a symbolic printer (see section Your Printers), the result of the evaluation. For instance, if the symbolic printer `file' is defined as `> $n.%.', then `#O' returns `foo.c.ps' when printing `` `foo.c' `` to PostScript. `#o' would have returned `file'.
- #p the range of the page to print from this page. For instance if the user asked `--pages=1-10,15', and the current page is 8, then `#p' evaluates to `1-3,8'.
- $p^ number of the first page of this file appearing on the current sheet. Note that `$p.', evaluated at the end of sheet, is also the number of the last page of this file appearing on this sheet.
- $p- interval of the page number of the current file appearing on the current sheet. It is the same as `$p^-$p.', if `$p^' and `$p.' are different, otherwise it is equal to `$p.'.
- %p. current page number

- $p. page number for this file
- %p# total number of pages printed
- $p# number of pages of the current file
- $p< number of the first page of the current file
- $p> number of the last page of the current file
- %q localized equivalent for `Page %p.'
- $q localized equivalent for `Page $p.'
- %Q localized equivalent for `Page %p./%p#'
- $Q localized equivalent for `Page $p./$p#'
- $s< number of the first sheet of the current file
- %s. current sheet number
- $s. sheet number for the current file
- $s> number of the last sheet of the current file
- %s# total number of sheets
- $s# number of sheets of the current file
- %t current time in 12-hour am/pm format
- $t file modification time in 12-hour am/pm format
- $t1
- $t2
- $t3 Content of tag 1, 2 and 3. Tags are pieces of text `a2ps` fetches in the files, according to the style. For instance, in `mail-folder` style, tag 1 is the title of the mail, and tag 2 its author.
- %T current time in 24-hour format `hh:mm'
- $T file modification time in 24-hour format `hh:mm'
- #v number of virtual sheets
- %V the version string of `a2ps`.
- #w medium width in PostScript points
- %W current date in `mm/dd/yy' format
- $W file modification date in `mm/dd/yy' format

Exit status

The following exit values are returned:

`0'
: `a2ps` terminated normally.

`1'
: an error occured.

`2'
: bad argument was given.

`3'

unknown language was given.

Library Files

To be general and to allow as much customization as possible, `a2ps` avoids to hard code its knowledge (encodings, PostScript routines, etc.), and tries to split it in various files. Hence it needs a path, i.e., a list of directories, in which it may find the files it needs.

The exact value of this library path is available by `a2ps --list=defaults'. Typically its value is:

```
gargantua ~ $ a2ps --list=defaults
Configuration status of a2ps 4.10
More stuff deleted here
Internals:
  verbosity level     = 2
  file command        = /usr/ucb/file -L
  temporary directory =
  library path        =
        /inf/soft/infthes/demaille/.a2ps
        /usr/local/share/a2ps/sheets
        /usr/local/share/a2ps/ps
        /usr/local/share/a2ps/encoding
        /usr/local/share/a2ps/afm
        /usr/local/share/a2ps/printers
        /usr/local/share/a2ps
```

You may change this default path through the configuration files (see section Your Library Path), and with the environment variable `A2PS_LIBRARY`.

If you plan to define yourself some files for `a2ps`, they should be in one of those directory.

Configuration Files

`a2ps` reads several files before the command line options. In the order, they are:

1. the system configuration file (usually `` `/usr/local/etc/a2ps.cfg' ``)
2. the user's home configuration file (`` `$HOME/.a2ps/a2psrc' ``)
3. the local file (`` `./.a2psrc' ``)

In these files, empty lines and lines starting with `#' are comments.

The other lines have all the following form:

```
Topic: Arguments
```

where Topic: is a keyword related to what you are customizing, and Arguments the customization. Arguments may be spread on several lines, provided that the last character of a line to continue is a `\'.

In the following sections, each Topic: is detailed.

Your Library Path

To define the default library path, you can use:

Configuration Setting: **LibraryPath:** *path*

Set the library path the path.

Configuration Setting: **AppendLibraryPath:** *path*

Add path at the end of the current library path.

Configuration Setting: **PrependLibraryPath:** *path*

Add path at the beginning of the current library path.

The environment variable `A2PS_LIBRARY` overrides these entries.

Note that for users configuration files, it is better not to set the library path, because the system's configuration has certainly been built to cope with your system's peculiarities. Use `AppendLibraryPath:' and `PrependLibraryPath:'.

Your Default Options

Configuration Setting: **Options:** *options+*

Give `a2ps` a list of command line options. options+ is any sequence of regular command line options (see section Command line options).

It is the correct way to define the default behavior you expect from `a2ps`. If for instance you want to use `Letter` as medium, then use:

```
Options: --medium=Letter
```

It is exactly the same as always giving `a2ps` the option `--medium=Letter' at run time.

The quoting mechanism is a bit painful: you need to quote the whole options+. For instance

```
Options: --right-title='Page $p' --center-title='Hello world'
```

is wrong. Write

```
Options: '--right-title=Page $p' '--center-title=Hello world'
```

Your Media

Configuration Setting: **Medium:** *name dimensions*

Define the medium name to have the dimensions (in PostScript points, i.e., 1/72 of inch).

There are two formats supported:

long

in which you must give both the size of the whole sheet, and the size of the printable area:

```
# A4 for Desk Jets
#       name      w       h      llx    lly    urx     ury
Medium: A4dj     595     842     24     50     571     818
```

where wxh are the dimension of the sheet, and the four other stand for lower left x and y, upper right x and y.

short

in which a surrounding margin of 24 points is used

```
# A4
#       name      w       h
Medium: A4       595     842
```

is the same as

```
# A4
#       name      w       h
Medium: A4       595     842     24     24     571     818
```

Your Printers

A general scheme is used, so that whatever the way you should address the printers on your system, the interface is still the same. Actually, the interface is so flexible, that you should understand `named destination' when we write `printer'.

Configuration Setting: **Printer:** *name [PPD-key] destination*

Specify the destination of the output when the option `-P name' is given. If PPD-key is given, declare the printer name to be described by the `PPD` file `` `PPD-key.ppd' ``.

The destination must be of one of the following forms:

`| command'

in which case the output is piped into command.

`> file'

in which case the output is saved into file.

Configuration Setting: **UnknownPrinter:** *[PPD-key] destination*

Specify the destination of the output when when the option `-P name' is given, but there is no `Printer:' entry for name.

Configuration Setting: **DefaultPrinter:** *[PPD-key] destination*

Specify the destination of the output when when the option `-d' (send to default output) is given.

Meta Sequences expansion is performed on destination (see section Meta Sequences). Recall that `#o' is evaluated to the destination name, i.e., the argument given to `-P'.

For instance

```
# My Default Printer is called dominique
DefaultPrinter: | lp -d dominique

# `a2ps foo.c -P bar' will pipe into `lp -d bar'
UnknownPrinter: | lp -d #o

# `a2ps -P foo' saves into the file `foo'
Printer: foo > foo.ps
Printer: wc | wc
Printer: lw | lp -d printer-with-a-rather-big-name

# E.g. `a2ps foo.c bar.h -P file' will save into `foo.c.ps'
Printer: file > $n.#.

# E.g. `a2ps foo.c bar.h -P home' will save into `foo.ps'
# in user's home
Printer: home > ${HOME}/$N.#.

# Here we address a printer which is not PostScript
Printer: deskj | gs -q -sDEVICE=ljet3d -sOutputFile=- - \
         | lpr -P laserwriter -h -l
```

MS-DOS users, and non-PostScript printer owners should take advantage in getting good configuration of these entries.

Your Shortcuts

You can define some kind of `Macro Options' which stands for a set of options.

Configuration Setting: **UserOption:** *shortcut options...*

Define the shortcut to be the list of options.... When `a2ps` is called with `--user-option=shortcut' (or `-=shortcut', consider the list of options....

Examples are

```
# This emulates a line printer: no features at all
```


```
# call a2ps -=lp to use it
UserOption: lp -1m --pretty-print=plain -B --borders=no

# When printing mail, I want to use the right style sheet with strong
# highlight level, and total stripping
UserOption: mail -Email -g --strip=3
```

Your PostScript magic number

`a2ps` produces full DSC compliant PostScript which is something that only PS wizards may understand(3). Adobe said

```
Thou shalt start your PostScript DSC compliant files with

%!PS-Adobe-3.0
```

The bad news is that some printers will reject this header. Then you may change this header without any worry since the PostScript produced by `a2ps` is also 100% PostScript level 1(4).

Configuration Setting: **OutputFirstLine:** *magic-number*

Specify the header of the produced PostScript file to be magic-number. Typical values include `%!PS-Adobe-2.0', or just `%!'.

Your Page Labels

In the PostScript file is dropped information on where sheets begin and end, so that post processing tools know where is the physical page 1, 2 etc. With this information can be also stored a label, i.e., a human readable text (typically the logical page numbers), which is for instance what `GhostView` shows as the list of page numbers.

`a2ps` lets you define what you want in this field.

Configuration Setting: **PageLabelFormat:** *format*

Specify the format to use to label the PostScript pages. format can use Meta Sequences (see section Meta Sequences). Two macro meta sequences are predefined for this: `#{pl.short}' and `#{pl.long}'.

Your Meta Sequences

Meta sequences are actually used so much, that it is pleasant to write sophisticated strings, that one would like not to repeat all the time. As with user short cuts, you can define macro meta sequences:

Configuration Setting: **MacroMetaSequence:** *key value*

Define the meta sequence `#{key}' to be a short cut for value. key must not have any character from `:()'.

As as example, here is a macro for `psnup`, which encloses all the option passing one would like.

Delegations are then easier to write:

```
MacroMetaSequence: psnup psnup -#v -q #?j|-d|| #?r||-c| -w#w -h#h
```

It is strongly suggested to follow a `.' (dot) separated hierarchy, starting with:

`pro'

for variables used in prologues (see section Designing PostScript Prologues). Moreover, specify the name of the prologue (e.g., `pro.matrix.gray').

`pl'

for page label formats. See section Your Page Labels the option `--page-label' in section Input Options.

`toc'

for toc formats. See the option `--toc' in section Input Options.

Your Delegations

There are some files you don't really want `a2ps` to pretty-print, typically page description files (e.g., PostScript files, `roff` files, etc.). You can let `a2ps` delegate the treatment of these files to other applications. The behavior at run time depends upon the option `--delegate' (see section Input Options).

Defining a Delegation

Configuration Setting: **Delegation:** *name in:out command*

Define the delegation name. It is to be applied upon files of type in when output type is out(5) thanks to command. Both in and out are `a2ps` type keys such as defined in `` `sheets.map' `` (see section The Entry in `` `sheets.map' ``).

command should produce the file on its standard output. Of course meta sequences substitution is performed on command (see section Meta Sequences). In particular, command should use the input file `$f'.

```
# In general, people don't want to pretty-print PostScript files.
# Pass the PostScript files to psnup
Delegation: PsNup ps:ps \
        psselect #?V||-q| -p#?p|#p|-| $f | \
        psnup -#v -q #?j|-d|| #?r||-c| -w#w -h#h
```

Advantage should be taken from the macro meta sequences, to encapsulate the peculiarities of the various programs.

```
# Passes the options to psnup.
# The files (in and out) are to be given
MacroMetaSequence: psnup psnup -#v #?V||-q| #?j|-d|| #?r||-c| -w#w -h#h
```


```
# Passes to psselect for PS page selection
MacroMetaSequence: psselect psselect #?V||-q| -p#?p|#p|-|

# In general, people don't want to pretty-print PostScript files.
# Pass the PostScript files to psnup
Delegation: PsNup ps:ps     #{psselect} $f | #{psnup}
```

Temporary file names (`#f0' to `#f9') are available for complex commands.

```
# Pass DVI files to dvips.
# A problem with dvips is that even on failure it dumps its prologue,
# hence it looks like a success (output is produced).
# To avoid that, we use an auxiliary file and a conditional call to
# psnup instead of piping.
Delegation: dvips dvi:ps    #{dvips} $f -o #f0 && #{psnup} #f0
```

Guide Line for Delegations

First of all, select carefully the applications you will use for the delegations. If a filter is known to cause problems, try to avoid it in delegations(6). As a thumb rule, you should check that the PostScript generating applications produce files that start by:

```
%!PS-Adobe-3.0
```

There are some services expected from the delegations. The delegations you may write should honor:

the input file
: available through the meta sequence `$f'.

the medium
: the dimension of the medium selected by the user are available through `#w' and `#h'.

the page layout
: the number of virtual pages is `#v'.

the page range
: the page range (in a form `1-2,4-6,10-' for instance) is available by `#p'.

the verbosity level
: please, do not make your delegations verbose by default. The silent mode should always be requested, unless `#?V' is set.

If ever you need several commands, do not use `;' to separate them, since it may prevent detection of failure. Use `&&' instead.

The slogan "*the sooner, the better*" should be applied here: in the processing chain, it is better to ask a service to the first application that supports it. An example will make it clear: when processing a `DVI` file, `dvips` knows better the page numbers than `psselect` would. So a `DVI` to PostScript delegation should ask the page selection (`#p') to `dvips`, instead of using `psselect` later in the chain.

There is a list of possible delegations, on the page `Delegations for a2ps'. If you know others, please let know.

Your Internal Details

There are settings that only meant for `a2ps` that you can tune by yourself.

Configuration Setting: **TemporaryDirectory:** *path*

Specify a directory in which the temporary files are put. Default is `` `/tmp' ``, or `` `/var/tmp' ``, or `` `$TMPDIR' `` according your environment.

Configuration Setting: **FileCommand:** *command*

The command to run to call `file(1)` on a file. If possible, make it follow the symbolic links.

Documentation Format

In various places a documentation can be given. Since some parts of this document and of web pages are extracted from documentations, some tags are needed to provide a better layout. The format is a mixture made out of Texinfo like commands, but built so that quick and easy processing can be made.

These tags are:

`code('text`)code'

Typeset text like a piece of code. This should be used for keys, variables, options etc. For instance the documentation of the `bold` prologue mentions the `bw` prologue:

```
FIXME: the doc of bold.pro, et matrix
```

`uref('link`)uref('text`)uref'

Specifies a hyper text link displayed as text.

`@example'

`@end example'

They must be alone on the line. The text between these tags is displayed in a code-like fonts. This should be used for including a piece of code. For instance, in the documentation of the `c` style sheet:

```
FIXME: c.ssh
```

`@itemize'

`@item' text

`@end itemize'

Typeset a list of items. The opening and closing tags must be alone on the line.

Map Files

Many things are defined through files. There is a general scheme to associate an object to the files to use: map files. They are typically used to:

- resolve aliases. For instance the ISO-8859-1 encoding is also called ISO Latin 1, or Latin 1 for short. The `encoding.map' file will map these three names to the same Encoding Description File.
- cope with broken files systems. For instance, the-one-you-know-I-don't-need-to-name cannot handle files named `Courier-BoldOblique.afm': it is the same as `Courier-Bold.afm'. The `fonts.map' file is here to associate a font file name to a font name.

The syntax of these files is:

- any empty line, or any line starting by a `#' is a comment.
- a line with the format

  ```
  ***                 path
  ```

  requests that the file designated by path be included at this point.
- any other line has the format

  ```
  key                 value
  ```

  meaning that when looking for key (e.g., name of a font, an encoding etc.), a2ps should use value (e.g., font file name, encoding description file name etc.).

The map files used in a2ps are:

`encoding.map'
: Resolving encodings aliases.

`fonts.map'
: Mapping font names to font file names.

`sheets.map'
: Rules to decide what style sheet to use.

Font Files

Even when a PostScript printer knows the fonts you want to use, using these fonts requires some description files.

Fonts Map File

See section Map Files, for a description of the map files. This file associates the font-key to a font name. For instance:

```
Courier                 pcrr
```


```
Courier-Bold            pcrb
Courier-BoldOblique     pcrbo
Courier-Oblique         pcrro
```

associates to font named `Courier`, the key `pcrr`. To be recognized, the font name must be exact: `courier` and `COURIER` are not admitted.

A small tool is provided with `a2ps`: `make_fonts_map.sh`. It collects the information on the font files `a2ps` can see, and generates a valid font map file, `` `fonts.map.new' ``.

Fonts Description Files

There are two kinds of data `a2ps` needs to use a font:

- the AFM file (`` `font-key.afm' ``), which describes the metrics information corresponding to font;
- in the case font is not known from the printer, the PFA or PFB file which is down loaded to the printer. These files are actually the PostScript programs which execution produces the characters to be drawn on the page, in this font.

Adding More Font Support

FIXME: A few words on how to make a2ps use more fonts. Don't forget some words on broken fonts such as those seen for Cyrillic support.

Style Sheet Files

The style sheets are defined in various files. See see section Pretty Printing for the structure of these files. As for most other features, there is main file, a road map, which defines in which condition a style sheet should be used (see section Map Files). This file is `` `sheets.map' ``.

It format is simple:

`pattern style-key'

if the current file name matches pattern, then select style style-key (i.e. file `` `style-key.ssh' ``).

`** pattern style-key'

if the result of a call to `file(1)` matches pattern, then select style style-key. Note that only the first two fields of the answer of `file(1)` are considered. E.g., in the following case

```
gargantua $ file a2ps
a2ps:   ELF 32-bit MSB executable SPARC Version 1,\
        dynamically linked, stripped
```

`ELF32-bit' will be matched against pattern.

The special style-key `binary' tells `a2ps` to consider that the file should not be printed, and will be ignored, unless option `--print-anyway' is given.

If a style name can't be found, the plain style is used.

Two things are to retain from this:

1. `a2ps` won't guess anything for a file given through the standard input.
2. if `file` is wrong on some files, `a2ps` may use bad style sheets. In this case, do try option `--guess', compare it with the output of `file`, and if the culprit is `file`, go and complain to your system administrator :-), or fix it by defining your own filename pattern matching rules.

Encodings

`a2ps` is trying to support the various usual encodings that its users use. This chapter presents what an encoding is, how the encodings support is handled within `a2ps`, and some encodings it supports.

What is an Encoding

This section is actually taken from the web pages of Alis Technologies inc.

Document encoding is the most important but also the most sensitive and explosive topic in Internet internationalization. It is an essential factor since most of the information distributed over the Internet is in text format. But the history of the Internet is such that the predominant - and in some cases the only possible - encoding is the very limited ASCII, which can represent only a handful of languages, only three of which are used to any great extent: English, Indonesian and Swahili.

All the other languages, spoken by more than 90% of the world's population, must fall back on other character sets. And there is a plethora of them, created over the years to satisfy writing constraints and constantly changing technological limitations. The ISO international character set registry contains only a small fraction; IBM's character registry is over three centimeters thick; Microsoft and Apple each have a bunch of their own, as do other software manufacturers and editors.

The problem is not that there are too few but rather too many choices, at least whenever Internet standards allow them. And the surplus is a real problem; if every Arabic user made his own choice among the three dozen or so codes available for this language, there is little likelihood that his "neighbor" would do the same and that they would thus be able to understand each other. This example is rather extreme, but it does illustrate the importance of standards in the area of internationalization. For a group of users sharing the same language to be able to communicate,

1. the code used in the shared document must always be identified (labeling)
2. they must agree on a small number of codes - only one, if possible (standards);
3. their software must recognize and process all codes (versatility)

Certain character sets stand out either because of their status as an official national or international standard, or simply because of their widespread use.

First off, there is the ISO 8859 standards series that standardize a dozen character sets that are useful for a large number of languages using the Latin, Cyrillic, Arabic, Greek and Hebrew alphabets. These standards have a limited range of application (8 bits per character, a maximum of 190 characters, no combining) but where they suffice (as they do for 10 of the 20 most widely used languages), they should be used on the Internet in preference to other codes. For all other languages, national standards should preferably be

chosen or, if none are available, a well-known and widely-used code should be the second choice.

Even when we limit ourselves to the most widely used standards, the overabundance remains considerable and this significantly complicates life for truly international software developers and users of several languages, especially when such languages can only be represented by a single code. It was to resolve this problem that both Unicode and the ISO 10646 International standard were created. Two standards? Oh no! Their designers soon realized the problem and were able to cooperate to the extent of making the character set repertoires and coding identical.

ISO 10646 (and Unicode) contain over 30,000 characters capable of representing most of the living languages within a single code. All of these characters, except for the *Han* (Chinese characters also used in Japanese and Korean), have a name. And there is still room to encode the missing languages as soon as enough of the necessary research is done. Unicode can be used to represent several languages, using different alphabets, within the same electronic document.

Encoding Files

The support of the encodings in `a2ps` is completely taken out of the code. That is to say, adding, removing or changing anything in its support for an encoding does not require programming, nor even being a programmer.

See section What is an Encoding, if you want to know more about this.

Encoding Map File

See section Map Files, for a description of the map files.

The meaningful lines of the `` `encoding.map' `` file have the form:

```
alias      key
iso-8859-1 latin1
latin1     latin1
l1         latin1
```

where

alias

specifies any name under which the encoding may be used. It influences the option `--encoding', but also the encodings dynamically required, as for instance in the `mail` style sheet (support for MIME).

When encoding is asked, the lower case version of encoding must be equal to alias.

key

specifies the prefix of the file describing the encoding (`` `key.edf' ``, section Encoding Description Files).

Encoding Description Files

The encoding description file describing the encoding key is named `key.edf'. It is subject to the same rules as any other a2ps file:

- please make the name portable: alpha-numerical, at most 8 characters,
- empty lines and lines starting by `#' are ignored.

The entries are

`Name:'

Specifies the full name of the encoding. Please, try to use the official name if there is one.

```
Name: ISO-8859-1
```

`Documentation/EndDocumentation'

Introduces the documentation on the encoding (see section Documentation Format). Typical informations expected are the other important names this encoding has, and the languages it covers.

```
Documentation
Also known as ISO Latin 1, or Latin 1.  It is a superset
of ASCII, and covers most West-European languages.
EndDocumentation
```

`Substitute:'

Introduces a font substitution. The most common fonts (e.g., Courier, Times-Roman...) do not support many encodings (for instance it does not support Latin 2). To avoid that Latin 2 users have to replace everywhere calls to Courier, a2ps allows to specify that whenever a font is called in an encoding, then another font should be used.

For instance in `latin2.edf' one can read:

```
# Fonts from Ogonkify offer full support of ISO Latin 2
Substitute: Courier               Courier-Ogonki
Substitute: Courier-Bold          Courier-Bold-Ogonki
Substitute: Courier-BoldOblique   Courier-BoldOblique-Ogonki
Substitute: Courier-Oblique       Courier-Oblique-Ogonki
```

`Default:'

Introduces the name of the font that should be used when a font (not substituted as per the previous item) is called but provides to poor a support of the encoding. The Courier equivalent is the best choice.

```
Default: Courier-Ogonki
```

`Vector:'

Introduces the PostScript encoding vector, that is a list of the 256 PostScript names of the characters. Note that only the printable characters are named in PostScript (e.g., `bell' in ASCII (^G) should not be named). The special name `.notdef' is to be used when the character is not printable.

Warning. Make sure to use real, official, PostScript names. Using names such as `c123' may be sign you use unusual names. On the other hand PostScript names such as `afii8879' are common.

Some Encodings

Most of the following information is a courtesy of Alis Technologies inc. and of Roman Czyborra's page about The ISO 8859 Alphabet Soup. See section What is an Encoding presents an instructive presentation of the encodings.

The known encodings are: Encoding: **ASCII** *(`ascii.edf')*

US-ASCII.

Encoding: **CP1250** *(`cp1250.edf')*

Microsoft's CP-1250 encoding (aka CeP).

Encoding: **HPRoman** *(`hp.edf')*

The 8 bits Roman encoding for HP.

Encoding: **IBMPC** *(`ibmpc.edf')*

Several characters may be missing, especially Greek letters and some mathematical symbols.

Encoding: **ISO-8859-1** *(`latin1.edf')*

The ISO-8859-1 character set, often simply referred to as Latin 1, can represent most Western European languages including: Albanian, Catalan, Danish, Dutch, English, Faroese, Finnish, French, Galician, German, Irish, Icelandic, Italian, Norwegian, Portuguese, Spanish and Swedish.

Encoding: **ISO-8859-10** *(`latin6.edf')*

Latin 6 (or ISO-8859-10) adds the last letters from Greenlandic and Lapp which were missing in Latin 4, and thereby covers all Scandinavia.

Support is provided thanks to Ogonkify.

Encoding: **ISO-8859-2** *(`latin2.edf')*

The Latin 2 character set supports the Slavic languages of Central Europe which use the Latin alphabet. The ISO-8859-2 set is used for the following languages: Czech, Croat, German, Hungarian, Polish, Romanian, Slovak and Slovenian.

Support is provided thanks to Ogonkify.

Encoding: **ISO-8859-3** *(`latin3.edf')*

This character set is used for Esperanto, Galician, Maltese and Turkish.

Support is provided thanks to Ogonkify.

Encoding: **ISO-8859-4** *(`latin4.edf')*

Some letters were added to the ISO-8859-4 to support languages such as Estonian, Latvian and Lithuar It is an incomplete precursor of the Latin 6 set.

Support is provided thanks to Ogonkify.

Encoding: **ISO-8859-5** *(`iso5.edf')*

The ISO-8859-5 set is used for various forms of the Cyrillic alphabet. It supports Bulgarian, Byelorussian, Macedonian, Serbian and Ukrainian.

The Cyrillic alphabet was created by St. Cyril in the 9th century from the upper case letters of the Greek alphabet. The more ancient Glagolithic (from the ancient Slav glagol, which means "word"), was created for certain dialects from the lower case Greek letters. These characters are still used by Dalmatian Catholics in their liturgical books. The kings of France were sworn in at Reims using a Gospel in Glagolithic characters attributed to St. Jerome.

Note that Russians seem to prefer the KOI8-R character set to the ISO set for computer purposes. KOI8-R is composed using the lower half (the first 128 characters) of the corresponding American ASCII character set.

Encoding: **ISO-8859-9** *(`latin5.edf')*

The ISO 8859-9 set, or Latin 5, replaces the rarely used Icelandic letters from Latin 1 with Turkish letters.

Support is provided thanks to Ogonkify.

Encoding: **KOI8** *(`koi8.edf')*

KOI-8 (+Ëë) is a subset of ISO-IR-111 that can be used in Serbia, Belarus etc.

Encoding: **Macintosh** *(`mac.edf')*

For the Macintosh encoding. The support is not sufficient, and a lot of characters may be missing at the end of the job (especially Greek letters).

Pretty Printing

The main feature of `a2ps` is its pretty-printing capabilities. Two different levels of pretty printing can be reached:

- basic (normal highlight level) in which what you print is what you wrote.
- string (heavy highlight level), in which in general, some keywords are replaced by a Symbol character which best represents them. For instance, in most languages `<=' and `>=' will be replaced by the corresponding single character from the font Symbol.

Note that the difference is up to the author of the style sheet.

Syntactic limits

`a2ps` is *not* a powerful syntactic pretty-printer: it just handles lexical structures, i.e., if in your favorite language

```
IF IF == THEN THEN THEN := ELSE ELSE ELSE := IF
```

is legal, then `a2ps` is not the tool you need. Indeed `a2ps` just looks for some keywords, or some sequences.

Known Style Sheets

Style Sheet: **68000** *(`68000.ssh')*

Style Sheet: **a2ps configuration file** *(`a2psrc.ssh')*

Style Sheet: **a2ps style sheet** *(`ssh.ssh')*

-g substitutes the LaTeX symbols.

Style Sheet: **Ada** *(`ada.ssh')*

Style Sheet: **Bourne Shell** *(`sh.ssh')*

Some classical program names, or builtin, are highlighted in the second level of pretty-printing.

Style Sheet: **C** *(`c.ssh')*

This style does not high light the function definitions. Another style which high lights them, GNUish C, is provided (gnuc.ssh). It works only if you respect some syntactic conventions.

Style Sheet: **C Shell** *(`csh.ssh')*

Written by Jim Diamond. Some classical program names, and/or builtins, are highlighted in the second level of pretty-printing.

Style Sheet: **C++** *(`cpp.ssh')*

Style Sheet: **CAML** *(`caml.ssh')*

This style should also suit other versions of ML (caml light, SML etc.)

Style Sheet: **ChangeLog** *(`chlog.ssh')*

This style covers the usual ChangeLog files.

Style Sheet: **Claire** *(`claire.ssh')*

Claire is a high-level functional and object-oriented language with advanced rule processing capabilities. It is intended to allow the programmer to express complex algorithms with fewer lines and in an elegant and readable manner.

To provide a high degree of expressivity, CLAIRE uses:

- A very rich type system including type intervals and second-order types (with dual static/dynamic typing),
- Parametric classes and methods,
- An object-oriented logic with set extensions,
- Dynamic versioning that supports easy exploration of search spaces.

To achieve its goal of readability, CLAIRE uses

- set-based programming with an intuitive syntax,
- simple-minded object-oriented programming,
- truly polymorphic and parametric functional programming,
- a powerful-yet-readable extension of DATALOG to express logical conditions,
- an entity-relation approach with explicit relations, inverses, unknown values and relational
- operations.

More information on claire can be found on claire home page.

Style Sheet: **Common Lisp** *(`clisp.ssh')*

Written by Juliusz Chroboczek.

Style Sheet: **Coq Vernacular** *(`coqv.ssh')*

This style is devoted to the Coq v 5.10 vernacular language.

Style Sheet: **Eiffel** *(`eiffel.ssh')*

Style Sheet: **Emacs Lisp** *(`elisp.ssh')*

Style Sheet: **Encapsulated PostScript** *(`eps.ssh')*

Illegal PostScript operators are highlighted as Errors.

Style Sheet: **Fortran** *(`fortran.ssh')*

Written by Denis Girou.

Style Sheet: **GNUish C** *(`gnuc.ssh')*

Declaration of functions are highlighted _only_ if you start the function name in the first column, and it is followed by an opening parenthesis. In other words, if you write

```
int main (void)
```

it won't work. Write:

```
int
main (void)
```

Style Sheet: **idl** *(`idl.ssh')*

Written by Robert S. Mallozzi. Style sheet for IDL (Interactive Data Language). http://www.rsinc.com

Style Sheet: **InstallShield 5** *(`is5rul.ssh')*

Written by Alex. InstallShield5 _TM_ RUL script.

Style Sheet: **Java** *(`java.ssh')*

Documentation comments are mapped to strong comments, and any other comment is plain comment.

Style Sheet: **LACE** *(`lace.ssh')*

This is meant for the Eiffel equivalent of the Makefiles.

Style Sheet: **Lex** *(`lex.ssh')*

In addition to the C constructs, it highlights the declaration of states, and some special `%' commands.

Style Sheet: **Mail Folder** *(`mail.ssh')*

To use from elm and others, it is better to specify -g -Email, since the file sent to printer is no longer truly a mailfolder. This style also suits to news. --strip options are also useful (they strip "useless" headers).

Whenever the changes of encoding are clear, a2ps sets itself the encoding for the parts concerned.

Tag 1 is the subject, and Tag 2 the author of the mail/news.

Note: This style sheet is _very_ difficult to write. Please don't report behavior you don't like. Just send me improvements, or write a Bison parser for mails.

Style Sheet: **Makefile** *(`make.ssh')*

Special tokens, and non terminal declarations are highlighted.

Style Sheet: **MATLAB 4** *(`matlab4.ssh')*

Written by Marco De la Cruz. Note that comments in the code should have a space after the %

Style Sheet: **Modula 2** *(`modula2.ssh')*

Written by Peter Bartke.

Style Sheet: **Modula 3** *(`modula3.ssh')*

Style Sheet: **o2c** *(`o2c.ssh')*

Style Sheet: **Oberon** *(`oberon.ssh')*

Style Sheet: **Objective C** *(`objc.ssh')*

Written by Paul Shum.

Style Sheet: **Octave** *(`octave.ssh')*

Written by C.P. Earls.

Style Sheet: **Oracle parameter file** *(`initora.ssh')*

Written by Pierre Mareschal. For init.ora parameter files.

Style Sheet: **Oracle PL/SQL** *(`plsql.ssh')*

Written by Pierre Mareschal. This style is to be checked.

Style Sheet: **Oracle SQL** *(`sql.ssh')*

Written by Pierre Mareschal. a2ps-sql Pretty Printer Version 1.0.0 beta - 18-MAR-97 For comments, support for -- /*..*/ and //. This style is to be checked.

Style Sheet: **Oracle SQL-PL/SQL-SQL*Plus** *(`oracle.ssh')*

Written by Pierre Mareschal. 18-MAR-97 For comments, support for -- /*..*/ and //. This style is to be checked.

Style Sheet: **Pascal** *(`pascal.ssh')*

The standard Pascal is covered by this style. But some extension have been added too, hence modern Pascal programs should be correctly handled. Heavy highlighting maps mathematical symbols to their typographic equivalents.

Style Sheet: **Perl** *(`perl.ssh')*

Written by Denis Girou.

Style Sheet: **Perl (Old)** *(`oldperl.ssh')*

Written by Denis Girou.

Style Sheet: **PostScript** *(`ps.ssh')*

Only some keywords are high lighted, because otherwise listings are quickly becoming a big bold spot.

Style Sheet: **PostScript Printer Description** *(`ppd.ssh')*

The support is not yet complete. Tag 1 is the printer ModelName.

Style Sheet: **PreScript** *(`pre.ssh')*

This style defines commands in the canonic syntax of a2ps. It is meant to be used either as an input language, and to highlight the table of contents etc.

It can be a good choice of destination language for people who want to produce text to print (e.g. pretty-printing, automated documentation etc.) but who definitely do not want to learn PostScript, nor to require the use of LaTeX.

Style Sheet: **PreTeX** *(`pretex.ssh')*

This style sheets provides LaTeX-like commands to format text. It is an alternative to the PreScript style sheet, in which formating commands are specified in a more a2ps related syntax.

It provides by the use of LaTeX like commands, a way to describe the pages that this program should produce.

Style Sheet: **Prolog** *(`prolog.ssh')*

Under construction.

Style Sheet: **Promela** *(`promela.ssh')*

There is no way for this program to highlight send and receive primitives.

Style Sheet: **Python** *(`python.ssh')*

Style Sheet: **Reference Card** *(`card.ssh')*

This style sheet is meant to process help messages generated by Unix applications. It highlights the options (-short or --long), and their arguments. Normal use of this style sheet is through the shell script card (part of the a2ps package), but a typical hand-driven use is:

```
program --help | a2ps -Ecard
```

Style Sheet: **Sather** *(`sather.ssh')*

Style Sheet: **Scheme** *(`scheme.ssh')*

This style sheet is looking for a maintainer and/or comments.

Style Sheet: **SDL-88** *(`sdl88.ssh')*

Written by Jean-Philippe Cottin. --strip-level=2 is very useful: it cancels the graphical information left by graphic editors. Only the pure specification is then printed.

Style Sheet: **SQL 92** *(`sql92.ssh')*

Written by Pierre Mareschal. 18-MAR-97 This style is to be checked.

Style Sheet: **Symbols** *(`symbols.ssh')*

This style sheet should be a precursor for any style sheet which uses LaTeX like symbols.

Style Sheet: **TeX** *(`tex.ssh')*

Written by Denis Girou. This is the style for (La)TeX files. It's mainly useful for people who develop (La)TeX packages. With -g, common mathematical symbols are represented graphically.

Style Sheet: **Texinfo** *(`texinfo.ssh')*

Definitely being built. With the option -g, the nodes will be printed on separated pages which title is the name of the node.

Style Sheet: **TeXScript** *(`texscript.ssh')*

TeXScript is the new name of what used to be called PreScript. New PreScript has pure a2ps names, PreTeX has pure TeX names, and TeXScript mixes both.

Style Sheet: **tk** *(`tk.ssh')*

Since everything, or almost, is a string, what is printed is not always what you would like.

Style Sheet: **Tool Command Language** *(`tcl.ssh')*

Since everything, or almost, is a string, what is printed is not always what you would like.

Style Sheet: **Unity** *(`unity.ssh')*

Written by Jean-Philippe Cottin. The graphic conversion of the symbols (option -g) is nice.

Style Sheet: **VERILOG** *(`verilog.ssh')*

Written by Edward Arthur. This style is devoted to the VERILOG hardware description language.

Style Sheet: **VHDL** *(`vhdl.ssh')*

Written by Thomas Parmelan. Non-textual operators are not highlighted. Some logical operators are printed as graphical symbols in the second level of pretty-printing.

Style Sheet: **VRML** *(`vrml.ssh')*

Written by Nadine Richard. According to Grammar Definition Version 2.0 ISO/IEC CD 14772

Style Sheet: **Yacc** *(`yacc.ssh')*

Special tokens, and non terminal declarations are highlighted.

Style Sheet: **zsh** *(`zsh.ssh')*

Type Setting Style Sheets

This section presents a few style sheets that define page description languages (compared to most other style sheet meant to pretty print source files).

Symbol

The style sheet `Symbol` introduces easy to type keywords to obtain the special characters of the PostScript font `Symbol`. The keywords are named to provide a LaTeX taste. These keywords are also the names used when designing a style sheet, hence to get the full list, see section A Bit of Syntax.

If you want to know the correspondence, it is suggested to print the style sheet file of `Symbol`:

```
a2ps -g symbol.ssh
```

PreScript

`PreScript` has been designed in conjunction with `a2ps`. Since bold sequences, special characters etc. were implemented in `a2ps`, we thought it would be good to allow direct access to those features: `PreScript` became an input language for `a2ps`, where special font treatments are specified in an `ssh` syntax (see section Style Sheets Implementation).

The main advantages for using `PreScript` are:

- it is fairly simple,
- `a2ps` is small and easy to install, hence it is available on every UNIX platform.

It can be a good candidate for generation of PostScript output (syntactic pretty-printers, generation of various reports etc.).

Syntax

Every command name begins with a backslash (`\'). If the command uses an argument, it is given between curly braces with no spaces between the command name and the argument.

The main limit on `PreScript` is that no command can be used inside another command. For instance the following line will be badly interpreted by `a2ps`:

```
\Keyword{Problems using \keyword{recursive \copyright} calls}
```

The correct way to write this in `PreScript` is

```
\Keyword{Problems using} \keyword{recursive} \copyright \Keyword{calls}.
```

Everything from an unquoted % to the end of line is ignored (comments).

PreScript Commands

These commands required arguments.

`\keyword{text}'

`\Keyword{text}'

Highlight lightly/strongly the given text. Should be used only for a couple of adjacent words.

`\comment{text}'

`\Comment{text}'

The text is given a special face. The text may be removed if option `--strip' is used.

`\label{text}'

`\Label{text}'

text should be considered as a definition, or an important point in the structure of the whole text.

`\string{text}'

Write text with string's face (e.g., in font Times).

`\error{text}'

Write text with error's face (generally a very different face, so that you see immediately).

`\symbol{text}'

text is written in the PostScript symbol font. This feature is not compatible with LaTeX. It is recommended, when possible, to use the special keywords denoting symbols, which are compatible with LaTeX (see section Symbol).

`\header{text}'

`\footer{text}'

Use text as header (footer) for the current page. If several headers or footers are defined on the same page, the last one is taken into account.

`\encoding{key}'

Change dynamically the current encoding. After this command, the text is printed using the encoding corresponding to key.

Examples

`PreScript` and `a2ps` can be used for one-the-fly formating. For instance, on the `` `passwd' `` file:

```
ypcat passwd |
 awk -F: \
   '{print "\Keyword{" $5 "} (" $1 ") \rightarrow\keyword{" $7 "}"}'\
 | a2ps -Epre -P
```

PreTeX

The aim of the PreTeX style sheet is to provide something similar to `PreScript`, but with a more LaTeX like syntax.

Special characters

`$' is ignored in `PreTeX` for compatibility with LaTeX, and `%' introduces a comment. Hence they are the only symbols which have to be quoted by a `\'. The following characters should also be quoted to produce good LaTeX files, but are accepted by `PreScript`: `_', `&', `#'.

Note that *inside a command*, like `\textbf`, the quotation mechanism does not work in `PreScript` (`\textrm{#$%}` writes `#$%') though LaTeX still requires quotation. Hence whenever special characters or symbols are introduced, they should be at the outer most level.

PreTeX Commands

These commands required arguments.

`\section{Title}'

`\subsection{Title}'

`\subsubsection{Title}.'

Used to specify the title of a section, subsection or subsubsection.

`\textbf{text}'

`\textit{text}'

`\textbi{text}'

`\textrm{text}'

write text in bold, italic, bold-italic, Times. Default font is Courier.

`\textsy{text}'

text is written in the PostScript symbol font. This feature is not compatible with LaTeX. It is recommended, when possible, to use the special keywords denoting symbols, which are compatible with LaTeX (See the style sheet `Symbol`).

`\header{text}'

`\footer{text}'

Use text as header (footer) for the current page. If several headers or footers are defined on the same page, the last one is taken into account.

`\verb+text+'

Quote text so that no special sequence will be interpreted. In `\verb+quoted string+' `+' can be any symbol in `+', `!', `|', `#', `='.

`\begin{document}'

`\end{document}'

`\begin{itemize}'

`\end{itemize}'

`\begin{enumerate}'

`\end{enumerate}'

`\begin{description}'

`\end{description}'

These commands are legal in LaTeX but have no sense in PreTeX. Hence there are simply ignored and not printed (if immediately followed by an end-of-line).

Differences with LaTeX

The following symbols, inherited from the style sheet `Symbol`, are not supported by LaTeX:

`\Alpha', `\apple', `\Beta', `\carriagereturn', `\Chi', `\Epsilon', `\Eta', `\florin', `\Iota', `\Kappa', `\Mu', `\Nu', `\Omicron', `\omicron', `\radicalex', `\register', `\Rho', `\suchthat', `\Tau', `\therefore', `\trademark', `\varUpsilon', `\Zeta'.

LaTeX is more demanding about special symbols. Most of them must be in so-called math mode, which means that the command must be inside `$' signs. For instance, though

```
If \forall x \in E, x \in F then E \subseteq F.
```

is perfectly legal in PreTeX, it should be written


```
If $\forall x \in E, x \in F$ then $E \subseteq F$.
```

for LaTeX. Since in PreTeX every `$' is discarded (unless quoted by a `\'), the second form is also admitted.

TeXScript

`TeXScript` is a replacement of the old version of `PreScript`: it combines both the `a2ps`-like and the LaTeX-like syntaxes through inheritance of both `PreScript` and `PreTeX`.

In addition it provides commands meant to ease processing of file for `a2ps` by LaTeX.

Everything between `%%TeXScript:skip' and `%%TeXScript:piks' will be ignored in `TeXScript`, so that there can be inserted command definitions for LaTeX exclusively.

The commands `\textbi' (for bold-italic) and `\textsy' (for symbol) do not exist in LaTeX. They should be defined in the preamble:

```
%%TeXScript:skip
\newcommand{\textbi}[1]{\textbf{\textit{#1}}}
\newcommand{\textsy}[1]{#1}
%%TeXScript:piks
```

There is no way in TeXScript to get an automatic numbering. There is no equivalent to the LaTeX environment `enumerate`. But every command beginning by `\text` is doubled by a command beginning by `\magic'. `a2ps` behaves the same way on both families of commands. Hence, if one specifies that arguments of those functions should be ignored in the preamble of the LaTeX document, the numbering is emulated. For instance

```
\begin{enumerate}
\magicbf{1.}\item First line
\magicbf{2.}\item Second line
\end{enumerate}
```

will be treated the same way both in TeXScript and LaTeX.

`\header' and `\footer', are not understood by LaTeX.

Faces

A face is an attribute given to a piece of text, which specifies how it should look like. Since `a2ps` is devoted to pretty-printing source files, the faces it uses are related to the syntactic entities that can be encountered in a file.

The faces `a2ps` uses are:

`Plain'

This corresponds to the text body.

`Keyword'

`Keyword_strong'

These are related to the keywords that may appear in a text.

`Comment'

`Comment_strong'

These are related to comments in the text. Remember that comments should be considered as non essential ("*Aaaeaaarg*" says the programmer); indeed, the user might suppress the comments thanks (?) to the option `--strip-level'. Hence, **never** use these faces just because you think they look better on, say, strings.

`Label'

`Label_strong'

These are used when a point of extreme importance, or a sectioning point, is met. Typically, functions declarations etc.

`String'

Used mainly for string and character literals.

`Error'

Used to underline the presence of an error. For instance in Encapsulated PostScript, some PostScript operators are forbidden: they are underlined as errors.

Actually, there is also the face `Symbol', but this one is particular: it is not legal changing its font.

Style sheets semantics

`a2ps` pretty prints a source file thanks to style sheets, one per language. In the following is described how the style sheets are defined. You may skip this section if you don't care how `a2ps` does this, and if you don't expect to implement new styles.

Name and key

Every style sheet has both a key, and a name. The name can be clean and beautiful, with any character you might want. The key is in fact the prefix part of the file name, and is alpha-numerical, lower case, and less than 8 characters long.

Anywhere `a2ps` needs to recognize a style sheet by a name, **it uses the key** (in the `` `sheets.map' `` file, with the option `-E', etc.).

As an example, C++ is implemented in a file called `` `cpp.ssh' ``, in which the name is declared to be `C++'.

The rationale is that not every system accepts any character in the file name (e.g., no `+' in MS-DOS). Moreover, it allows to make symbolic links on the ssh files (e.g., `ln -s cpp.ssh c++.ssh' let's you use `-E c++').

Comments

ssh files can include the name of its author, a version number, a documentation note and a requirement on the version of `a2ps`. For instance, if a style sheet requires `a2ps` version 4.9.6, then `a2ps` version 4.9.5 will reject it.

Alphabets

`a2ps` needs to know the beginning and the end of a word, especially keywords. Hence it needs two alphabets: the first one specifying by which letters an identifier can begin, and the second one for the rest of the word. If you prefer, a keyword starts with a character belonging to the first alphabet, and a character not pertaining to the second is a separator.

Case sensitivity

If the style is case insensitive, then matching is case insensitive (keywords, operators and sequences).

P-Rules

A P-rule (Pretty printing rule), or rule for short, is a structure which consists of two items:

lhs

left-hand side

its source string, with which the source file is compared;

rhs

right hand side

a list of faced strings which will replace the text matched in the pretty-printed output. A faced string is composed of

a string, or a reference to a part of the source string (see section 'Back-reference Operator' in Regex manual)

the face to use to print it

Just a short example: `(foo, bar, Keyword_strong)' as a rule means that every input occurrence of `foo' will be replaced by `bar', written with the `Keyword_strong` face.

If the destination string is empty, then `a2ps` will use the source string. This is different from giving the source string as a destination string if the case is different. An example will make it fairly clear.

Let `foobar` be a case insensitive style sheet including the rules `(foo, "", Keyword)' and `(bar, bar, Keyword)'. Then, on the input `FOO BAR', `a2ps` will produce `FOO bar' in `Keyword`.

`a2ps` implements two different ways to match a string. The difference comes from that some keywords are sensitive to the delimiters around them (such as `unsigned' and `int' in `C`, which are definitely not the same thing as `unsignedint'), and others not (in `C`, `!=' is "different from" both in `a != b' and `a!=b').

The first ones are called keywords in `a2ps` jargon, and the seconds are operators. Operators are matched anywhere they appear, while keywords need to have separators around them (see section Alphabets).

Let us give a more complicated example: that of the `Yacc` rules. A rule in `Yacc` is of the form:

```
a_rule : part1 part2 ;
```

Suppose you want to highlight these rules. To recognize them, you will write a regular expression specifying that:

1. it must start at the beginning of the line,
2. then there is string composed of symbols, which is what you want to highlight,
3. and a colon, which can be preceded by blank characters.

The regexp you want is: `/^[a-zA-Z0-9_]*[\t ]*:/'. But with the rule `/^[a-zA-Z0-9_]*[\t]*:/, "", Label_strong' the blanks and the colon are highlighted too. Hence you need to specify some parts in the regexp (see section 'Back-reference Operator' in Regex manual), and use a longer list of destination strings. The correct rule is

```
(/^\\([a-zA-Z0-9_]*\\)\\([\t ]*:\\)/, \1 Label_strong, \2 Plain)
```

Since it is a bit painful to read, regexps can be spread upon several lines. It is strongly suggested to break them by groups, and to document the group:

```
(/^\\([a-zA-Z0-9_]*\\)/    # \1. Name of the rule
 /\\([\t ]*:\\)/           # \2. Trailing space and colon
 \1 Label_strong, \2 Plain)
```

Sequences

A sequence is a string between two markers, along with a list of exceptions. A marker is a fixed string. Typical examples are comments, string (with usually `"' as opening and closing markers, and `\\' and `\"' as exceptions) etc. Three faces are used: one for the initial marker, one for the core of the sequence, and a last one for the final maker.

Optional entries

There are two levels of pretty-printing encoded in the style sheets. By default, `a2ps` uses the first level, called normal, unless the option `-g' is specified, in which case, heavy highlighting is invoked, i.e., optional keywords, operators and sequences are considered.

Style Sheets Implementation

A Bit of Syntax

Here are the lexical rules underlying the style sheet language:

- the separators are white space, form feed, new line, and tab.
- `#' introduces a comment, ended at the end of the line.

- special characters are the separators, plus `#', `"', `,', `(', `)', `+' and `/'. Any other character is a re character.
- the list of the structuring keywords is

  `alphabet`, `alphabets`, `are`, `case`, `documentation`, `end`, `exceptions`, `first`, `in`, `insensitive`, `is`, `keywords`, `operators`, `optional`, `second`, `sensitive`, `sequences`, `style`
- the list of the keywords designating faces is

  `Comment`, `Comment_strong`, `Encoding`, `Error`, `Index1`, `Index2`, `Index3`, `Index4`, `Invisible`, `Keyword`, `Keyword_strong`, `Label`, `Label_strong`, `Plain`, `String`, `Symbol`, `Tag1`, `Tag2`, `Tag3`, `Tag4`
- the list of keywords designating special sequences is

  `C-char`, `C-string`
- the list of keywords representing special characters is

  `---`, `\Alpha`, `\Beta`, `\Chi`, `\Delta`, `\Downarrow`, `\Epsilon`, `\Eta`, `\Gamma`, `\Im`, `\Iota`, `\Kappa`, `\Lambda`, `\Leftarrow`, `\Leftrightarrow`, `\Mu`, `\Nu`, `\Omega`, `\Omicron`, `\Phi`, `\Pi`, `\Psi`, `\Re`, `\Rho`, `\Rightarrow`, `\Sigma`, `\Tau`, `\Theta`, `\Uparrow`, `\Upsilon`, `\Xi`, `\Zeta`, `\aleph`, `\alpha`, `\angle`, `\approx`, `\beta`, `\bullet`, `\cap`, `\carriagereturn`, `\cdot`, `\chi`, `\circ`, `\clubsuit`, `\cong`, `\copyright`, `\cup`, `\delta`, `\diamondsuit`, `\div`, `\downarrow`, `\emptyset`, `\epsilon`, `\equiv`, `\eta`, `\exists`, `\florin`, `\forall`, `\gamma`, `\geq`, `\heartsuit`, `\in`, `\infty`, `\int`, `\iota`, `\kappa`, `\lambda`, `\langle`, `\lceil`, `\ldots`, `\leftarrow`, `\leftrightarrow`, `\leq`, `\lfloor`, `\mu`, `\nabla`, `\neq`, `\not`, `\not\in`, `\not\subset`, `\nu`, `\omega`, `\omicron`, `\oplus`, `\otimes`, `\partial`, `\perp`, `\phi`, `\pi`, `\pm`, `\prime`, `\prod`, `\propto`, `\psi`, `\radicalex`, `\rangle`, `\rceil`, `\register`, `\rfloor`, `\rho`, `\rightarrow`, `\sigma`, `\sim`, `\spadesuit`, `\subset`, `\subseteq`, `\suchthat`, `\sum`, `\supset`, `\supseteq`, `\surd`, `\tau`, `\theta`, `\therefore`, `\times`, `\trademark`, `\uparrow`, `\upsilon`, `\varUpsilon`, `\varcopyright`, `\vardiamondsuit`, `\varphi`, `\varpi`, `\varregister`, `\varsigma`, `\vartheta`, `\vartrademark`, `\vee`, `\wedge`, `\wp`, `\xi`, `\zeta`
- a string starts and finishes with `"', and may contain anything. Regular `C` escaping mechanism is used.
- a regular expression starts and finishes with `/', and may contain anything. Regular `C` escaping mechanism is used. Regexps can be split in several parts, *à la* C strings (i.e., `/part 1/ /part 2/').
- any sequence of regular characters which is not a keyword, is a string (consider this as a shortcut, avoiding extraneous `"').

Style Sheet Header

The definition of the name of the style sheet is:

```
style name is
```


```
  # body of the style sheet
end style
```

The following constructions are optional:

`version`

To define the version number of the style sheet

```
version is version-number
```

`written`

To define the author(s).

```
written by authors
```

Giving your email is useful for bug reports about style sheets.

```
written by "Some Body <Some.Body@some.whe.re>"
```

`requires`

To specify the version of `a2ps` it requires. `a2ps` won't accept a file which requires a higher version number than its own.

```
requires a2ps a2ps-version-number
```

`documentation`

To leave extra comments people should read.

```
documentation is
   strings
end documentation
```

strings may be a list of strings, without comas, in which case new lines are automatically inserted between each item. See section Documentation Format for details on the format.

Please, write useful comments, not `This style is devoted to C files', since the name is here for that, nor `Report errors to mail@me.somewhere', since `written by` is there for that.

```
documentation is
    "Not all the keywords are used, to avoid too much"
    "bolding. Heavy highlighting (code(-g)code), covers"
    "the whole language."
end documentation
```

Syntax of the Words

There are two things `a2ps` needs to know: what is symbol consistent, and whether the style is case insensitive.

`alphabet`

To define two different alphabets, use

```
first alphabet is string
second alphabet is string
```

If both are identical, you may use the shortcut

```
alphabets are string
```

The default alphabets are

```
first alphabet is
  "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_"
second alphabet is
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_\
0123456789"
```

Note that it is on purpose that no characters interval are used.

`case`

```
case insensitive        # e.g., C, C++ etc.
case sensitive          # e.g., Perl, Sather, Java etc.
```

The default is `case insensitive`.

Inheriting from Other Style Sheets

It is possible to extend an existing style. The syntax is:

```
ancestors are
   ancestor_1[, ancestor_2...]
end ancestors
```

where ancestor1 etc. are style sheet keys.

For semantics, the rules are the following:

- the ancestors are read in order;
- the definition of the current style is read last;
- it is always the last item read which wins (last defined alphabets, case sensitivity, keywords, operators and sequences).

As an example, both `C++` and `Objective C` style sheets extend the `C` style sheet:

```
style "Objective C" is
#[...]
ancestors are
```


```
   c
end ancestors
#[...]
end style
```

To the biggest surprise of the author, mutually dependent style sheets do work!

Syntax for the P-Rules

See section P-Rules, for the definition of P-rule.

Because of various short cuts, there are many ways to declare a rule:

```
rules     ::= rule_1 `,' rule_2...
rule      ::= `(' lhs rhs `)'
            | lhs srhs ;
lhs       ::= string | regex ;
rhs       ::= srhs `,' ...
srhs      ::= latex-keyword | expansion face
expansion ::= string | `\'num | <nothing>;
face      ::= face-keyword | <nothing>;
```

The rules are the following:

- If the left-hand side (lhs) is a regular expression, then it is compiled with the following syntax bits:

  ```
  #define RE_SYNTAX_A2PS \
     (RE_SYNTAX_EMACS | RE_CHAR_CLASSES | RE_INTERVALS)
  ```

  See section 'Regular Expression Syntax' in Regex manual, for detailed description of the regular expressions used.
- If no expansion is specified, then the matched string is used. For instance `(/fo*/, NULL, Keyword)' applied on the source `"fooooo"' produces `"fooooo"' in `Keyword`.
- If no face is given, then
  - if the context defines the default face, then this face is used;
  - if no default face is given, `PLAIN` is used.

Declaring the keywords and the operators

Basically, keywords and operators are lists of rules. The syntax is:

```
keywords are
  rules
end keywords
```

or


```
keywords in face-keyword are
  rules
end keywords
```

in which case the default face is set to face-keyword.

As an example:

```
keywords in Keyword_strong are
  /foo*/,
  "bar" "BAR" Keyword,
  -> \rightarrow
end keywords
```

is valid.

The syntax for the operators is the same, and both constructs can be qualified with an `optional` flag, in which case they are taken into account in the heavy highlighting mode (see section Pretty Printing Options).

This is an extract of the `C` style sheet:

```
optional operators are
   -> \rightarrow,
   && \wedge,
   || \vee,
   != \neq,
   == \equiv,
   # We need to protect these, so that <= is not replaced in <<=
   <<=,
   >>=,
   <= \leq,
   >= \geq,
   ! \not
end operators
```

Note how `<<=' and `>>=' are protected (there are defined to be written as is when met in the source). This is to prevent the two last characters of `<<=' from being converted into a `less or equal' sign.

The order in which you define the elements of a category (but the sequences) does not matter. But since `a2ps` sorts them at run time, it may save time if the alphabetical `C`-order is more or less followed.

You should be aware that when declaring a keyword with a regular expression as lhs, then `a2ps` automatically makes this expression matching only if there are no character of the first alphabet both just before, and just after the string.

In term of implementation, it means that

```
keywords are
  /foo\\|bar/
```


```
end keywords
```

is exactly the same as

```
operators are
  /\\b\\(foo\\|bar\\)\\b/
end operators
```

This can cause problems if you use anchors (e.g. $, or ^) in keywords: the matcher will be broken. In this particular case, define your keywords as operators, taking care of the `\\b' by yourself.

See section 'Match-word-boundary Operator' in Regex manual, for details on `\b'.

Declaring the sequences

Sequences admit several declarations too:

```
sequences      ::= sequences are
                     sequence_1 `,' sequence_2...
                   end sequences
sequence       ::= rule in_face close_opt exceptions_opt
                 | C-string
                 | C-char
                 ;
close_opt      ::= rule
                 | closers are
                     rules
                   end closers
                 | <nothing>
                 ;
exceptions_opt ::= exceptions are
                     rules
                   end exceptions
                 | <nothing>
                 ;
```

The rules are:

- The default face is then in_face.
- If no closing rule is given, `"\n"' (i.e., end-of-line) is used.

As a first example, the definition of `C-string` is:

```
sequences are
 "\"" Plain String "\"" Plain
     exceptions are
        "\\\\", "\\\""
     end exceptions
```


```
end sequences
```

The following example comes from `ssh.ssh', the style sheet for style sheet files, in which there are two kinds of pseudo-strings: the strings (`"example"'), and the regular expressions (`/example/'). We do not want the content of the pseudo-strings in the face `String'.

```
sequences are
  # The comments
  "#" Comment,

  # The name of the style sheet
  "style " Keyword_strong (Label + Index1) " is" Keyword_strong,

  # Strings are exactly the C-strings, though we don't want to
  # have them in the "string" face
  "\"" Plain "\""
     exceptions are
        "\\\\", "\\\""
     end exceptions,

  # Regexps
  "/" Plain "/"
     exceptions are
        "\\\\", "\\\/"
     end exceptions

end sequences
```

The order between sequences does matter. For instance in Java, `/**' introduces strong comments, and `/*' comments. `/**' *must* be declared before `/*', or it will be hidden.

There are actually some sequences that could have been implemented as operators with a specific regular expression (that goes up to the closer). Nevertheless be aware of a big difference: regular expression are applied to a single line of the source file, hence, they cannot match on several lines. For instance, the `C' comments,

```
/*
 * a comment
 */
```

cannot be implemented with operators, though `C++' comments can:

```
//
// a comment
//
```

A tutorial on style sheets

In this section a simple example of style sheet is entirely covered: that of `ChangeLog' files.

`ChangeLog' files are some kind of memory of changes done to files, so that various programmers can understand what happened to the sources. This helps a lot, for instance, in guessing what recent changes may have introduced new bugs.

Example and syntax

First of all, here is a sample of a `ChangeLog' file, taken from the `misc/' directory of the original a2ps package:

```
Sun Apr 27 14:29:22 1997  Akim Demaille  <demaille@inf.enst.fr>

        * base.ps: Merged in color.ps, since now a lot is
          common [added box and underline features].

Fri Apr 25 14:05:20 1997  Akim Demaille  <demaille@inf.enst.fr>

        * color.ps: Added box and underline routines.

Mon Mar 17 20:39:11 1997  Akim Demaille  <demaille@gargantua.enst.fr>

        * base.ps: Got rid of CourierBack and reencoded_backspace_font.
          Now the C has to handle this by itself.

Sat Mar  1 19:12:22 1997  Akim Demaille  <demaille@gargantua.enst.fr>

        * *.enc: they build their own dictionaries, to ease multi
          lingual documents.
```

The syntax is really simple: A line specifying the author and the date of the changes, then a list of changes, all of them starting with an star followed by the name of the files concerned, then optionally between parentheses the functions affected, and then some comments.

Implementation

Quite naturally the style will be called ChangeLog, hence:

```
style ChangeLog is
written by "Akim Demaille <demaille@inf.enst.fr>"
version is 1.0
requires a2ps 4.9.5

documentation is
```


```
   "This is a tutorial style sheet.\n"
end documentation
  ...
end style
```

A first interesting and easy entry is that of function names, between `(' and `)':

```
sequences are
  "(" Plain Label ")" Plain
end sequences
```

A small problem that may occur is that there can be several functions mentioned separated by commas, that we don't want to highlight this way. Commas, here, are exceptions. Since regular expressions are not yet implemented in `a2ps`, there is a simple but stupid way to avoid that white spaces are all considered as part of a function name, namely defining two exceptions: one which captures a single comma, and a second, capturing a comma and its trailing space.

For the file names, the problem is a bit more delicate, since they may end with `:', or when starts the list of functions. Then, we define two sequences, each one with one of the possible closers, the exceptions being attached to the first one:

```
sequences are
  "* " Plain Label_strong ":" Plain
     exceptions are
        ", " Plain, "," Plain
     end exceptions,
  "* " Plain Label_strong " " Plain
end sequences
```

Finally, let us say that some words have a higher importance in the core of text: those about removing or adding something.

```
keywords in Keyword_strong are
  add, added, remove, removed
end keywords
```

Since they may appear in lower or upper, of mixed case, the style will be defined as case insensitive.

Finally, we end up with this style sheet file, in which an optional highlighting of the mail address of the author is done. Saving the file is last step. But do not forget that a style sheet has both a name as nice as you may want (such as `Common Lisp'), and a key on which there are strict rules: the prefix must be alpha-numerical, lower case, with no more than 8 characters. Let's chose `` `chlog.ssh' ``.

```
# This is a tutorial on a2ps' style sheets
style ChangeLog is
written by "Akim Demaille <demaille@inf.enst.fr>"
version is 1.0
requires a2ps 4.9.5
```


```
documentation is
   "Second level of high lighting covers emails."
end documentation

sequences are
  "(" Plain Label ")" Plain
     exceptions are
        ", " Plain, "," Plain
     end exceptions,
  "* " Plain Label_strong ":" Plain
     exceptions are
        ", " Plain, "," Plain
     end exceptions,
  "* " Plain Label_strong " " Plain
end sequences

keywords in Keyword_strong are
  add, added, remove, removed
end keywords

optional sequences are
   < Plain Keyword > Plain
end sequences
end style
```

The Entry in `sheets.map'

The last touch is to include the pattern rules about `ChangeLog' files (which could appear as `ChangeLog.old' etc.) in `sheets.map':

```
# ChangeLog files
ChangeLog*                chlog
```

This won't work... Well, not always. Not for instance if you print `misc/ChangeLog'. This is not a bug, but truly a feature, since sometimes one gets more information about the type of a file from its path, than from the file name.

Here, to match the preceding path that may appear, just use `*':

```
# ChangeLog files
*ChangeLog*               chlog
```

If you want to be more specific (`FooChangeLog' should not match), use:

```
# ChangeLog files
ChangeLog*                chlog
```

```
*/ChangeLog*                chlog
```

More Sophisticated Rules

The example we have presented until now uses only basic features, and does not take advantage of the regexp. In this section we should how to write more evolved pretty printing rules.

The target will be the lines like:

```
Sun Apr 27 14:29:22 1997  Akim Demaille  <demaille@inf.enst.fr>

Fri Apr 25 14:05:20 1997  Akim Demaille  <demaille@inf.enst.fr>
```

There are three fields: the date, the name, the mail. These lines all start at the beginning of line. The last field is the easier to recognize: is starts with a `<', and finishes with a `>'. Its rule is then `/<[^>]+>/'. It is now easier to specify the second: it is composed only of words, at least one, separated by blanks, and is followed by the mail: `/[[:alpha:]]+\\([\t]+[[:alpha:]]+\\)*/'. To concatenate the two, we introduce optional blanks, and we put each one into a pair of `\\('-`\\)' to make each one a recognizable part:

```
\\([[:alpha:]]+\\([ \t]+[[:alpha:]]+\\)*\\)
\\(.+\\)
\\(<[^>]+>\\)
```

Now the first part is rather easy: it starts at the beginning of the line, finishes with a digit. Once again, it is separated from the following field by blanks. Split by groups (see section 'Grouping Operators' in Regex manual), we have:

```
^
\\([^\t ].*[0-9]\\)
\\([ \t]+\\)
\\([[:alpha:]]+\\([ \t]+[[:alpha:]]+\\)*\\)
\\(.+\\)
\\(<[^>]+>\\)
```

Now the destination is composed of back references to those groups, together with a face:

```
# We want to highlight the date and the maintainer name
optional operators are
  (/^\\([^\t ].*[0-9]\\)/                          # \1. The date
   /\\([ \t]+\\)/                                  # \2. Spaces
   /\\([[:alpha:]]+\\([ \t]+[[:alpha:]]+\\)*\\)/ # \3. Name
   /\\(.+\\)/                                      # \5. space and <
   /\\(<[^>]+\\)>/                                 # \6. email
   \1 Keyword, \2 Plain, \3 Keyword_strong,
   \5 Plain, \6 Keyword, > Plain)
end operators
```

Notice the way regexps are written, to ease reading.

PostScript

This chapter is devoted to the information which is only relevant to PostScript.

Page Device Options

Page device is a PostScript level 2 feature that offers an uniform interface to control printer's output device. `a2ps` protects all page device options inside an if block so they have no effect in level 1 interpreters. Although all level 2 interpreters support page device, they do not have to support all page device options. For example some printers can print in duplex mode and some can not. Refer to the documentation of your printer for supported options.

Here are some usable page device options which can be selected with the `-D' option (`--setpagedevice'). For a complete listing, see PostScript Language Reference Manual: section 4.11 Device Setup.

`Collate boolean`
: how output is organized when printing multiple copies

`Duplex boolean`
: duplex (two side) printing

`ManualFeed boolean`
: manual feed paper tray

`OutputFaceUp boolean`
: print output `face up' or `face down'

`Tumble boolean`
: how opposite sides are positioned in duplex printing

Statusdict Options

The `statusdict` is a special storage entity in PostScript (called a dictionary), in which some variables and operators determine the behavior of the printer. This is an historic horror that existed before page device definitions were defined. They are even more printer dependent, and are provided only for the people who don't have a level printer. In any case, refer to the documentation of your printer for supported options.

Here are some statusdict definitions in which you might be interested:

`manualfeed boolean`
: Variable which determine that the manual fed paper tray will be used. Use is `-Smanualfeed::true'.

`setmanualfeed boolean`
: Idem as the previous point, but use is `-Ssetmanualfeed:true'.

`setduplexmode boolean`

If boolean, then print in Duplex mode. Use if `-Ssetduplexmode:true'.

Colors in PostScript

Nevertheless, here are some tips on how to design your PostScript styles. It is strongly recommended to use `gray.pro' or `color.pro' as a template.

There are two PostScript instructions you might want to use in your new PostScript prologue:

`setgray`
: this instruction must be preceded by a number between 0 (black) and 1 (white). It defines the gray level used.

`setrgbcolor`
: this instruction must be preceded by three numbers between 0 (0 %) and 1 (100%). Those three numbers are related to red, green and blue proportions used to designate a color.

`a2ps` uses two upper level procedures, `BG` and `FG`, but both use an argument as in `setrgbcolor`. So if you wanted a gray shade, just give three times the same ratio.

a2ps PostScript Files

`a2ps` uses several types of PostScript files. Some are standards, such as font files, and others are meant for a2ps only.

All a2ps files have two parts, one being the comments, and the other being the content, separated by the following line:

```
% code follows this line
```

Designing PostScript Prologues

It is pretty known that satisfying the various human tastes is an NEXPTIME-hard problem, so `a2ps` offers ways to customize its output through the prologue files. But since the authors feel a little small against NEXPTIME, they agreed on the fact that **you** are the one who will design the look you like.

Hence in this section, you will find what you need to know to be able to customize `a2ps` output.

Basically, `a2ps` uses faces which are associated to their "meaning" in the text. `a2ps` let's you change the way the faces look.

Definition of the faces

There are three things that define a face:

Its font
: You should never call the font by yourself, because sometimes `a2ps` may decide that another font

would be better. This is what happens for instance if a font does not support the encoding you us

Hence, never set the font by yourself, but ask `a2ps` to do it. This is done through a line:

```
%Face: face real-font-name size
```

This line tells `a2ps` that the font of face is real-font-name. It will replace this line by the correct PostScript line to call the needed font, and will do everything needed to set up the font.

The size of the text body is `bfs`.

Its background color

There are two cases:

You want a background color, then give the RGB (see section Colors in PostScript) ratio and `true` to `BG`:

```
0.8 0.8 0 true BG
```

You don't want a background color, then call `BG` with `false`:

```
false BG
```

Its foreground color

As `BG`, call `FG` with an RGB ratio:

```
0 0.5 0 FG
```

Its underlining

`UL` requires a boolean argument, depending whether you want or not the current face to be underlined.

```
true UL
```

Its boxing

Requiring a boolean, `BX` let's a face have a box drawn around.

Prologue File Format

Prologue files for `a2ps` must have `pro' as suffix. Documentation (reported with `--list-prologues') can be included in the comment part:

```
Documentation
This prologue is the same as the prologue code(pb)code, but using
the bold version of the fonts.
EndDocumentation
% code follows this line
```

See section Documentation Format for more on the format.

A step by step example

We strongly suggest our readers not to start from scratch, but to copy one of the available styles (see the result of `a2ps --list=prologues'), to drop it in one of `a2ps' directories (say `$HOME/.a2ps', and to patch it until you like it.

Here, we will start from `color.pro', trying to give it a funky look.

Say you want the keywords to be in Helvetica, drawn in a flashy pink on a light green. And strong keywords, in Times Bold Italic in brown on a soft Hawaïan sea green (you are definitely a fine art *amateur*).

Then you need to look for `k' and `K':

```
/k {
  false BG
  0 0 0.9 FG
%Face: Keyword Courier bfs
  Show
} bind def

/K {
  false BG
  0 0 0.8 FG
%Face: Keyword_strong Courier-Bold bfs
  Show
} bind def
```

and turn it into:

```
/k {
  0.2 1 0.2 true BG
  1 0.2 1 FG
%Face: Keyword Helvetica bfs
  Show
} bind def

/K {
  0.4 0.2 0 true BG
  0.5 1 1 FG
%Face: Keyword_strong Times-BoldItalic bfs
  Show
} bind def
```

Waouh! It looks great!

A bit trickier: let change the way the line numbers are printed.

First, let's look for the font definition:


```
%%BeginSetup
% The font for line numbering
/f# /Helvetica findfont bfs .6 mul scalefont def
%%EndSetup
```

Let it be in Times, twice bigger than the body font.

```
%%BeginSetup
% The font for line numbering
/f# /Times-Roman findfont bfs 2 mul scalefont def
%%EndSetup
```

How about its foreground color?

```
% Function print line number (<string> # -)
/# {
  gsave
    sx cw mul 2 div neg 0 rmoveto
    f# setfont
    0.8 0.1 0.1 FG
    c-show
  grestore
} bind def
```

Let it be blue. Now you know the process: just put `0 0 1' as `FG` arguments.

Programming with the Library

`a2ps` offers to the programmer an access to its generating routines. This section documents the API.

But since this section is empty, or almost, if I were you, I would go in contrib/sample to see how it works...

Initialization and Closing of Calls to `liba2ps`

Function: struct a2ps_job * **a2ps_job_new** *(void)*

Build a new storage unit for the library.

Function: void **a2ps_read_sys_config** *(struct a2ps_job * job)*

Set job with the default settings defined in the configuration file of the system.

Function: void **a2ps_read_config** *(struct print_job * job, char * path, char * filename)*

Set job with the default settings defined in the configuration file `` `path/filename' ``. path can be `NULL`, filename cannot.

Print Jobs

A print job should be seen as associated to a single output.

Function: void **a2ps_open_output_session** *(struct a2ps_job * job)*

Initialize job for a new print job.

Function: void **a2ps_close_output_session** *(struct print_job * job)*

Closes the current print job, and sends the output.

File Jobs

A file job should be seen as a single input.

Function: void **a2ps_new_input_session** *(struct a2ps_job * job, char * name)*

Create and open a new file job in job. name can be `NULL`, in which case its name is defaulted to that of `stdin`.

Function: void **a2ps_close_input_session** *(struct print_job * job)*

End the current input session.

Printing Functions

These are the functions to be used to send content to `liba2ps`.

Function: void **a2ps_print_char** *(struct print_job * job, unsigned char c, face_t face)*

Print the char c in face in job.

Function: void **a2ps_print_string** *(struct print_job * job, unsigned char * string, face_t face)*

Print the `C` string string in face in job.

Function: void **a2ps_print_buffer** *(struct print_job * job, unsigned char * buffer, size_t len, face_t face)*

Print the len first characters contained in buffer with face into job.

Contributions

Card

Many users of `a2ps` have asked for a reference card, presenting a summary of the options. In fact, something closely related to the output of `a2ps --help'.

The first version of this reference card was a PreScript file (see section PreScript) to be printed by `a2ps`. Very soon a much better scheme was found: using a style sheet to pretty print directly the output of `a2ps --help'! A first advantage is then that the reference cards can be printed in the tongue you choose.

A second was that this treatment could be applied to any application supporting a `--help'-like option.

Invoking `card`

```
card [options] applications
```

`card` is a shell script which tries to guess how to get your applications' help message (typically by the options `--help' or `-h'), and pretty prints it thanks to `a2ps` (or the content of the environment variable `A2PS' if it is set).

Supported options are:

Option: **-h**

Option: **--help**

print a short help message and exit successfully.

Option: **-V**

Option: **--version**

report the version and exit successfully.

Option: **-D**

Option: **--debug**

enter in debug mode.

Option: **-l** *language*

Option: **--language=language**

specify the language in which the reference card should be printed. language should be the symbol used by `LC_ALL` etc. (such as `fr', `it' etc.).

If the applications don't support internationalization, English will be used.

Option: **--command=command**

Don't try to guess the applications' way to report their help message, but rather use the call command. A typical example is

```
card --command="cc -flags"
```

Any option that is not recognized by `card` is passed to `a2ps` (see section Command line options). Be aware that these options must *not* be separated from their arguments. For instance

```
card gmake gtar --command="cc -flags" -Pdisplay
```

builds the reference card of `GNU make`, `GNU tar` (automatic detection of `--help' support), and `cc` thanks to `-flags'. Since `-P' is not supported by `card`, it is passed to `a2ps`.

Caution when Using `card`

Remember that `card` runs the programs you give it, and the commands you supplied. Hence if there is a silly programs that has a weird behavior given the option `-h' etc., beware of the result.

It is even more clear using `--command': avoid running `card --command="rm -rf *"', because the result will be exactly what you think it will be!

psmandup

I personally hate to print documents of hundreds of pages on a single sided printer. Too bad, here there are no Duplex printers. The idea is then simply first to print the odd pages, then the even in reversed order. To make sure one flips the page in the meanwhile, the second half should be printed from the manual feed tray.

Make a shell script that automates this, and you get `psmandup`

Invoking `psmandup`

```
psmandup [file]
```

produce a manual duplex version of the PostScript file (or of the standard input if no file is given, or if file is `-'). Once the first half is printed, put the sheet stack in the manual feed tray for the second half.

Supported options are:

Option: **-h**

Option: **--help**

print a short help message and exit successfully.

Option: **-V**

Option: **--version**

report the version and exit successfully.

Option: **-D**

Option: **--debug**

enter in debug mode.

Option: **-o** *file*

Option: **--output=file**

specify the file in which is saved the output.

Option: **-n**

Option: **--no-fix**

`psmandup` will fail on ill designed PostScript. Actually it is the psutils that fail. To avoid this, by default the PostScript file is sanitized by `fixps`.

When given this option, don't run `fixps`. This is meant to be used when `fixps` has already been used higher in the processing chain.

`psmandup` makes the assumptions that the printer is Level 2, and support manual feeding. The file should be reasonably sane, otherwise `psmandup` fails miserably.

Typical use is

```
psmandup file.ps | lp
```

or can be logged into `a2ps`' printer commands (see section Your Printers).

`fixps`

The shell script `fixps` tries its best to fix common problems in PostScript files that may prevent post processing. It makes heavy use of the `psutils`. It is a good idea to use `fixps` upstream in the PostScript to PostScript delegations.

Invoking `fixps`

```
fixps [file]
```

sanitize the PostScript file (or of the standard input if no file is given, or if file is `-').

Supported options are:

Option: **-h**

Option: **--help**

print a short help message and exit successfully.

Option: **-V**

Option: **--version**

report the version and exit successfully.

Option: **-D**

Option: **--debug**

enter in debug mode.

Option: **-o** *file*

Option: **--output=file**

specify the file in which is saved the output.

a2ps Emacs mode

FIXME: Document me.

Various

We collected in this chapter the things that had no place in the rest of the document.

Security issues

Note. I am not really aware of security problems. I am just making assumptions upon my poor knowledge, so if somebody sees things that should be reported here, or problem I'm not aware of, please contact us so that this note gets extended or fixed.

One should understand that any program that has not been written to be secure is never secure. It is of course the case of `a2ps`.

Do not panic, there are no reason for you to worry. Nevertheless we can yet imagine ways to obtain illegal rights thanks to some features of `a2ps`, especially virtual printers.

If `a2ps` is run by root, then the files it may create are owned by root. This can for instance happen if you install `a2ps` as one of `lp` or `lpr` filter.

Then, if one of the virtual printer creates say a shell script, it is owned by root too. With just a bit of habit, it should not be difficult then to access privileged access to the system.

In what conditions could it happen? **Only** if there are some printers defined in the system configuration file of `a2ps`, or in root's home directory `` `.a2ps/a2psrc' ``. Hence, make sure to carefully write the commands to the preconfigured printers.

As you can see, this is quite science fiction, nevertheless, you might wanted to know.

Non PostScript printers

Here are some tips we have on how to use a non PostScript printer. If somebody feels like writing a more precise documentation, he really is welcome.

From Clayton Weaver

(for a bj200ex on os2 from linux) This is a shell script called as a filter from an ":if=/usr/local/bin/filtername" entry in /etc/printcap on linux:

```
#!/bin/bash

gs -q -sDevice=bj200 -sOutputfile=- -r360 -dNOPAUSE -dSAFER -
```

The device is a gs device name, input is the ending "-" (stdin), output is stdout, -q is "no messages". Can be used with

```
lpr -Pprintername -Ffilter_path_and_name infile.ps
```

or just `lpr -Pprintcap_printername infile.ps' if the filter is specified in the printcap `:if=' entry for that "printcap_printername". Either way, lpr on linux runs it in a sub shell, so you don't want "exec" before the gs command (that kills the sub shell and leaves gs running with no way to communicate its stdout back to lpr).

You can print directly from gs, I've done it on OS/2 by specifying "lpt1" as the output device, but printing to a remote printer I need lpr and lpd to handle transmitting the job, so this way works better for me.

Regards, Clayton Weaver (Seattle)

From Pierre Juillot

To print toto.ps on my desk-jet, I hit:

```
> gsdj toto.ps
```

where `gsdj' is

```
#!/bin/sh
#                               05-06-1996
# to print a PostScript file using GhostScript on the
# desk-jet 500 (dj) or the printer specified via  the
# environment variable LPOPTS
# using GhostScript
# exec gs -q -dNOPAUSE -sPAPERSIZE=a4 -sDEVICE=djet500 \
exec gs -q -dNOPAUSE -sPAPERSIZE=a4 -sDEVICE=deskjet \
```


```
     -sOutputFile="|lp -onb -ddj -or $LPOPTS " $* quit.ps
```

These tips in a2ps

To use these examples, first see section Your Printers. The translations would be in one of your configuration files, say `$HOME/.a2ps/a2psrc':

```
Printer: lpt1 | gs -q -sDevice=bj200 -sOutputfile=- -r360 -dNOPAUSE \
        -dSAFER > lpt1
UnknownPrinter:  | gs -q -dNOPAUSE -sPAPERSIZE=a4 -sDEVICE=deskjet \
     -sOutputFile="|lp -onb -ddj -or #o" quit.ps
```

Frequently asked questions

Please, before sending us mail, make sure the problem you have is not known, and explained. Moreover, avoid using the mailing list for asking question about the options, etc. It has been built for announces and suggestions, not to contact the authors.

Why Does ...?

Error related questions.

Relocation Error

```
When I have just compiled a2ps, and I first try it, I have weird
errors such as:

gargantua ~T $ a2ps --help
ld.so.1: a2ps: fatal: relocation error: symbol not found:
       ps_comment_hook: referenced in a2ps
zsh: 7294 killed     a2ps --help
```

This is due to a mix between several versions of the `liba2ps`. To fix the problem, make sure to run `make install' in the `lib/' directory and then in the `src/' directory. Removing the previous `liba2ps.so' files may help.

Printer Errors

There are two ways that printing can fail: silently, or with an diagnostic.

First check that you did not give exotic options to an old printer (typically, avoid printing on two sides on a printer that does not support it). Avoid use of options `-D' (see section Page Device Options) and `-S' (see section Statusdict Options).

If the trouble persists, please try again but with the option `--debug' (a PostScript error handler is down-loaded), and then send us:

1. the file that gives problems
2. the error message that was printed.

Cannot Print in Duplex

If your printer is too old, then `a2ps` will not be able to send it the code it needs when `-s2' is specified. This is because your printer uses an old and not standardized interface for special features.

So you need to

1. specify that you want Duplex mode: `-s2',
2. remove at hand the standardized call to the Duplex feature: `-DDuplex',
3. add the non standard call to the Duplex features. Try `-Ssetduplexmode:true'.

Since this is painful to hit, a User Option (see section Your Shortcuts) should help.

Printing goes beyond the frame of the paper

You are currently printing with a bad medium, for instance using A4 paper within `a2ps`, while your printer uses Letter paper. See section Sheets Options, option `--medium' for more.

What I get on the printer is long and incomprehensible

You are probably printing a PostScript file or equivalent. Try to print with `-Z': `a2ps` will try to do his best to find what is the program that can help you (see section Your Delegations). In case of doubt, don't hesitate to save into a file, and check the content with GhostView, or such:

```
$ a2ps my_weird_file -Z -o mwf.ps
$ ghostview mwf.ps
```

If it is not correct, ask for help around you.

Why Not ...?

Why not having used yacc and such

There are several reasons why we decided not to use grammars to parse the files. First it would have made the design of the style sheets much more tricky, and today `a2ps` would know only 4 or 5 languages.

Second, it limits the number of persons who could build a style sheet.

Third, mixing several parsers within one program is not easy. Moreover, its would have been ten times bigger.

Fourth, we did not feel the need for such a powerful tool: handling the keywords and the sequences is just what the users expect.

Fifth, any extension of `a2ps` would have required to recompile.

And last but not least, using a parser requires that the sources are syntactic bug free, what is too strong a requirement.

Nevertheless, `PreScript` gives the possibility to have on the one hand a syntactic parser which would produce `PreScript` code, and on the other hand, `a2ps`, which would make it PostScript. This schema seems to us a good compromise. If it is still not enough for you, you can use the library (see section Programming with the Library).

How Can I ...?

How can I leave room to bind?

The option `--margin[=size]' is meant for this. See section Sheets Options.

Printing stdin

`a2ps` prints the standard input if you give no file name, or if you gave `-' as file name. It cannot automatically pretty print, nor call delegations if you don't specify the language (see section Style Sheet Files), or if you don't supply a name to the standard input (`--stdin=name') with which it could guess the language.

Change the Font

Well, yes. See section Designing PostScript Prologues, for details. Make sure that all the information `a2ps` needs is available (see section Font Files).

The Old Option -b

By the past, `a2ps` had an option `-b' with which the fonts were bold. Since now the fonts are defined by prologues (see section Designing PostScript Prologues) this option no longer makes sense. A replacement prologue is provided: `bold'. To use it, give the option `--prologue=bold'.

What next?

There are some features we would like to implement. Some are the following ones. If you feel like helping, don't hesitate to contact us.

EPSF

`a2ps` should be able to generate EPS files in a short while.

Spelling checking, syntax fixing.

When printing a human file (only English may be supported), spelling should be checked. When pretty-printing, syntax errors should be underlined.

No just kidding :). Do it yourself...

Glossary

This section settles some terms used through out this document, and provides the definitions of some terms you probably want to know about.

Adobe

Adobe is the firm who designed and owns the PostScript language. The patent that printer manufacturers must pay to Adobe is the main reason why PostScript printers are so expansive.

AFM file

AFM stands for Adobe Font Metrics. These files contain everything one needs to know about a font: the width of the characters, the available characters etc.

Charset

Code Set

Cf. Encoding.

Delegate

Another filter (application) which `a2ps` may call to process some files. This feature is especially meant for page description files (see section Your Delegations).

DSC

Document Structuring Conventions

Because PostScript is a language, any file describing a document can have an arbitrary complexity. To ease the processing of PostScript files, the document should follow some conventions. Basically there are two kinds of conventions to follow:

Page Independence

Special comments state where the pages begin and end. With these comments (and the fact that a page does start and end somewhere, which is absolutely not necessary in PostScript), very simple programs (such as `psnup`, `psselect` etc.) can post process PostScript files.

Requirements

Special features may be needed to run correctly the file. Some comments specify what services are expected from the printer (e.g., fonts, duplex printing, color etc.), and other what features are provided by the file itself (e.g., fonts, procsets etc.), so that a print manager can decide that a file cannot be printed on that printer, or that it is possible if the file is slightly modified (e.g., adding a required font not known by the printer) etc.

The DSC are edited by Adobe. A document which respects them is said to be DSC compliant.

Of course `a2ps` follows all the DSC.

- Encoding Association of human readable characters, and computers' internal numbered representatio
In other words, they are the alphabets, which are different according to your country/mother tongue. E.g.
ASCII, Latin 1, corresponding to Western Europe etc.

See section What is an Encoding to know more about encodings.

- `GhostScript` `GhostScript` is a full PostScript interpreter running under many various systems (Unixes, MS-DOS, Mac etc.). It can be used either to view PostScript files (in general thanks to a graphic interface such as `GhostView` or `gv` ...), or to translate the PostScript in another format (for instance `PCL`, `PDF`, and many formats supported by some printers such as the Desk Jets.

Thanks to `GhostScript` many people not owning a PostScript printer are still able to use the PostScript technology.

- Face A virtual style given to some text. For instance, *Keyword*, *Comment* are faces.
- Headings Everything that goes around the page and is not part of the text body. Typically the title, footer etc.
- Key Many objects used in `a2ps`, such as encodings, have both a key and a name. The word name is used for a symbol, a label, which is only meant to be nice to read by a human. For instance `ISO Latin 1' is a name. `a2ps` never uses a name, but the key.

A key is the identifier of a unique object. This is information that `a2ps` processes, hence, whenever you need to specify an object to `a2ps`, use the key, not its name. For instance `latin1' is the unique identifier of the `ISO Latin 1' encoding.

- Logical page Cf. Virtual page.
- lhs
- left hand side See P-rule.
- Medium Official name (by Adobe) given to the output physical support. In other words, it means the description of a sheet, e.g., A4, Letter etc.
- Name See Key.
- Page A single side of a sheet.
- Page Description Language A language that describes some text (which may be enriched with pointers, pictures etc.) and its layout. `HTML`, PostScript, LaTeX, `roff` and others are such languages. A file written in those languages is not made to be read as is by a human, but to be transformed (or compiled) into a readable form.
- PCL FIXME:
- PFA file PostScript Font in ASCII format. This file can be directly down loaded to provide support for another font.
- PFB file PostScript Font in Binary format. In PFA files there are long sequences of hexadecimal digits. Here these digits are represented by their value, hence compressing 2 characters in a PFA into 1 in the PFB. This is the only advantage since a PFB file cannot be directly sent to printer: it must first be decompressed (hence turned into a PFA file) before being used.
- PostScript PostScript is a page description language. It is even more powerful than that: unlike to `HTML`, or `roff`, but as TeX and LaTeX, it is truly a programming language which main purpose is to draw (on sheets). Most programs are a list of instructions that describes lines, shades of gray, or text to draw on a page.

This is the language that most printers understand.

PostScript is a trademark of Adobe Systems Incorporated.

- PPD file PostScript Printer Description file. These files report everything one needs to know about a printer: the known fonts, the patches that should be down loaded, the available memory, the trays, the way to ask it duplex printing etc.

PostScript has pretended to be a device independent page description language, and the `PPD` files are here to prove that device independence was a failure.

- ProcSet Set of (PostScript) procedures.
- Prologue PostScript being a language, a typical PostScript program (i.e. a typical PostScript file) consists of two parts. The first part is composed of resources, such as fonts, procsets, etc. and the second part of calls to these procedures. The first part is called the prologue, and the second, the script.
- P-rule FIXME:
- rhs
- right hand side See P-rule.
- Script See Prologue.
- Sheet The physical support of the printing: it may support one or two pages, depending on your printing options.
- Style sheet Set of rules used by `a2ps` to give a face to the strings of a file. In `a2ps`, each programming language which is supported is defined via one style-sheet.
- Virtual page Area on a physical page in which `a2ps` draws the content of a file. There may be several virtual pages on a physical page.

Genesis

Here are some words on `a2ps` and its history.

History

The initial version was a shell program written by Evan Kirshenbaum. It was very slow and contained many bugs.

A new version was written in `C` by Miguel Santana to improve execution speed and portability. Many new features and improvements have been added since this first version. Many contributions (changes, fixes, ideas) were done by `a2ps` users in order to improve it.

From the latest version from Miguel Santana (4.3), Emmanuel Briot implemented bold faces for keywords in `Ada`, `C` and `C++`.

From that version, Akim Demaille generalized the pretty-printing capabilities, implemented more languages support, and other features.

Contributors

Since the very beginning of `a2ps`, many people have brought help:

- Oscar Nierstrasz
- Tim Clark
- tullemans@apolloway.prl.philips.nl
- Johan Vromans
- craig.r.stevenson@att.com
- erikt@cs.umu.se
- wstahw@lso.win.tue.nl
- mai@wolfen.cc.uow.oz
- Johan Garpendahl
- John Interrante
- Chris Adamo
- Larry Barbieri
- Denis Girou
- Thomas Parmelan
- Paul Shum
- Graham Jenkins, thanks to whom `a2ps` (but also GNU `Libtool`) is ported onto Pyramid Systems.
- Didier Verna wrote `a2ps-compile-regexp` for the `a2ps-mode`.
- Jim Diamond.
- Christian Mondrup who regularly tracks down bugs appearing on DEC architectures.
- Michael Taeschner who tracked the same bug, but with another look, on Irix 6.2.
- Peter Bartke
- Joachim Backes who caught many bugs, and problems in the documentation.

Translators

Some people worked on the translation of `a2ps`:

- Daniele Ghiotti (Italian)
- Tomek Burdziak (Polish)
- Miguel A. Varo (Maintainer for Spanish and Catalan)
- Michael Wiedmann (Maintainer for German)
- Christian Kirsch (German)
- Erwin Dieterich (German)

- Juliusz Chroboczek (Polish). He is also the author of `Ogonkify` (see section 'Overview' in Ogonkify manual).
- Marcel van der Laan (Dutch)
- Lorenzo M. Catucci (Maintainer for Italian)
- Choi Jun Ho (Korean)
- Turgut Uyar (Turkish)
- Jiri Pavlovsky (Maintainer for Czech)

Thanks

Patrick Andries, from Alis Technologies inc. and Roman Czyborra (see his home page), provided us with important information on encodings. We strongly recommend that you go and read these pages: there is a lot to learn.

Juliusz Chroboczek worked a lot on the integration of the products of Ogonkify (such as Latin 2 etc. fonts) in `a2ps`. Without his help, and the time is devoted to both `a2ps` and `ogonkify`, many non west-European people would still be unable to print easily texts written in their mother tongue.

Denis Girou brought a constant and valuable support through out the genesis of pretty-printing `a2ps`. His comments on both the program and the documentation are the origin of many pleasant features (such as `--prologue').

Copying

The subroutines and source code in the `a2ps` package are "free"; this means that everyone is free to use them and free to redistribute them on a free basis. The `a2ps`-related programs are not in the public domain; they are copyrighted and there are restrictions on their distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of these programs that they might get from you.

Specifically, we want to make sure that you have the right to give away copies of the programs that relate to `a2ps`, that you receive source code or else can get it if you want it, that you can change these programs or use pieces of them in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of the `a2ps`-related code, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for the programs that relate to `a2ps`. If these programs are modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

The precise conditions of the licenses for the programs currently being distributed that relate to `a2ps` a found in the General Public Licenses that accompany them.

Concept Index

%

.

:

a

b

c

d

e

f

g

h

k

l

m

n

o

p

r

Function Index

TOC

a

a2ps, version 4.10

(2)

A classical Unix trick to make the difference between the option `-2', and the file `-2' is to type `./-2'.

(3)

Basically it means the the PostScript that `a2ps` generates is **really** modular. In other words, any kind of post-processing may be applied to the files it produced.

(4)

That is to say, there are no PostScript printers that don't understand these files.

(5)

Current `a2ps` only handles PostScript output, i.e. out=`ps'

(6)

Because hiding its use into `a2ps` just makes it even more difficult to the users to know why it failed. Let them use it at hand.

AUC TeX

A much enhanced LaTeX mode for GNU Emacs.

June 1993, for AUC TeX Version 7.3

by Kresten Krab Thorup updated for 7.3 by Per Abrahamsen

AUC TeX

TOC

Copying

(This text stolen from the TeXinfo 2.16 distribution).

The programs currently being distributed that relate to AUC TeX include lisp files for GNU Emacs, plus other separate programs (specifically `lacheck`). These programs are free; this means that everyone is free to use them and free to redistribute them on a free basis. The AUC TeX related programs are not in the public domain; they are copyrighted and there are restrictions on their distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of these programs that they might get from you.

Specifically, we want to make sure that you have the right to give away copies of the programs that relate to AUC TeX, that you receive source code or else can get it if you want it, that you can change these programs or use pieces of them in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of the AUC TeX related programs, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for the programs that relate to AUC TeX. If these programs are modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

The precise conditions of the licenses for the programs currently being distributed that relate to AUC TeX are found in the General Public Licenses that accompany them.

Introduction

AUC TeX is a comprehensive customizable integrated environment for writing input files for LaTeX using GNU Emacs. AUC TeX lets you run TeX/LaTeX and other LaTeX-related tools, such as output filters or post processors from inside Emacs. Especially `running LaTeX' is interesting, as AUC TeX lets you browse through the errors TeX reported, while it moves the cursor directly to the reported error, and displays some documentation for that particular error. This will even work when the document is spread over several files. AUC TeX automatically indents your `LaTeX-source', not only as you write it--you can also let it indent and format an entire document. It has a special outline feature, which can greatly help you `getting the overview' of a document. Apart from these special features,

TOC

AUC TeX provides a large range of handy Emacs macros, which in several different ways can help you write yo LaTeX documents fast and without pain. Here's a short list of the major features:

- Insertion of templates for logical-structural compositions such as environments and sections.
- Hot-keys for easy access to certain often used constructs, e.g., font changes, accented letters, and mathematical symbols.
- Running application programs (such as TeX), and then parsing the output so that errors in the document may be located easily.
- Support for multi-file documents.
- Online help for (La)TeX error messages.
- Outlining--i.e., manipulating the document as a composition of nested/sequential logical constructs.
- Instant formatting and indentation of the ASCII-document in order to make it easier to read.
- `Completion' (and thereby spell-checking) of partially written control sequences.

Inserting Frequently Used Commands

The most commonly used commands/macros of AUC TeX are those which simply insert templates for often used TeX and/or LaTeX constructs, like font changes, handling of environments, etc. These features are very simple, and easy to learn, and help you avoiding stupid mistakes like mismatched braces, or `\begin{}'-`\end{}' pairs.

Insertion of Quotes Dollars, and Braces

In TeX literal double quotes `"like this"' are seldom used, instead two single quotes are used `"like this"'. To help you insert these efficiently, AUC TeX allows you to continue to press " to insert two single quotes. To get a literal double quote, press " twice.

Command: **TeX-insert-quote** *count*

(") Insert the appropriate quote marks for TeX.

Inserts the value of `tex-open-quote` (normally `"') or `tex-close-quote` (normally `"') depending on the context. With prefix argument, always inserts `"' characters.

User Option: **TeX-open-quote**

String inserted by typing " to open a quotation.

User Option: **TeX-close-quote**

String inserted by typing " to open a quotation.

If you include the style file `german' `TeX-open-quote` and `TeX-close-quote` will both be set to `"'.

In AUC TeX, dollar signs should match like they do in TeX. This has been parially implemented, we assume dollar signs always match within a paragraph. The first `$' you insert in a paragraph will do nothing special. The second `$' will match the first. This will be indicated by moving the cursor temporarily over the first dollar sign. If you enter a dollar sign that matches a double dollar sign `$$' AUC TeX will automatically insert two dollar signs. If you enter a second dollar sign that matches a single dollar sign, the single sollar sign will automatically be converted to a double dollar sign.

Command: **TeX-insert-dollar** *arg*

($) Insert dollar sign.

Show matching dollar sign if this dollar sign end the TeX math mode. Ensure double dollar signs match up correctly by inserting extra dollar signs when needed.

With optional arg, insert that many dollar signs.

To avoid unbalanced braces, it is useful to insert them pairwise. You can do this by typing C-c {.

Command: **TeX-insert-braces**

(C-c {) Make a pair of braces and position the cursor to type inside of them.

Inserting Font Specifiers

Perhaps the most used keyboard commands of AUC TeX are the short-cuts available for easy insertion of font changing macros. They all put the font change inside a TeX group, a practice that help preventing subtle errors. The most significant advantage of using these command instead of typing it in yourself, is that the braces will always match correctly.

If you give an argument (that is, type C-u) to the font command, the innermost font will be replaced, i.e. the font in the TeX group around point will be changed. The following table shows the available commands, with `-!-` indicating the position where the text will be inserted.

C-c C-f C-r
: Insert roman {\rm -!-} text.

C-c C-f C-b
: Insert **bold face** `{\bf -!-}' text.

C-c C-f C-i
: Insert *italics* `{\it -!-\/}' text.

C-c C-f C-e
: Insert *emphasized* `{\em -!-\/}' text.

C-c C-f C-s
: Insert *slanted* `{\sl -!-\/}' text.

C-c C-f C-t
: Insert `typewriter` `{\tt -!-}' text.

C-c C-f C-c
: Insert SMALL CAPS `{\sc -!-}' text.

C-c C-f C-d
: Delete the innermost font specification containing point.

Command: **TeX-font** *arg*

(C-c C-f) Insert template for font change command.

If replace is not nil, replace current font. what determines the font to use, as specified by `TeX-font-list`.

User Option: **TeX-font-list**

List of fonts used by TeX-font.

Each entry is a list with three elements. The first element is the key to activate the font. The second element is the

string to insert before point, and the third element is the string to insert after point. An optional fourth element m always replace if not nil.

Inserting chapters, sections, etc.

Insertion of sectioning macros, that is `\chapter', `\section', `\subsection', etc. and accompanying `\label"s may be eased by using C-c C-s. This command is highly customizable, the following describes the default behavior.

When invoking you will be asked for a section macro to insert. An appropriate default is automatically selected by AUC TeX, that is either: at the top of the document; the top level sectioning for that document style, and any other place: The same as the last occurring sectioning command.

Next, you will be asked for the actual name of that section, and last you will be asked for a label to be associated with that section. The label will be prefixed by the value specified in `LaTeX-section-hooks`.

Command: **LaTeX-section** *arg*

(C-c C-s) Insert a sectioning command.

Determine the type of section to be inserted, by the argument arg.

- If arg is nil or missing, use the current level.
- If arg is a list (selected by C-u), go downward one level.
- If arg is negative, go up that many levels.
- If arg is positive or zero, use absolute level:
  - 0 : part
  - 1 : chapter
  - 2 : section
  - 3 : subsection
  - 4 : subsubsection
  - 5 : paragraph
  - 6 : subparagraph

The following variables can be set to customize the function.

`LaTeX-section-hooks`

Hooks to be run when inserting a section.

`LaTeX-section-label`

Prefix to all section references.

The precise behavior of `LaTeX-section` is defined by the contents of `LaTeX-section-hooks`.

User Option: **LaTeX-section-hooks**

List of hooks to run when a new section is inserted.

The following variables are set before the hooks are run

level

Numeric section level, default set by prefix arg to `LaTeX-section`.

name

Name of the sectioning command, derived from level.

title
: The title of the section, default to an empty string.

toc
: Entry for the table of contents list, default nil.

done-mark
: Position of point afterwards, default nil meaning after the inserted text.

A number of hooks are already defined. Most likely, you will be able to get the desired functionality by choosing from these hooks.

`LaTeX-section-level-hook`
: Query the user about the name of the sectioning command. Modifies level and name.

`LaTeX-section-title-hook`
: Query the user about the title of the section. Modifies title.

`LaTeX-section-toc-hook`
: Query the user for the toc entry. Modifies toc.

`LaTeX-section-section-hook`
: Insert LaTeX section command according to name, title, and toc. If toc is nil, no toc entry is inserted. If toc or title are empty strings, done-mark will be placed at the point they should be inserted.

`LaTeX-section-label-hook`
: Insert a label after the section command. Controlled by the variable `LaTeX-section-label`.

To get a full featured `LaTeX-section` command, insert

```
(setq LaTeX-section-hooks
      '(LaTeX-section-level-hook
        LaTeX-section-title-hook
        LaTeX-section-toc-hook
        LaTeX-section-section-hook
        LaTeX-section-label-hook))
```

in your `` `.emacs' `` file.

The behavior of `LaTeX-section-label-hook` is determined by the variable `LaTeX-section-label`.

User Option: **LaTeX-section-label**

Default prefix when asking for a label.

If it is a string, it is used unchanged for all kinds of sections. If it is nil, no label is inserted. If it is a list, the list is searched for a member whose car is equal to the name of the sectioning command being inserted. The cdr is then used as the prefix. If the name is not found, or if the cdr is nil, no label is inserted.

By default, chapters have a prefix of `cha:' while sections and subsections have a prefix of `sec:'. Labels are not automatically inserted for other types of sections.

Inserting Environment Templates

A large apparatus is available that supports insertions of environments, that is `\begin{}' -- `\end{}' pairs.

AUC TeX is aware of most of the actual environments available in a specific document. This is achieved by

examining your `\documentstyle' command, and consulting a precompiled list of environments available in a large number of styles.

You insert an environment with C-c C-e, and select an environment type. Depending on the environment, AUC TeX may ask more questions about the optional parts of the selected environment type.

Command: **LaTeX-environment**

(C-c C-e) AUC TeX will prompt you for an environment to insert. At this prompt, you may press TAB or SPC to complete a partially written name, and/or to get a list of available environments. After selection of a specific environment AUC TeX may prompt you for further specifications.

As a default selection, AUC TeX will suggest the environment last inserted or, as the first choice the value of the variable `LaTeX-default-environment`.

User Option: **LaTeX-default-environment**

Default environment to insert when invoking `LaTeX-environment'.

If the document is empty, or the cursor is placed at the top of the document, AUC TeX will default to insert a `document' environment.

Most of these are described further in the following sections, and you may easily specify more, as described in `Customizing environments'.

You can close the current environment with C-c], but we suggest that you use C-c C-e to insert complete environments instead.

Command: **LaTeX-close-environment**

(C-c]) Insert an `\end' that matches the current environment.

Floats

Figures and tables (i.e., floats) may also be inserted using AUC TeX. After choosing either `figure' or `table' in the environment list described above, you will be prompted for a number of additional things.

float-to
: This field is the option of float environments that controls how they are placed in the final document. In LaTeX this is a sequence of the letters `htbp' as described in the LaTeX manual. The value will default to the value of `LaTeX-float`.

caption
: This is the caption of the float.

label
: The label of this float. The label will have a default prefix, which is controlled by the variables `LaTeX-figure-label` and `LaTeX-table-label`.

Moreover, in the case of a `figure' environment, you will be asked if you want to insert a `center' environment inside the figure.

User Option: **LaTeX-float**

Default placement for floats.

User Option: **LaTeX-figure-label**

Prefix to use for figure labels.

User Option: **LaTeX-table-label**

Prefix to use for table labels.

Itemize-like

In an itemize-like environment, nodes (i.e., `\item's) may be inserted using C-c LFD.

Command: **LaTeX-insert-item**

(C-c LFD) Close the current item, move to the next line and insert an appropriate `\item' for the current environment. That is, `itemize' and `enumerate' will have `\item ' inserted, while `description' will have `\item[]' inserted.

Tabular-like

When inserting Tabular-like environments, that is, `tabular' `array' etc., you will be prompted for a template for that environment.

Customizing environments

See section Adding Support for Environments, for how to customize the list of known environments.

Advanced Editing Features

The previous chapter described how to write the main body of the text easily and with a minimum of errors. In this chapter we will describe some features for entering more specialized sorts of text, and for indenting and navigating through the document.

Entering Mathematics

TeX is written by a mathematician, and has always contained good support for formatting mathematical text. AUC TeX supports this tradition, by offering a special minor mode for entering text with many mathematic symbols. You can enter this mode by typing C-c ~.

Command: **LaTeX-math-mode**

(C-c ~) Toggle LaTeX-math-mode. This is a minor mode rebinding the key `LaTeX-math-abbrev-prefix` to allow easy typing of mathematical symbols. ` will read a character from the keyboard, and insert the symbol as specified in `LaTeX-mathlist`. If given a prefix argument, the symbol will be surrounded by dollar signs.

You can use another prefix key (instead of `) by setting the variable LaTeX-math-abbrev-prefix.

User Option: **LaTeX-math-abbrev-prefix**

A string containing the prefix of `LaTeX-math-mode` commands; This value defaults to `.

The variable `LaTeX-mathlist` holds the actual mapping.

User Option: **LaTeX-mathlist**

A list containing key-command mappings to use in `LaTeX-math-mode`. The car of each element is the key and the

cdr is the macro name.

Completion

Emacs lisp programmers probably know the `lisp-complete-symbol` command, usually bound to M-TAB. Users of the wonderful ispell mode know and love the `ispell-complete-word` command from that package. Similarly, AUC TeX has a `TeX-complete-symbol` command, usually bound to C-c TAB. Using `LaTeX-complete-symbol` makes it easier to type and remember the names of long LaTeX macros. In order to use `TeX-complete-symbol`, you should write a backslash and the start of the macro. Typing C-c TAB will now complete as much of the macro, as it unambiguously can. For example, if you type ``\renewc" and then `C-c TAB, it will expand to ``\renewcommand". Command: **TeX-complete-symbol**

(C-c TAB) Complete TeX symbol before point.

A more direct way to insert a macro is with `TeX-insert-macro`, bound to C-c C-m. It has the advantage over completion that it knows about the argument of most standard LaTeX macros, and will prompt for them. It also knows about the type of the arguments, so it will for example give completion for the argument to `\include'. Some examples are listed below.

Command: **TeX-insert-macro**

(C-c C-m) Prompt (with completion) for the name of a TeX macro, and if AUC TeX knows the macro, prompt for each argument.

Completions work because AUC TeX can analyze TeX files, and store symbols in emacs lisp files for later retrieval. See section Automatic Customization, for more information.

Commenting

It is often necessary to comment out temporarily a region of TeX or LaTeX code. This can be done with the commands C-c ; and C-c %. C-c ; will comment out all lines in the current region, while C-c % will comment out the current paragraph. To uncomment, simply type C-u C-c ; to uncomment all lines in the region, or C-u C-c % uncomment all comment lines around point.

By default, these commands will insert or remove a single `%'. To insert more than one, give an argument. C-u 5 C-c % will add five `%' to each line, while C-u - 5 C-c % will remove up to 5 `%' from each line.

Command: **TeX-comment-region** *count*

(C-c ;) Add or remove `%' from the beginning of each line in the current region, as specified by count.

If count is nil (no prefix argument), 1 `%' will be added to each line.

If count is a list (a non-numeric prefix argument), 1 `%' will be removed from each line.

If count is positive, count `%"s will be added to each line.

If count is negative, count `%"s will be removed from each line.

Command: **TeX-comment-paragraph** *count*

(C-c %) Add or remove `%' from the beginning of each line in the current paragraph, as specified by count. When removing `%"s the paragraph is considered to consist of all preceding and succeeding lines starting with a `%', until the first non-comment line.

If count is nil (no prefix argument), 1 `%' will be added to each line.

If count is a list (a non-numeric prefix argument), 1 `%' will be removed from each line.

If count is positive, count `%"s will be added to each line.

If count is negative, count `%"s will be removed from each line.

Marking, Formatting and Indenting

AUC TeX contains very advanced handling of indentation and reformatting of the LaTeX source. If you have already tried AUC TeX with `auto-fill-mode` enabled, you may have noted that the source is automatically indented and formatted as you write it. More over, AUC TeX is able to format sections of text on demand.

It is important to realize, that AUC TeX comes with `formatting' in two fashions. Either letting TeX format the file, or letting AUC TeX make the ASCII document look better.

Indentation is done by LaTeX environments and by TeX groups, that is the body of an environment is indented by the value of `LaTeX-indent-level` (default 2). Also, items of an `itemize-like' environment are indented by the value of `LaTeX-item-indent`, default -2. This indentation makes it easier to see the structure of the document, and to catch errors such as a missing close brace. Thus, the indentation is done for precisely the same reasons that you would indent ordinary computer programs.

The following is a short sample of an itemize environment indented by AUC TeX. If more environment are nested, they are indented `accumulated' just like most programming languages usually are seen indented in nested constructs.

```
\begin{itemize}
\item Insertion of templates for logical-structural compositions such as
  environments and sections.
\item Hot-keys for easy access to certain often used constructs, e.g.,
  font changes, accented letters, and mathematical symbols.
\item Running application programs (such as \TeX), and then parsing
  the output so that errors in the document may be located
  easily.
\item Support for multi-file documents.
\item Online help for \AllTeX\ error messages.
\item Outlining\Dash i.e., manipulating the document as a composition
  of nested/sequential logical constructs.
\item Instant formatting and indentation of the \ascii-document in
  order to make it easier to read.
\item `Completion' (and thereby spell-checking) of partially written
  control sequences.
\end{itemize}
```

You can format and indent single lines, paragraphs, environments, or sections.

TAB

`LaTeX-indent-line` will indent the current line.

LFD

`reindent-then-newline-and-indent` indents the current line, and then inserts a new line (much like RET) and move the cursor to an appropriate position by the left margin.

M-q

Alias for C-c C-q C-p

C-c C-q C-p

`LaTeX-format-paragraph` will reformat or `fill' the current paragraph.

C-c C-q C-e

`LaTeX-format-environment` will reformat or `fill' the current environment. This may e.g. be the `document' environment, in which case the entire document will be formatted.

C-c C-q C-s

`LaTeX-format-section` will reformat or `fill' the current logical sectional unit.

M-g

Alias for C-c C-q C-r

C-c C-q C-r

`LaTeX-format-region` will format or `fill' the current region.

Warning: The formatting cannot handle tabular-like environments. Those will be completely messed-up if you try to format them.

User Option: **LaTeX-indent-level**

Number of spaces to add to the indentation for each `\begin' not matched by a `\end'.

User Option: **LaTeX-item-indent**

Number of spaces to add to the indentation for `\item"s in list environments.

User Option: **TeX-brace-indent-level**

Number of spaces to add to the indentation for each `{' not matched by a `}'.

Outlining the Document

Minor mode for editing outlines with selective display. Headings are lines which start with asterisks: one for major headings, two for sub-headings, etc. Lines not starting with asterisks are body lines.

Body text or sub-headings under a heading can be made temporarily invisible, or visible again. Invisible lines are attached to the end of the heading, so they move with it, if the line is killed and yanked back. A heading with text hidden under it is marked with an ellipsis (...).

C-c C-o C-n

`outline-next-visible-heading` move by visible headings

C-c C-o C-p

`outline-previous-visible-heading`

C-c C-o C-f

`outline-forward-same-level` similar but skip sub-headings

C-c C-o C-b

`outline-backward-same-level`

C-c C-o C-u

`outline-up-heading move from` sub heading to heading

C-c C-o C-t

`hide-body` make all text invisible (not headings).

C-c C-o C-a

`show-all` make everything in buffer visible.

C-c C-o C-o

`outline-minor-mode` leave outline mode.

The remaining commands are used when point is on a heading line. They apply to some of the body or sub-headings of that heading.

C-c C-o C-h

`hide-subtree` make body and sub-headings invisible.

C-c C-o C-s

`show-subtree` make body and sub-headings visible.

C-c C-o C-i

`show-children` make direct sub-headings visible. No effect on body, or sub-headings 2 or more levels down. With arg N, affects sub-headings N levels down.

C-c C-o C-c

`hide-entry` make immediately following body invisible.

C-c C-o C-e

`show-entry` make it visible.

C-c C-o C-l

`hide-leaves` make body under heading and under its sub-headings invisible. The sub-headings remain visible.

C-c C-o C-x

`show-branches` make all sub-headings at all levels visible.

User Option: **outline-prefix-char**

The prefix char (C-c C-o) used to activate the outline commands. You most likely want to set this to something shorter.

In LaTeX mode the default headings are the sectioning commands, and as top level headings `\documentstyle', `\appendix', `TeX-trailer-start', and `TeX-header-end'. You can also define extra headings at any level, by setting the variable `TeX-outline-extra'.

User Option: **TeX-outline-extra**

List of extra TeX outline levels.

Each element is a list with two entries. The first entry is the regular expression matching a header, and the second is the level of the header. See `LaTeX-section-list` for existing header levels.

You can use outlining in other modes as well, the function to do this is `outline-minor-mode`.

Command: **outline-minor-mode** *prefix*

Enters outline minor mode. With prefix: non-nil: turn outline mode on. nil: turn outline mode off.

User Option: **outline-regexp**

Regular expression matching header lines. When using outline-minor mode outside AUC TeX, this is per default set to lines beginning with one or more stars.

User Option: **outline-level-function**

Function calculating the level of a header. Outside AUC TeX, this is per default the length of the string matched by `outline-regexp`.

Formatting and Printing

The most powerful features of AUC TeX may be those allowing you to run (La)TeX and other external commands like BibTeX and `makeindex` from within Emacs, viewing and printing the results, and moreover allowing you to *debug* your documents.

Executing Commands

Formatting the document with TeX or LaTeX, viewing with a a previewer, printing the document, running BibTeX, making an index, or checking the document with `lacheck` all require running an external command.

There are two ways to run an external command, you can either run it on all of the current document with `TeX-command-master`, or on the current region with `TeX-command-region`.

Command: **TeX-command-master**

(C-c C-c) Query the user for a command, and run it on the master file associated with the current buffer. The name of the master file is controlled by the variable `TeX-master`. The available commands are controlled by the variable `TeX-command-list`.

See section Installation of AUC TeX for a discussion about `TeX-command-list` and section Multifile Documents for a discussion about `TeX-master`.

Command: **TeX-command-region**

(C-c C-r) Query the user for a command, and run it on the "region file". Some commands (typically those invoking TeX or LaTeX) will write the current region into the region file, after extracting the header and tailer from the master file. The name of the region file is controlled by the variable `TeX-region`. The name of the master file is controlled by the variable `TeX-master`. The header is all text up to the line matching the regular expression `TeX-header-end`. The trailer is all text from the line matching the regular expression `TeX-trailer-start`. The available commands are controlled by the variable `TeX-command-list`.

AUC TeX will allow one process for each document, plus one process for the region file to be active at the same time. Thus, if you are editing n different documents, you can have n plus one processes running at the same time. If the last process you started was on the region, the commands described in section Catching the errors and section Controlling the output will work on that process, otherwise they will work on the process associated with the current document.

User Option: **TeX-region**

The name of the file for temporarily storing the text when formatting the current region.

User Option: **TeX-header-end**

A regular expression matching the end of the header. By default, this is `\begin{document}' in LaTeX mode and `%**end of header' in TeX mode.

User Option: **TeX-trailer-start**

A regular expression matching the start of the trailer. By default, this is `\end{document}' in LaTeX mode and `\bye'

in TeX mode.

AUC TeX will try to guess what command you want to invoke, but by default it will assume that you want to run TeX in TeX mode and LaTeX in LaTeX mode. You can overwrite this by setting the variable `TeX-command-default`.

User Option: **TeX-command-default**

The default command to run in this buffer. Must be an entry in `TeX-command-list`.

If you want to overwrite the values of `TeX-header-end`, `TeX-trailer-start`, or `TeX-command-default`, you can do that for all files by setting them in either `TeX-mode-hook`, `plain-TeX-mode-hook`, or `LaTeX-mode-hook`. To overwrite them for a single file, define them as file variables (see section 'File Variables' in The Emacs Editor). You do this by putting special formatted text near the end of the file.

```
% Local Variables:
% TeX-header-end: "% End-Of-Header"
% TeX-trailer-start: "% Start-Of-Trailer"
% TeX-command-default: "SliTeX"
% End:
```

Catching the errors

Once you've formatted your document you may `debug' it, i.e. browse through the errors (La)TeX reported.

Command: **TeX-next-error**

(C-c `) Go to the next error reported by TeX. The view will be split in two, with the cursor placed as close as possible to the error in the top view. In the bottom view, the error message will be displayed along with some explanatory text.

Normally AUC TeX will only report real errors, but you may as well ask it to report `bad boxes' as well.

Command: **TeX-toggle-debug-bad-boxes**

(C-c C-w) Toggle whether AUC TeX should stop at bad boxes (i.e. over/under full boxes) as well as at normal errors.

As default, AUC TeX will display that special `*help*' buffer containing the error reported by TeX along with the documentation. There is however an `expert' option, which allows you to display the real TeX output.

User Option: **TeX-display-help**

When non-nil AUC TeX will automatically display a help text whenever an error is encountered using `TeX-next-error` (C-c `).

Checking for problems

Running TeX or LaTeX will only find regular errors in the document, not examples of bad style. Furthermore, description of the errors may often be confusing. The utility `lacheck` can be used to find style errors, such as forgetting to escape the space after an abbreviation or using `...' instead of `\ldots' and many other problems like that. You start `lacheck` with C-c C-c C h e c k RET. The result will be a list of errors in the `*compilation*' buffer. You can go through the errors with C-x ` (`next-error`, see section 'Compilation' in The Emacs Editor), which will move point to the location of the next error.

Controlling the output

A number of commands are available for controlling the output of an application running under AUC TeX

Command: **TeX-kill-job**

(C-c C-k) Kill currently running external application. This may be either of TeX, LaTeX, previewer BibTeX etc.

Command: **TeX-recenter-output-buffer**

(C-c C-l) Recenter the output buffer so that the bottom line is visible.

Command: **TeX-home-buffer**

(C-c ^) Go to the `master' file in the document associated with the current buffer, or if already there, to the file where the current process was started.

Multifile Documents

If you spread a document over many files (as you are likely to do if there are multiple authors, or if you have not yet discovered the power of the outline commands (see section Outlining the Document)), that can be done by having a "master" file from where you include the various files with the TeX macro `\input' or the LaTeX macro `\include'. These files may also include other files themselves. However, to format the document you must run the commands on the top level file.

When you for example ask AUC TeX to run a command on the master file, it has no way to know what the name of the master file is. If it finds the line indicating the end of the header in a master file (`TeX-header-end`), it can figure out for itself that this is a master file. Otherwise, it will ask for the name of the master file associated with the buffer. To avoid asking you again, AUC TeX will automatically insert the name of the master file as a file variable (see section 'File Variables' in The Emacs Editor). You can also insert the file variable yourself, by putting the following text at the end of your files.

```
% Local Variables:
% TeX-master: "mymaster"
% End:
```

You should always set this variable to the top level document. If you always use single file documents, or always use the same name for your top level documents, you can set this variable in your `.emacs' file.

```
(setq-default TeX-master t) ; Only use single file douments.

(setq-default TeX-master "master") ; All master files called "master".
```

User Option: **TeX-master**

The master file associated with the current buffer. If the file being edited is actually included from another file, you can tell AUC TeX the name of the master file by setting this variable. If there are multiple levels of nesting, specify the top level file.

If this variable is `nil`, AUC TeX will query you for the name.

If the variable is `t`, AUC TeX will assume the file is a master file itself.

If the variable is `'shared`, AUC TeX will query for the name, but not change the file.

It is suggested that you use the File Variables (see section 'File Variables' in The Emacs Editor) to set this variable permanently for each file.

User Option: **TeX-one-master**

Regular expression matching ordinary TeX files.

You should set this variable to match the name of all files, where automatically adding a file variable with the name of the master file is a good idea. When AUC TeX adds the name of the master file as a file variable, it does not need to ask next time you edit the file.

If you dislike AUC TeX automatically modifying your files, you can set this variable to `"<none>"'. By default, AUC TeX will modify any file with an extension of `.tex'.

AUC TeX keeps track of macros, environments, labels, and style files that are used in a given document. For this to work with multifile documents, AUC TeX has to have a place to put the information about the files in the document. This is done by having an `` `auto' `` subdirectory placed in the directory where your document is located. Each time you save a file, AUC TeX will write information about the file into the `` `auto' `` directory. When you load a file, AUC TeX will then read the information in the `` `auto' `` directory about the file you loaded, *and the master file specified by* `TeX-master`. Since the master file (perhaps indirectly) includes all other files in the document, AUC TeX will get information from all files in the document. This means that you will get from each file e.g. completion for all labels defined anywhere in the document.

You must create the `` `auto' `` directory manually for this to work. If you don't do this, the other files in the document will not know anything about each other, except for the name of the master file. See section Automatic Customization for a Directory.

Automatic Parsing of TeX files.

AUC TeX depends heavely on being able to extract information from the buffers by parsing them. Since parsing the buffer can be somewhat slow, AUC TeX will store the parsed information in an `` `auto' `` subdirectory in the directory where the TeX files are stored, see section Automatic Customization for a Directory. The information in the `` `auto' `` directory is also useful for multifile documents see section Multifile Documents, since it allows each file to access the parsed information from all the other files in the document. This is done by first reading the information from the master file, and then recursively the information from each file stored in the master file.

When you issue the first AUC TeX command on a new buffer, AUC TeX will try to read the information about the file from the `` `auto' `` directory. If this fails, for example if you have not edited the file before, it will parse the current buffer.

When you save the buffer it will be parsed again, and the information will be placed in the `` `auto' `` directory for use next time the file -- or another file in the document -- is edited.

It is possible to disable the parsing by setting the buffer local variables `TeX-parse-self` and `TeX-auto-save`.

```
(setq-default TeX-parse-self nil) ; Disable parse on load.
(setq-default TeX-auto-save nil) ; Disable parse on save.
```

This can also be done on a per file basis, by changing the file local variables.

```
% Local Variables:
```


```
% TeX-parse-self: nil
% TeX-auto-save: nil
% End:
```

Even when you have disabled the automatic parsing, you can force the generation of style information by pressing C-c C-n.

User Option: **TeX-parse-self**

Parse file after loading it if no style hook is found for it.

User Option: **TeX-auto-save**

Automatically save style information when saving the buffer.

Command: **TeX-normal-mode** *arg*

(C-c C-n) Remove all information about this buffer, and apply the style hooks again. Save buffer first including style information. With optional argument, also reload the style hooks.

Instead of disabling the parsing entirely, you can also speed it significantly up by limiting the information it will search for (and store) when parsing the buffer. You can do this by setting the default values for the buffer local variables `TeX-auto-regexp-list` and `TeX-auto-parse-length` in your `` `.emacs' `` file.

```
;; Only parse \documentstyle information.
(setq-default TeX-auto-regexp-list 'LaTeX-auto-minimal-regexp-list)
;; The documentstyle command is usually near the beginning.
(setq-default TeX-auto-parse-length 2000)
```

This example will speed the parsing up significantly, but AUC TeX will not longer be able to provide completion for labels, macros, environments, or bibitems specified in the document, not will it know what files belong to the document.

These variables can also be specified on a per file basis, by changing the file local variables.

```
% Local Variables:
% TeX-auto-regexp-list: TeX-auto-full-regexp-list
% TeX-auto-parse-length: 999999
% End:
```

User Option: **TeX-auto-regexp-list**

List of regular expresions used for parsing the current file.

User Option: **TeX-auto-parse-length**

Maximal length of TeX file that will be parsed.

The pre-specified lists of regexps are defined below. You can use these before loading AUC TeX by quoting them, as in the example above.

Constant: **LaTeX-auto-minimal-regexp-list**

Only parse documentstyle.

Constant: **LaTeX-auto-label-regexp-list**

Only parse LaTeX labels.

Constant: **LaTeX-auto-regexp-list**

Parse common LaTeX commands.

Constant: **plain-TeX-auto-regexp-list**

Parse common plain TeX commands.

Constant: **TeX-auto-full-regexp-list**

Parse all TeX and LaTeX commands that AUC TeX can use.

The Anti-American Conspiracy

AUC TeX contains some support for ease the use of various European character sets. This support is contained in two packages "Keymap Mode" in `min-key.el', which support general remapping of the keyboard, as well as converting between various character set encodings, and "Dead Key Mode" in `ltx-dead.el', which support dead keys for entering accents. Both packages depend on the "Minor Mode Keymap" support in `min-bind.el', but at least "Keymap Mode" is otherwise useful even outside AUC TeX.

If you have the necessary software, AUC TeX will also work with some Japanese versions of TeX and Emacs.

Support for the Scandinavian Languages

For some languages, it makes sense to remap the keyboard. This is true for languages that add a small set of regular letters compared to the English character set. The Scandinavian languages belong to this category. Danish, Norwegian, and Swedish each add three extra letters compared to English, all of which are equal to the other letters in the alphabet. They are **not** considered accented versions of the "regular" letters.

Since these letters have a well defined place on any Scandinavian keyboard, it makes sense to remap these keys to insert the equivalent TeX macros. This can also be done on a keyboard with international layout, just map the keys at the locations where the national letters normally are found.

In order to get the original binding of the keys, we can use a little trick. If you press one of the national keys twice, the national letter will first be inserted, and then replaced with the original binding of the key. This works since these national letters practically never occur twice in row in a Scandinavian word.

The keyboard mapping is implemented as a minor mode. To enter the minor mode for the first time, type M-x keyboard-query-mapping. It will first prompt you for a language to use. Currently only Danish, Swedish, German, and Russian are defined, so you have to chose one of those. Norway use the same character set as Denmark.

Next, it will prompt you for what encoding of the language's character set to use for the keyboard and screen. The same encodings are available for both the keyboard and screen, but they are not equally good. Choosing e.g. TeX for the keyboard encoding is not very useful. Below is a table of currently supported encodings.

`ISO 646 (DK or SE)'

> If your terminal thinks {, |, and } are letters, this is the encoding for you. You can use it for either Keyboard or Screen encoding.

`Keyboard Layout'

> If you have an international keyboard layout, use this for the keyboard encoding. If you do not have an international keyboard layout -- but wish you had -- you can use it for the Screen encoding instead.

`Digraphs'

Two letter combinations that form poor substitutes for the real national letters. It can be used as the Screen encoding, if you for example are sending electronic mail to someone on an unknown machine.

`TeX'

This is what you want to use for the Screen encoding when writing a TeX document.

`ISO 8859 Latin 1'

Only useful if you have hacked Emacs to understand 8-bit characters.

Finally, it will ask you if you want to get the original binding by pressing a key twice. For the Scandinavian languages, you want this.

Command: **keyboard-query-mapping**

(M-x keyboard-query-mapping) Query the user for what mapping to use, and use it.

After you have entered `keyboard-mode` once, you can leave and enter it with M-x keyboard-mode. It will use the same mapping as last time you called `keyboard-query-mapping`.

Command: **keyboard-query-mapping**

(M-x keyboard-mode) Toggle keyboard-mode. With prefix arg, turn keyboard-mode on iff arg is positive.

This is still inconvenient, so you may want to define a special function to turn your favorite mapping on and of, as shown in the following example.

```
(defun tex-dk-double-mode (arg)
  "DK TeX keys on international keyboard."
  (interactive "P")
  (require 'min-key)
  (if (or (and (null arg) keyboard-mode)
          (<= (prefix-numeric-value arg) 0))
      (keyboard-mode -1)
    (keyboard-set-mapping "Danish" "Keyboard Layout" "TeX" t)
    (keyboard-mode 1)))
```

To enter new character sets or new encodings, you must edit the variable `language-encodings` in the file `min-key.el'. Please send any new useful additions to me.

User Option: **language-encodings**

Language dependent mapping between different encodings.

This list determines how to map between different encodings of the character set of a language.

Each element in the list is itself a list.

The first element is the name of the language.

The second element is a short name of the language.

The third element is a list of short strings, each representing a possible encoding of the language's character set.

The fourth element is a list of long strings, each representing the same encodings of the language's character set.

The remaining elements each represent the encodings of one character. The encodings must appear in the same

sequence as they were found in the second element.

Support for Russian

See section Support for the Scandinavian Languages for a general explanation of keyboard remapping.

For Russian it not only makes sense to remap the keyboard. You have to do it, in order to get the Cyrillic character set from an ordinary QWERTY keyboard. AUC TeX will map QWERTY into the Russian keyboard layout YAWERTY, either with AMS Cyrillic encoding or the KOI8 character set encoding. The standard Russian keyboard layout JCUKEN is not implemented, contributions welcome.

The Russian character set encodings were kindly contributed by Justin R. Smith `<jsmith@king.mcs.drexel.edu>', please direct questions to him.

You can get a Cyrillic font for X11 from Serge Vakulenko `<vak@kiae.su>'. You will need an Emacs capable of displaying 8 bit characters to use it.

English keyboard	Russian keyboard	Mixed keyboard
qwerty	jcuken	yawerty
q Q	i-short I-short	ya Ya
w W	tse Tse	ve ve
e E	u U	ie Ie
r R	ka Ka	ar Er
t T	ie Ie	te Te
y Y	en En	iery Iery
u U	ghe Ghe	u U
i I	sha Sha	i I
o O	shcha Shcha	o O
p P	z Z	pe Pe
a A	ef Ef	a A
s S	iery Iery	es Es
d D	ve ve	de De
f F	a A	ef Ef
g G	pe Pe	ge Ge
h H	er Er	kha Kha
j J	o O	i-short I-short
k K	el El	ka Ka
l L	de De	el El
z Z	ya Ya	z Z
x X	che Che	ier Ier
c C	es Es	tse Tse
v V	em Em	zhe Zhe
b B	i I	be Be
n N	te Te	en En
m M	ierik Ierik	em Em
[{	kha Kha	sha Sha


```
|   ] }          ier Ier      shcha Shcha   |
|   ; :          zhe Zhe                    |
|   ' "        e-rev E-rev                  |
|   , <           be Be                     |
|   . >           yu Yu                     |
|   ` ~                          yu Yu      |
|   3 #                           3 ier     |
|   4 $            4 "                      |
|   5 %            5 :                      |
|   6 ^            6 ,                      |
|   7 &            7 .                      |
|   = +                        che Che      |
|   \ |                      e-rev E-rev    |
-------------------------------------------
```

Dvorak and QWERTY

QWERTY is a well established standard for how the keys on a keyboard are placed. It is almost universally accepted, most countries use it with only minor modifications to allow for any additional letters. However, it is far from optimal. It is even said that QWERTY was purposefully designed to slow down the typist, in order to protect the early typewriters.

Modern research has shown that large speedups in typing can be obtained by designing a new keyboard radically different from the current standard. Specifically, it should be split in two, and characters should be entered by pressing several keys at once.

However, some small speedups can be obtained simply by remapping the keys using the same physical layout. The Dvorak layout is an example of this. It is usually not worth the frustration of having to master a different keyboard layout, but some people have made the effort.

With `keyboard-mode` you can use QWERTY layout on a Dvorak keyboard, or vice versa. Simply type M-x keyboard-query-mapping, select the "language" `Keyboard Layout', and specify the layout on the keyboard, and what layout you want on the screen. Answer n when it asks you about pressing the key twice to get the original binding. You can then switch between QWERTY and Dvorak with M-x keyboard-mode.

See section Support for the Scandinavian Languages for a more detailed description of these functions.

Accents on Dead Keys

In some languages, like German and Dutch, you can make variants of certain letters by giving them an accent mark. A common way to enter these combined characters on a typewriter is by first typing a "dead" key which prints the accent mark without moving the printers head, and then typing the letter. In AUC TeX we simulate this by marking certain keys as dead. When you press one of those keys, it will wait for another key to be pressed. To enter the accent by itself, press the dead key twice. Examples: ' e inserts `\'{e}'. ' ' inserts `'. ' i inserts `\'{\i}'.

To enable this feature, you must first specify what accents you are interested in. This is done by setting the variable `LaTeX-dead-keys`.

User Option: **LaTeX-dead-keys**

List of characters used as dead keys. Valid characters are `', ``', `"', `^', and `~'. By default, all the valid characters are used.

This is best done in your `.emacs' file. Then, you simply enable the dead keys with M-x LaTeX-dead-mode.

Command: **LaTeX-dead-mode**

(M-x LaTeX-dead-mode) Toggle `LaTeX-dead-mode`. With prefix arg, turn `LaTeX-dead-mode` on iff arg is positive.

This mode allows users to conveniently enter characters with accents, as found in some European languages.

The supported accents are defined by `LaTeX-dead-keys`. Modify that variable to change the list of dead keys. (You most likely want to do that).

Converting Between Encodings of a Character Set

You can convert between any of the encodings for any of the supported character sets, even the special character set called `Keyboard Layout'. However, the most useful conversion is for the Latin1 character set. You convert the current region by typing convert-character-query. It will ask you for a character set and two encodings of the character set.

Command: **convert-character-query**

(convert-character-query) Convert characters in region, query user for character set and two encodings.

This function can be useful, not only when you want to import a file written with a different character set encoding, but also when you write TeX documents normally. You can use this function to edit the TeX document using a character set that your terminal will display correctly, and then convert the file to TeX format before you format it.

Japanese TeX

To write Japanese text with AUC TeX you need to have versions of TeX and Emacs that support Japanese. There exist at least two variants of TeX for Japanese text, and AUC TeX can be used with both, as well as with the two Japanese-aware Emacses, NEMACS and MULE.

To use the Japanese TeX variants, simply enter `japanese-tex-mode`, `japanese-latex-mode`, or `japanese-slitex-mode`, and everything should work. If not, send mail to Shinji Kobayashi (`<koba@flab.fujitsu.co.jp>', who kindly donated the code for supporting Japanese in AUC TeX. None of the primary AUC TeX maintainers understand Japanese, so they can not help you.

Automatic Customization

Since AUC TeX is so highly customizable, it makes sense that it is able to customize itself. The automatic customization consists of scanning TeX files and extracting symbols, environments, and things like that.

The automatic customization is done on three different levels. The global level is the level shared by all users at your site, and consists of scanning the standard TeX style files, and any extra styles added locally for all users on the site. The private level deals with those style files you have written for your own use, and use in different documents. You may have a `~/lib/TeX/' directory where you store useful style files for your own use. The local level is for a specific directory, and deals with writing customization for the files for your normal TeX documents.

If you compare with the environment variable `TEXINPUTS`, the global level corresponds to the directories built into TeX. The private level corresponds to the directories you add yourself, except for `.', which is the local level.

By default AUC TeX will search for customization files in all the global, private, and local style directories, but you can also set the path directly. This is useful if you for example want to add another person's style hooks to your p Please note that all matching files found in `TeX-style-path` are loaded, and all hooks defined in the files will be executed.

User Option: **TeX-style-path**

List of directories to search for AUC TeX style files. Each must end with a slash.

By default, when AUC TeX searches a directory for files, it will recursively search through subdirectories.

User Option: **TeX-file-recurse**

If not nil, search TeX directories recursively.

By default, AUC TeX will ignore files name `` `.' ``, `` `..' ``, `` `SCCS' ``, `` `RCS' ``, and `` `CVS' ``.

User Option: **TeX-ignore-file**

Regular expression matching file names to ignore.

These files or directories will not be considered when searching for TeX files in a directory.

Automatic Customization for the Site

Assuming that the automatic customization at the global level was done when AUC TeX was installed, your choice is now. Will you use it? If you use it, you will benefit by having access to all the symbols and environments available for completion purposes. The drawback is slower load time when you edit a new file and perhaps too many confusing symbols when you try to do a completion.

You can disable the automatic generated global style hooks by setting the variable `TeX-auto-global` to nil.

User Option: **TeX-macro-global**

Directories containing the site's TeX style files.

User Option: **TeX-style-global**

Directory containing hand generated TeX information. Must end with a slash.

These correspond to TeX macros shared by all users of a site.

User Option: **TeX-auto-global**

Directory containing automatically generated information.

For storing automatic extracted information about the TeX macros shared by all users of a site.

Automatic Customization for a User

You should specify where you store your private TeX macros, so AUC TeX can extract their information. The extracted information will go to the directory `TeX-auto-private`

Use M-x TeX-auto-generate to extract the information.

User Option: **TeX-macro-private**

Directories where you store your personal TeX macros. Each must end with a slash.

User Option: **TeX-auto-private**

Directory containing automatically generated information. Must end with a slash.

These correspond to the personal TeX macros.

Command: **TeX-auto-generate** *TEX AUTO*

(M-x TeX-auto-generate) Generate style hook for TEX and store it in AUTO. If TEX is a directory, generate style hooks for all files in the directory.

User Option: **TeX-style-private**

Directory containing hand generated information. Must end with a slash.

These correspond to the personal TeX macros.

Automatic Customization for a Directory

AUC TeX can update the style information about a file each time you save it, and it will do this if the directory `TeX-auto-local` exist. `TeX-auto-local` is by default set to `'"auto/"', so simply creating an `` `auto' `` directory will enable automatic saving of style information.

The advantage of doing this is that macros, labels, etc. defined in any file in a multifile document will be known in all the files in the document. The disadvantage is that saving will be slower. To disable, set `TeX-auto-local` to nil.

User Option: **TeX-style-local**

Directory containing hand generated TeX information. Must end with a slash.

These correspond to TeX macros found in the current directory.

User Option: **TeX-auto-local**

Directory containing automatically generated TeX information. Must end with a slash.

These correspond to TeX macros found in the current directory.

Writing Your own Style Support

See section Automatic Customization for a discussion about automatically generated global, private, and local style files. The hand generated style files are equivalent, except that they by default are found in `` `style' `` directories instead of `` `auto' `` directories.

If you write some useful support for a public TeX style file, please send it to us.

A Simple Style File

Here is a simple example of a style file.

```
;;; book.el - Special code for book style.
```


```
(TeX-add-style-hook "book"
 (function (lambda () (setq LaTeX-largest-level
                            (LaTeX-section-level ("chapter"))))))
```

This file specifies that the largest kind of section in a LaTeX document using the book document style is chapter. The interesting thing to notice is that the style file defines an (anonymous) function, and adds it to the list of loaded style hooks by calling `TeX-add-style-hook`.

The first time the user indirectly tries to access some style specific information, such as the largest sectioning command available, the style hooks for all files directly or indirectly read by the current document is executed. The actual files will only be evaluated once, but the hooks will be called for each buffer using the style file.

Function: **TeX-add-style-hook** *style hook*

Add hook to the list of functions to run when we use the TeX file style.

Adding Support for Macros

The most common thing to define in a style hook is new symbols (TeX macros). Most likely along with a description of the arguments to the function, since the symbol itself can be defined automatically.

Here are a few examples from `` `latex.el' ``.

```
(TeX-add-style-hook "latex"
 (function
  (lambda ()
     (TeX-add-symbols
     '("arabic" TeX-argument-counter-hook)
     '("label" TeX-argument-define-label-hook)
     '("ref" TeX-argument-label-hook)
     '("newcommand" TeX-argument-define-macro-hook [ "Number of arguments" ] t)
     '("newtheorem" TeX-argument-define-environment-hook
       [ TeX-argument-environment-hook "Numbered like" ]
       t [ TeX-argument-counter-hook "Within counter" ])))))
```

Function: **TeX-add-symbols** *symbol ...*

Add each symbol to the list of known symbols.

Each argument to `TeX-add-symbols` is a list describing one symbol. The head of the list is the name if the symbol, the remaining elements describe each argument.

If there are no additional elements, the symbol will be inserted with point inside braces. Otherwise, each argument of this function should match an argument of the TeX macro. What is done depends on the argument type.

If a macro is defined multiple times, AUC TeX will chose the one with the longest definition (i.e. the one with the most arguments).

Thus, to overwrite

```
        '("tref" 1) ; one argument
```

you can specify


```
'("tref" TeX-argument-label-hook ignore) ; two arguments
```

`ignore` is a function that does not do anything, so when you insert a `tref' you will be prompted for a label and no more.

`string`
: Use the string as a prompt to prompt for the argument.

`number`
: Insert that many braces, leave point inside the first.

`nil`
: Insert empty braces.

`t`
: Insert empty braces, leave point between the braces.

`other symbols`
: Call the symbol as a function. You can define your own hook, or use one of the predefined argument hooks.

`list`
: If the car is a string, insert it as a prompt and the next element as initial input. Otherwise, call the car of the list with the remaining elements as arguments.

`vector`
: Optional argument. If it has more than one element, parse it as a list, otherwise parse the only element as above. Use square brackets instead of curly braces, and is not inserted on empty user input.

A lot of argument hooks have already been defined. The first argument to all hooks is a flag indicating if it is an optional argument. It is up to the hook to determine what to do with the remaining arguments, if any. Typically the next argument is used to overwrite the default prompt.

`TeX-argument-conditional-hook`
: Implements if EXPR THEN ELSE. If EXPR evaluates to true, parse THEN as an argument list, else parse ELSE as an argument list.

`TeX-argument-literal-hook`
: Insert its arguments into the buffer. Used for specifying extra syntax for a macro.

`TeX-argument-free-hook`
: Parse its arguments but use no braces when they are inserted.

`TeX-argument-eval-hook`
: Evaluate arguments and insert the result in the buffer.

`TeX-argument-file-hook`
: Prompt for a tex or sty filename, and use it without the extension. Run the file hooks defined for it.

`TeX-argument-label-hook`
: Prompt for a label completing with known labels.

`TeX-argument-macro-hook`
: Prompt for a TeX macro with completion.

`TeX-argument-environment-hook`
: Prompt for a LaTeX environment with completion.

`TeX-argument-cite-hook`
: Prompt for a BibTeX citation.

`TeX-argument-counter-hook`
: Prompt for a LaTeX counter.

`TeX-argument-savebox-hook`
: Prompt for a LaTeX savebox.

`TeX-argument-file-hook`
: Prompt for a filename in the current directory, and use it without the extension.

`TeX-argument-input-file-hook`
: Prompt for a filename in the current directory, and use it without the extension. Run the style hooks for the file.

`TeX-argument-define-label-hook`
: Prompt for a label completing with known labels. Add label to list of defined labels.

`TeX-argument-define-macro-hook`
: Prompt for a TeX macro with completion. Add macro to list of defined macros.

`TeX-argument-define-environment-hook`
: Prompt for a LaTeX environment with completion. Add environment to list of defined environments.

`TeX-argument-define-cite-hook`
: Prompt for a BibTeX citation.

`TeX-argument-define-counter-hook`
: Prompt for a LaTeX counter.

`TeX-argument-define-savebox-hook`
: Prompt for a LaTeX savebox.

`TeX-argument-corner-hook`
: Prompt for a LaTeX side or corner position with completion.

`TeX-argument-lr-hook`
: Prompt for a LaTeX side with completion.

`TeX-argument-tb-hook`
: Prompt for a LaTeX side with completion.

`TeX-argument-pagestyle-hook`
: Prompt for a LaTeX pagestyle with completion.

`TeX-argument-verb-hook`
: Prompt for delimiter and text.

`TeX-argument-pair-hook`
: Insert a pair of numbers, use arguments for prompt. The numbers are surrounded by parentheses and separated with a comma.

`TeX-argument-size-hook`
: Insert width and height as a pair. No arguments.

`TeX-argument-coordinate-hook`
: Insert x and y coordinates as a pair. No arguments.

If you add new hooks, you can assume that point is placed directly after the previous argument, or after the macro name if this is the first argument. Please leave point located after the argument you are inserting. If you want point to be located somewhere else after all hooks have been processed, set the value of `exit-mark`. It will point nowhere, until the argument hook sets it.

Adding Support for Environments

Adding support for environments is very much like adding support for TeX macros, except that each environment normally only takes one argument, an environment hook. The example is again a short version of `latex.el'.

```
(TeX-add-style-hook "latex"
 (function
  (lambda ()
    (LaTeX-add-environments
     '("document" LaTeX-document-hook)
     '("enumerate" LaTeX-item-hook)
     '("itemize" LaTeX-item-hook)
     '("list" LaTeX-list-hook)))))
```

The only hook that is general useful is `LaTeX-item-hook`, which is used for environments that contain items. It is completely up to the environment hook to insert the environment, but the function `LaTeX-insert-environment` may be to some help. The hook will be called with the name of the environment as its first argument, and extra arguments can be provided by adding them to a list after the hook.

If an environment is defined multiple times, AUC TeX will chose the one with the longest definition. Thus, if you have an enumerate style file, and want it to replace the standard LaTeX enumerate hook above, you could define an `enumerate.el' file as follows, and place it in the appropriate style directory.

```
(TeX-add-style-hook "latex"
 (function
  (lambda ()
    (LaTeX-add-environments
     '("enumerate" LaTeX-enumerate-hook foo)))))

(defun LaTeX-enumerate-hook (environment &optional ignore) ...)
```

The symbol `foo` will be passed to `LaTeX-enumerate-hook` as the second argument, but since we only added it to overwrite the definition in `latex.el' it is just ignored.

Function: **LaTeX-add-environments** *env ...*

Add each env to list of loaded environments.

Function: **LaTeX-insert-environment** *env [extra]*

Insert environment of type env, with optional argument extra.

Adding Other Information

You can also specify bibliographical databases and labels in the style file. This is probably of little use, since this information will usually be automatically generated from the TeX file anyway.

Function: **LaTeX-add-bibliographies** *bibliography ...*

Add each bibliography to list of loaded bibliographies.

Function: **LaTeX-add-labels** *label ...*

Add each label to the list of known labels.

Supporting Other Format Packages

You can add support for new TeX format packages. These are treated specially, in that they are loaded immediately after you enter `VirTeX-mode`, and they actually define the major mode.

Here is the file defining `slitex-mode`.

```
;;;  SLITEX.el - Special code for SliTeX mode.

(require 'tex-init)
(require 'ltx-misc)

(TeX-add-style-hook "SLITEX"
 (function
  (lambda ()
    (setq mode-name "SliTeX")
    (setq major-mode 'SliTeX-mode)
    (setq TeX-command-default TeX-command-SliTeX)
    (LaTeX-mode-initialization)
    (run-hooks 'SliTeX-mode-hook)
    (TeX-run-style-hooks "latex" "slitex"))))
```

The file should be placed in the directory `TeX-format-directory`.

User Option: **TeX-format-directory**

Directory containing information about TeX format packages. Must end with a slash.

You must also add an entry to `TeX-format-list` in order to make it known to the rest of the system.

User Option: **TeX-format-list**

List of format packages to consider when choosing a TeX mode.

A list with an entry for each format package available at the site.

Each entry is a list with three elements.

1. The name of the format package.
2. The name of the major mode.
3. A regexp typically matched in the beginning of the file.

When entering `tex-mode`, each regexp is tried in turn in order to find what major mode to enter.

Automatic Extraction of New Things

The automatic TeX information extractor works by searching for regular expressions in the TeX files, and storing the matched information. You can add support for new constructs to the parser, something that is needed when you add new commands to define symbols.

For example, in the file `` `macro.tex' `` I define the following macro.


```
\newcommand{\newmacro}[5]{%
\def#1{#3\index{#4@#5~cite{#4}}\nocite{#4}}%
\def#2{#5\index{#4@#5~cite{#4}}\nocite{#4}}%
}
```

AUC TeX will automatically figure out that `newmacro' is a macro that takes five arguments. However, it is not smart enough to automatically see that each time we use the macro, two new macros are defined. We can specify this information in a style hook file.

```
;;; macro.el - Special code for my own macro file.

;;; New Macro Definition Command

(require 'tex-auto)

(defvar TeX-newmacro-regexp
  '("\\\\newmacro{\\\\\\([a-zA-Z]+\\)}{\\\\\\([a-zA-Z]+\\)}"
    (1 2) TeX-auto-multi)
  "Matches \newmacro definitions.")

(defvar TeX-auto-multi nil
  "Temporary for parsing \newmacro definitions.")

(defun TeX-macro-cleanup ()
  "Support for \\newmacro macro."

  (mapcar (function (lambda (list)
                      (mapcar (function (lambda (symbol)
                                          (setq TeX-auto-symbol
                                                (cons symbol
                                                      TeX-auto-symbol))))
                              list)))
          TeX-auto-multi))

(setq TeX-auto-prepare-hooks (cons (function (lambda ()
                                               (setq TeX-auto-multi nil)))
                                   TeX-auto-prepare-hooks))

(setq TeX-auto-cleanup-hooks (cons 'TeX-macro-cleanup TeX-auto-cleanup-hooks))

;;; HOOK

(TeX-add-style-hook "macro"
 (function
  (lambda ()
    (TeX-auto-add-regexp TeX-newmacro-regexp)
    (TeX-add-symbols '("newmacro"
                       TeX-argument-macro-hook
                       (TeX-argument-macro-hook "Capitalized macro: \\")
                       t
```


```
                "BibTeX entry: "
                nil)))))
```

When this file is first loaded, it adds a new entry to `TeX-newmacro-regexp`, and defines a function to be called before the parsing starts, and one to be called after the parsing is done. It also declares a variable to contain the data collected during parsing. Finally, it adds a style hook which describes the `newmacro' macro, as we have seen it before.

So the general strategy is: Add new entry to `TeX-newmacro-regexp`. Declare variable to contain intermediate data during parsing. Add hook to be called before and after parsing. In this case, the hook before parsing just initialize the variable, and the hook after parsing collect the data from the variable, and add them to the list of symbols found.

Variable: **TeX-auto-regexp-list**

List of regular expressions matching TeX macro definitions.

The list has the following format ((REGEXP MATCH TABLE) ...), that is, each entry is a list with three elements.

REGEXP. Regular expression matching the macro we want to parse.

MATCH. A number or list of numbers, each representing one parenthesized subexpression matched by REGEXP.

TABLE. The symbol table to store the data. This can be a function, in which case the function is called with the argument MATCH. Use `TeX-match-buffer` to get match data. If it is not a function, it is presumed to be the name of a variable containing a list of match data. The matched data (a string if MATCH is a number, a list of strings if MATCH is a list of numbers) is put in front of the table.

Variable: **TeX-auto-prepare-hooks** *nil*

List of hooks to be called before parsing a TeX file.

Variable: **TeX-auto-cleanup-hooks** *nil*

List of hooks to be called after parsing a TeX file.

Installation of AUC TeX

Before you do anything else, make sure that you have the latest version of TeXinfo installed. It is available by ftp from `prep.ai.mit.edu'. You need at least version 2.16.

First, you should consult the file `` `tex-site.el' `` and edit it to fit your local site. Be sure to get the following two variables right, or you will not be able to complete the installation procedure:

User Option: **TeX-lisp-directory**

The directory where you want to install the AUC TeX lisp files.

User Option: **TeX-macro-global**

Directories containing the site's TeX style files.

You probably also need to change `TeX-command-list` to make sure that the commands used for starting TeX, printing, etc. work on your system.

Finally, edit `TeX-printer-list` to contain the printers available at your site.

Next, edit the file `` `Makefile' `` in the AUC TeX directory to set up paths for installation of the files. Be sure that

`bindir`
: Is set to the directory where you want the LaCheck binary to be installed.

`infodir`
: Is set to the directory where you want the AUC TeX "info" documentation to be installed.

`aucdir`
: Is set to the directory where you want the lisp files for AUC TeX to be installed. This **must** be the same directory as you specified for `TeX-lisp-directory` in `` `tex-site.el' ``. If you unpacked AUC TeX in that directory (`TeX-lisp-directory`), you should set `aucdir` to `.' in order to avoid copying the byte compiled lisp files.

  If you already have an old version of AUC TeX in that directory, you must delete it before installing the new version. Otherwise the wrong version of the files may be loaded during the byte compilation.

`mandir`
: Is set to the directory where you want to install the (unformatted) man page for LaCheck.

`elispdir`
: Is set to the directory where the lisp files from the standard Emacs distribution are found.

`CC`
: Is the compiler you want to use for compiling the LaCheck program. Is set to `gcc' by default. Change this is `cc' or whatever your system compiler is called if you do not have GCC installed.

Then type make all, and the whole thing will be `made'. If this succeeds, type make install, and it will be installed.

To extract information from your site's TeX macros, type make install-auto. This will only work if you have set `TeX-macro-global` correctly in `` `tex-site.el' ``.

Tell the users on your site to insert

```
(load-file "/path/for/auctex/tex-site.elc")
```

in their `` `.emacs' `` file in order to use AUC TeX.

Users who know the older keybindings from version 6.1 may also want to insert

```
(load "auc-tex")
```

in their `` `.emacs' ``

The History of AUC TeX

This chapter is for old users, who want to see what has been added in this version, and for the authors, who get nostalgic at times.

News in 7.3

Coordinater: Per Abrahamsen, 1993.

- More robust installation, especially for Lucid Emacs (I hope). Many people reported problems with this.
- Make `easymenu' work when byte-compiled. Many people reported this bug.
- Minimally updated the `` `README' `` file from version 6.0 (sigh). Thanks to Boris Goldowsky

`<boris@cs.rochester.edu>' for reporting this.

- Added `@finalout' to manual. Reported by Henrik Drabol `<hvd@ens004.ens.min.dk>'.
- Fixed M-q to work after an `\end{...}'. It will not work at the end of the buffer, but there are usually the local variables so it should (hopefully) not matter. Thanks to Shinji Kobayashi `<koba@flab.fujitsu.co.jp>' again.
- New variables `TeX-open-quote` and `TeX-close-quote` determine what is inserted by `TeX-insert-quote`. The `german' style file now use those variables instead of changing the keymap.
- Changes to the default settings in `tex-site.el', in particular a `Queue' command is added to display the print queue. Thanks to John Interrante `<interran@uluru.Stanford.EDU>' for code, and other members of the `auc-tex@iesd.auc.dk' mailing list for ideas.
- Make sure all outline mode commands are bound in `outline-minor-mode`.
- Added autoload for `TeX-command`. Thanks to Hanno Wirth `<wirth@igd.fhg.de>' for reporting this.
- Added support for AmSTeX and AmSLaTeX. Currently they are identical to TeX and LaTeX except for another default command.
- Added VorTeX style matching of dollar sign. The style is guaranteed to be VorTeX, since I lifted the code directly from VorTeX. Thanks to Pehong Chen `<phc@renoir.berkeley.edu>' for writing the VorTeX code. Thanks to Jak Kirman `<jak@cs.brown.edu>' for pointing out this nice VorTeX feature.
- Added information about AUC TeX mail addresses to the manual. Thanks to Dave Smith `<maa507@computing.lancaster.ac.uk>'.
- Added menu to for plain TeX. Suggested by Tim Carlson `<imsgtcar@sinc.oscs.montana.edu>'.
- Made the menus depend on `TeX-command-list`.
- Made it possible to specify `TeX-auto-regexp-list` in the local variable section of each file.
- Added variable `TeX-auto-parse-length` to specify maximal length of text that will be parsed.
- Added automatic parsing of BibTeX files and `bibitem' entries in order to get completion in `cite'. This was inspired by an add on made by Sridhar Anandakrishnan `<sak@essc.psu.edu>'.
- Added variable `TeX-byte-compile` to disable automatic byte compilation of style files when loaded. This is needed when using different Emacs versions.
- Added variable `TeX-translate-location-hook` to translate file and line information before showing an error, as requested by Thorbjoern Ravn Andersen `<ravn@imada.ou.dk>'.
- Added variable `TeX-auto-save` to allow disabling the automatic saving of style information, either per file in the file local variables, or globally by using `setq-default`. Use `TeX-normal-mode` to force style information to be saved.
- Try to create `auto' directory if it does not exists.
- Added chapter describing how to tune the TeX parsing.
- Allow (but do not encourage) a string value for `LaTeX-default-options`.
- Give `"' word syntax when german.sty is loaded. Suggested by Tim Geisler `<tmgeisle@immd8.informatik.uni-erlangen.de>'.
- Many corrections to the grammar in the manual. Thanks to Manfred Weichel `<Manfred.Weichel@sto.mchp.sni.de>'.
- Bind `TeX-home-buffer` to C-c ^ instead of C-c C-h which are reserved in Emacs 19. Suggested by Chris Moore `<Chris.Moore@src.bae.co.uk>'.

News in 7.2

Coordinater: Per Abrahamsen, 1993.

- `LaTeX-dead-mode` works again. Thanks to Patrick O'Callaghan `<poc@usb.ve>' for fixing it.
- Minor fixes to the documentation. Thanks to Shinji Kobayashi `<koba@flab.fujitsu.co.jp>'.
- Add `Compiling' to the mode line of all buffers, while there is a AUC TeX compilation process running. This is similar to the behavior of `compile` in Emacs 19.
- `TeX-normal-mode` will now save the buffer first to make sure it gets reparsed.
- Labels with underscores are now recognized. Thanks to Wolfgang Franzki `<W.Franzki@kfa-juelich.de>'
- Fix to `ghostview' printer specification. Thanks to Masayuki Kuwada `<kuwada@soliton.ee.uec.ac.jp>'.
- Recognize `abstract', `center', `titlepage', `verse', and `theindex' environments. Thanks to Masayuki Kuwada `<kuwada@soliton.ee.uec.ac.jp>'.
- Fix to `newsavebox' macro. Thanks to Shinji Kobayashi `<koba@flab.fujitsu.co.jp>' for reporting this.
- Menu support for GNU Emacs 19 and Lucid Emacs. Thanks to Alastair Burt `<burt@dfki.uni-kl.de>' for the initial Lucid Emacs version.
- C-c C-f C-d now deletes the current font. The current font is defined to be the innermost TeX group starting with a TeX macro that is terminated by a space.
- Giving C-c C-f a prefix argument will replace the current font, i.e. C-u C-c C-f C-b will change the current font to bold.

  The old functionality (putting the font around the region) has been removed. To make the region bold, type C-w C-c C-f C-b C-y instead.
- Chapter recognized as largest heading in the report style. Thanks to Shinji Kobayashi `<koba@flab.fujitsu.co.jp>' for reporting this.
- More support for Japanese style files. Thanks to Shinji Kobayashi `<koba@flab.fujitsu.co.jp>'.
- No longer put `Outline' in the mode line whenever `selective-display` is set. Thanks to Lawrence R. Dodd `<dodd@roebling.poly.edu>' for reporting this.
- Support for inserting calligraphic letters in `TeX-math-mode` with ` c LETTER. Thanks to Olaf Burkart `<burkart@zeus.informatik.rwth-aachen.de>'.
- `set-docstring` in `tex-math.el' should work better now. Thanks to Alastair Burt `<burt@dfki.uni-kl.de>' and Olaf Burkart `<burkart@zeus.informatik.rwth-aachen.de>'.
- Support for dviout preview on PC-9801. Thanks to Shinji Kobayashi `<koba@flab.fujitsu.co.jp>'.
- Inserting environment in empty buffer should work now. Thanks to Alastair Burt `<burt@dfki.uni-kl.de>'.
- Default float for figures changed from `tbp' to `htbp'.
- `LaTeX-format-environment` may work now. Thanks to Shinji Kobayashi `<koba@flab.fujitsu.co.jp>'.
- Better `LaTeX-close-environment`. Thanks to Thorbjoern Hansen `<thansen@diku.dk>'.
- Some support for Ispell 4.0.
- BibTeX in Emacs 19 need `tex-insert-quote`, make it autoload from AUC TeX instead of the standard `tex-mode`.
- `TeX-auto-generate` failed when repeated. Thanks to Peter Whaite `<peta@Thunder.McRCIM.McGill.EDU>' for reporting this.

News in 7.1

Coordinater: Per Abrahamsen, 1993.

- Allow multiple `%p' in print commands.

  Suggested by Cliff Krumvieda `<cliff@cs.cornell.edu>'.
- Improved backward compatibility in `auc-tex.el'. Thanks to Ralf Handl `<handl@cs.uni-sb.de>'.
- New style hook for `german.sty'.

  Disable smart quotes. Press C-c C-n to make it take effect.
- Allow files to have other extensions than "tex".

  But no longer allow files to have multiple dots. Sigh.
- Will no longer parse the buffer if it can use the saved state.
- New variable TeX-parse-self.

  Set it to nil if you never want to parse the buffer when you load it.
- Only offer to save files that belongs to the document.

  When you format the document with C-c C-c, AUC TeX will no longer offer to save your `RMAIL', `.newsrc', or other files that does not belong to the document. Suggested by Jim Hetrick `<hetrick@phys.uva.nl>'.
- FoilTeX support.

  Thanks to Sven Mattisson `<sven@tde.lth.se>'
- Smarter about when you need to reformat.

  Thanks to Chris Callsen `<chris@iesd.auc.dk>'.
- Japanese TeX

  Now supports Japanese TeX. Thanks to Shinji Kobayashi `<koba@keisu-s.t.u-tokyo.ac.jp>'.
- Works again under OS/2 and other case insensitive file systems.
- DEMACS support.

  Thanks to Shinji Kobayashi `<koba@keisu-s.t.u-tokyo.ac.jp>'.
- Better LaTeX-close-environment.

  Thanks to Piet van Oostrum `<piet@cs.ruu.nl>'.
- Ispell support.

  Thanks to Piet van Oostrum `<piet@cs.ruu.nl>'.
- Support for Russian letters.

  Thanks to Justin R. Smith `<jsmith@king.mcs.drexel.edu>'.
- SliTeX fixes.

  Many people.
- Fixes for spelling errors.

  Many people.

Version 7.0

Coordinator: Per Abrahamsen, 1993.

Alpha testers (in order of appearance): Piet van Oostrum `<piet@cs.ruu.nl>', Sven Mattisson `<sven@tde.lth.se>', Tim Geisler `<tmgeisle@immd8.informatik.uni-erlangen.de>', Fran E. Burstall `<F.E.Burstall@maths.bath.ac.uk>', Alastair Burt `<burt@dfki.uni-kl.de>', Sridhar Anandakrishnan `<sak@essc.psu.edu>', Kjell Gustafsson `<kjell@sccm.Stanford.EDU>', Uffe Kjaerulff `<uk@iesd.auc.dk>', Kurt Swanson `Kurt.Swanson@dna.lth.se', Mark Utting `<marku@cs.uq.oz.au>', Per Norman Oma `perno@itk.unit.no', Naji Mouawad `<nmouawad@math.uwaterloo.ca>', Bo Nygaard Bai `<bai@iesd.auc.dk>', and probably more.

- New keymap.

  The keymap has been changed in order to make it more intuitive to new users, and because the old bindings did not work well with the new buffer manipulation commands in tex-buf.el. To use the new bindings, load `tex-init.el' instead of `auc-tex.el'.

  The file `auc-tex.el' is still available and implements the old keybinding on top of the new code.

  Print out the reference card (`doc/ref-card.tex') to see the new bindings.

- Completely redesigned the buffer handling.

  No part of the interface or the customization variables remain the same, unless you use the compatibility functions in `auc-tex.el'. In that case the interactive commands remain similar in spirit, but the customization interface is still changed.

  The file `tex-buf.el' has been completely rewritten, and there are major cleanup in `tex-dbg.el', however the basic functionality remains the same in this file. The code for both `tex-buf.el' and `tex-dbg.el' should be much simpler now and easier to extent.

  `auc-tex.el' and `tex-site.el' was updated to support the new interface. I actually believe the moral equivalent to `TeX-region' to work now `:-)', at least I understand the code now.

  The two major functions are now `TeX-command-master' and `TeX-command-region'. Each function will prompt you for the command to execute. AUC TeX will make an educated guess on what command you want to run, and make that the default. The available commands are defined in the variable `TeX-command-list'.

  `TeX-command-master' will run the specified command on the buffers master file. You can have one command running for each master file. `TeX-command-region' will run the specified command on the current region, getting the header on trailer from the master file.

  You can have exactly one region command running, independent on how many master file commands that are running. Commands that operate on the active process (like `TeX-next-error') will chose the process associated with buffers master file, unless the last region process is more recent than all master file processes.

  AUC TeX now insist on knowing the master file for a buffer. If you do not specify it in the file variable section, and it is not obviously a master file itself, it will ask you. It will also add the master file name to the file variables, unless you disable this feature by setting `TeX-add-local' to nil. Furthermore, it will convert `%% Master:' lines to file variables, unless you disable it by setting `TeX-convert-master' to nil.

  Functionality removed (for now, it might appear again latter) include all other functions to start a command (e.g. `LaTeX-BibTeX'), and alternative ways to specify headers and trailers. The only place to get the header and trailer is from the master file (I can easily change that, if anyone have such needs).

- Style specific code isolated.

You can now add style specific information to AUC TeX by writing a style file somewhere in TeX-style-p

The main code is now organized around this principle.

- Automatically generate style files.

  AUC TeX can now automatically extract information from a TeX file, and will do this when you save a buffer.
- SliTeX mode.

  Just like LaTeX mode, except that the default command to format run on the buffer is `slitex'.
- `LaTeX-section` completely general.

  Rewrote `` `ltx-sec.el' ``.
  - Sectioning level, toc, and title queries can be individually turned off.
  - Label query can be turned on or off for selected sectioning levels.
  - Label prefix can be different for different sectioning levels.
  - If the title (or toc) is empty, point will be positioned there.
  - Users can add new hooks
- `TeX-insert-macro` much smarter.

  It will now prompt for the symbol with completions, and for many symbols it will also prompt for each argument. There are also completion on some of the arguments.
- Fixed center in figure environment.

  Thanks to Thomas Koenig `<ig25@rz.uni-karlsruhe.de>'.
- Changed `\M-` to `\e` in all keybindings in order to better support 8-bit input on some GNU Emacs. Thanks to Peter Dalgaard `<pd@kubism.ku.dk>'.

  Please, implementors of 8-bit input extensions to GNU Emacs. `\M-x` does *not* means x with the 8-bit set. It means pressing x while holding down the META key. Some systems (such as X11) are able to tell the different. Thus, even if you implement 256 byte keymaps, `\M-x` should still expand `meta-prefix-char` followed by an x in the keymap. This allows you to distinguish pressing x while holding down the META key from entering a literal 8-bit character.
- Made the outline commands aware of the document style.

  That is, if the document style is `article', `\section' will be one level below the `\documentstyle', while if the style is `book', `\section' will be three levels below `\documentstyle'. This will make `show-children` work better at the top level.
- The makefiles are closer to GNU coding standard.

  They now understand `prefix' and some other macros.
- Added hooks to be run after list of environments or list of completion names are updated, and also added a hook to be called after each file has been loaded. Thanks to Piet van Oostrum `<piet@cs.ruu.nl>'.
- Added `*' to lot of `(interactive)` declarations.
- The outline commands are now always accessible from LaTeX mode.
- Generalized the keyboard remapping and double modes.

  These are found in the file `` `min-key.el' ``.
- Smart Comments.

  Not really, but there are now two comment functions which use their arguments to determine what to do,

instead of four functions ignoring their arguments.

- Add outline headers.

  It is now possible to add extra outline headers, by setting the variable `TeX-outline-extra`.

- Smart quotes even smarter.

  If you press " twice, it will insert an real double quote instead of two (or four) single quotes. This is consistent with how remapping in `` `min-key.el' `` is done.

- Automatically untabify buffer when you save it.

  Hands up, everyone who have produced a `last revision' paper containing an unreadable list of data in the back, because TeX does not understands tabs.

- Call show-all when you change major mode.

  Thanks to Inge Frick's `<inge@nada.kth.se>' `` `kill-fix.el' `` enhancement, outline minor mode can now guarantee that all text is shown when you leave the minor mode, even if you leave the minor mode by changing the major mode.

- Updated documentation for 7.0.

  Also added key, variable, function, and concept indexes, as well as this history section and a new chapter on multifile documents (see section Multifile Documents).

Version 6.1

Coordinator: Per Abrahamsen, 1992.

- `TeX-region` might work now (heard that before?).

  Many people reported this one. Especially thanks to Fran Burstall `<F.E.Burstall@maths.bath.ac.uk>' and Bill Schworm `<bill@schworm.econ.ubc.ca>'.

- The specification format for the TeX command is more general.

  See the documentation for `LaTeX-command` and `plain-TeX-command`.

- The specification format for the preview commands is more general.

  See their respective documentation.

- The specification format for the print command is more general.

  See the documentation for `TeX-print-command`.

- `TeX-args` is marked as obsolete.
- The `"Emergency stop ..."' error.

  Some users of old TeX installations got might might be fixed now. Thanks to Philip Sterne `<sterne@dublin.llnl.gov>'.

- It is now possible to change the preview command.

  ... without loading TeX-site first. Thanks to Tim Bradshaw `<tim.bradshaw@edinburgh.ac.uk>'.

- New variable TeX-smart-quotes.

  Allow `` `german.sty' `` users (and others) to disable the mapping of double quote (" to `"' or `"'). Thanks to Daniel Hernandez `<danher@informatik.tu-muenchen.de>'.

- Many minor corrections to the documentation.

Thanks to Mainhard E. Mayer `<hardy@golem.ps.uci.edu>'.

- Make test for `HOSTTYPE` case insensitive.

  Thanks to Gisli Ottarsson `<gisli@liapunov.eecs.umisc.edu>'.

- `TeX-force-default-mode`

  Set to avoid AUC TeX's attempts to infer the mode of the file by itself.

Version 6.0

Coordinator: Kresten Krab Thorup, 1992.

Preliminary documentation is available in the directory `` `doc' ``. It isn't very well written, but I believe it covers most interesting points. Comments, suggestions, or even rewrites of sections are VERY WELCOME...

LaCheck has been incorporated in the package. The source code for it is available in the directory `` `lacheck' `` along with the documentation for it. Lacheck may also be used from the command line. It is bound to C-c $.

Some minor changes in:

`TeX-region`

Should work better with `Master:' option.

`LaTeX-environment`

Numerous new hooks added by Masayuki Kuwada.

`TeX-command-on-region`

Removed. C-c C-o used for `outline-minor-mode` instead.

And some additional minor fixes...

Ancient History

The origin of AUC TeX is `` `tex-mode.el' `` from Emacs 16. Lars Peter Fischer `<fischer@iesd.auc.dk>' wrote the first functions to insert font macros and Danish characters back in 1986. Per Abrahamsen `<abraham@iesd.auc.dk>' wrote the functions to insert environments and sections, to indent the text, and the outline minor mode in 1987. Kresten Krab Thorup `<krab@iesd.auc.dk>' wrote the buffer handling and debugging functions, the macro completion, and much more, including much improved indentation and text formatting functions. He also made the first public release in 1991, and was the main author and coordinator of every release up to and including 6.0.

Thanks should also go to all the people who have been a great help developing the AUC TeX system. Especially all the people on the `auc-tex' mailing list, who have been very helpful commenting and pointing out weak points and errors.

Some of the contributors are listed below. Others are mentioned in the lisp files or in the History section.

`<dduchier@csi.UOttawa.CA>'

Denys Duchier

`<ferguson@cs.rochester.edu>'

George Ferguson

`<simons@ibiza.karlsruhe.gmd.de>'

Martin Simons

`<smith@pell.anu.edu.au>'

Michael Smith
`<per@iesd.auc.dk>'
Per Hagen
`<handl@cs.uni-sb.de>'
Ralf Handl
`<sven@tde.lth.se>'
Sven Mattisson
`<kuwada@soliton.ee.uec.ac.jp>'
Masayuki Kuwada
`<tb06@pl118f.cc.lehigh.edu>'
Terrence Brannon
`<roseman@hustat.harvard.edu>'
Leonard Roseman

Special thanks to Leslie Lamport for supplying the source for the LaTeX error messages in the `tex-dbg.el' file.

Various Minor Modes

A number of minor modes are distributed with AUC TeX for no good reason, except that they use `min-bind.el' and they are so small that it does not matter.

Some of the minor modes have already been described elsewhere, the remaining will be described here.

Auto Indent Mode

This is a minor mode version of `indented-text-mode`. Enter and leave with M-x auto-indent-mode.

Command: **auto-indent-mode**

Toggle auto indent mode. With prefix arg, turn auto indent mode on iff arg is positive.

Auto indent mode works by invoking `indent-relative` for TAB, and using `indent-relative-maybe` as the `indent-line-function` for `auto-fill`, and LFD.

Auto Ispell Mode

In `auto-ispell-mode` SPC or RET automatically trigger `ispell-word` on previously typed word. M-TAB will make an ispell completion.

Enter and leave with M-x auto-ispell-mode.

Command: **auto-ispell-mode**

(M-x auto-ispell-mode) Toggle auto-ispell-mode. With prefix arg, turn `auto-ispell-mode` on iff arg is positive.

In `auto-ispell-mode`, inserting a space or a newline applies `ispell-word` on the previous word.

Pressing Esc-Tab activates ispell-complete-word.

Currently supported symbols in LaTeX-math-mode

` ?
: Help, list of math symbols.

` a
: `\alpha'

` b
: `\beta'

` d
: `\delta'

` e
: `\epsilon'

` f
: `\phi'

` g
: `\gamma'

` h
: `\eta'

` k
: `\kappa'

` l
: `\lambda'

` m
: `\mu'

` n
: `\nu'

` o
: `\omega'

` p
: `\pi'

` q
: `\theta'

` r
: `\rho'

` s
: `\sigma'

` t
: `\tau'

TOC

` u
`\upsilon'
` x
`\chi'
` y
`\psi'
` z
`\zeta'
` D
`\Delta'
` G
`\Gamma'
` Q
`\Theta'
` L
`\Lambda'
` Y
`\Psi'
` P
`\Pi'
` S
`\Sigma'
` U
`\Upsilon'
` V
`\Phi'
` O
`\Omega'
` C-f
`\rightarrow'
` C-b
`\leftarrow'
` C-p
`\uparrow'
` C-n
`\downarrow'
` <
`\leq'
` >
`\geq'
` ~

TOC

`\tilde'
` I
`\infty'
` A
`\forall'
` E
`\exists'
` !
`\not'
` i
`\in'
` *
`\times'
` .
`\cdot'
` {
`\subset'
` }
`\supset'
` [
`\subseteq'
`]
`\supseteq'
` \
`\backslash'
` /
`\setminus'
` +
`\cup'
` -
`\cap'
` (
`\langle'
`)
`\rangle'
` C-e
`\exp'
` C-s
`\sin'
` C-c
`\cos'

` C-^
    `\sup'
` C-_
    `\inf'
` C-d
    `\det'
` C-l
    `\lim'
` C-t
    `\tan'
` ^
    `\hat'
` v
    `\vee'

Wishlist

This is a list of projects for AUC TeX. Bug reports and requests we can not fix or honor right away will be added to this list. If you have some time for emacs lisp hacking, you are encouraged to try to provide a solution to one of the following problems.

- Add completion of macro arguments to `TeX-complete-symbol`.
- Make . check for abbreviations and sentences ending with capital letters. Make , and . remove italic correction.
- Make LaCheck check for itallic correction before `.' or `,'.
- Use Emacs 19 minubuffer history to choose between previewers, and other stuff. Suggested by John Interrante `<interran@uluru.Stanford.EDU>'.
- It is often too slow to parse the file each time it is saved, especially with Emacs 19. Check if `TeX-auto-generate` is useful for doing more occasionally, like updating the `` `TAGS' `` file.
- Control TeX compilation window.

  It is not good that half of the screen is wasted for the log when TeX is processing a file. It is possible to make the log screen smaller, but the best solution would be to have a user configurable window layout that could be defined by some variables as for VM or GNUS. Or a log screen that opens only in case of an error. -- Tim Geisler `<tmgeisle@immd8.informatik.uni-erlangen.de>'
- Make features.

  A new command `TeX-update` (C-c C-u) could be used to create an up to date dvi file by repeatedly running BibTeX, MakeIndex and (La)TeX, until an error occur or we are done.

  An alternative is to have an `Update' command that ensures the `dvi' file is up to date. This could be called before printing and previewing.
- Hook naming should be accoriding to the Emacs (19) guidelines. Specifically, only the hook *variable* should be names `-hook'. The hook functions should have other names.
- The documentation of the minor modes should be separated.

  They are distributed independently of AUC TeX, and should therefore also be documented independently of AUC TeX.

- Documentation of variables that can be set in a style hook.

  We need a list of what can safely be done in an ordinary style hook. You can not set a variable that AUC TeX depends on, unless AUC TeX knows that it has to run the style hooks first.

  Here is the start of such a list.

  `LaTeX-add-environments`
  `TeX-add-symbols`
  `LaTeX-add-labels`
  `LaTeX-add-bibliographies`
  `LaTeX-largest-level`

  The rules for what can be done in a format package hook are different.
- Correct indentation for table, math, and array environments.
- Customizable indentation for verbatim-like user defined environments.
- Special indentation after an `\item'.

```
\begin{itemize}
\item blabalaskdfjlas lajf adf
      lkfjl  af jasl lkf jlsdf jlf
\item f lk jldjf lajflkas flf af
\end{itemize}
```

- Completion for citations, counters, and sboxes.

  The two latter is easy the first is useful.
- Outline should be able to hide environments.

  Probably being one level below `LaTeX-lowest-level`.
- Outline should be (better) supported in TeX mode.

  At least, support headers, trailers, as well as TeX-outline-extra.
- Outline Minor Mode should support non-regexp outlines better.

  Something like TeX-outline-extra should be standard support in `` `min-out.el' ``.
- `TeX-header-start` and `TeX-trailer-end`.

  We might want these, just for fun (and outlines)
- Plain TeX and LaTeX specific header and trailer expressions.

  We should have a way to globally specify the default value of the header and trailer regexps.
- Add support for original `TeX-mode` keybindings.

  A third initialization file (`` `tex-mode.el' ``) containing an emulator of the standard `TeX-mode` would help convince some people to change to AUC TeX.
- Make `TeX-next-error` parse ahead and store the results in a list, using markers to remember buffer positions in order to be more robust with regard to line numbers and changed files. This is what `next-error` does. (Or did, until Emacs 19).
- When `LaTeX-environment` is given an argument, change the current environment. Be smart about `\item[]' versus `\item ' and labels like `fig:' versus `tab:'.
- Check out if lightening completion from Ultra TeX is anything for us.

- TeXinfo mode.
- BibTeX mode.
- Support for AMSLaTeX style files.
- Better LaTeX letter environment.

  Make a database system for addresses in the letter environment Make the letter environment hook produce `documentstyle' too
- Add code for submitting a report, using BAW's `reporter.el'.

Credit

A big smile and thanks should go to all the folks who cheered me up, during the long hours of programming... sorry folks, but I can't help including the list below, of comments I've got...

Kresten Krab Thorup

`<monheit@psych.stanford.edu>'

I'd like to say that I'm very much enjoying using auc-tex. Thanks for the great package!

`<georgiou@rex.cs.tulane.edu>'

I really enjoy working with auc-tex.

`<toy@soho.crd.ge.com>'

Thanks for your great package. It's indispensable now! Thanks!

`<ascott@gara.une.oz.au>'

Thanks for your time and for what appears to be a great and useful package. Thanks again

`<hal@alfred.econ.lsa.umich.edu>'

Thanks for providing auc-tex.

`<simons@ibiza.karlsruhe.gmd.de>'

I really enjoy using the new emacs TeX-mode you wrote. I think you did a great job.

`<clipper@csd.uwo.ca>'

I am having fun with auc-tex already.

`<ibekhaus@athena.mit.edu>'

Thanks for your work on auc-tex, especially the math-minor mode.

`<burt@dfki.uni-kl.de>'

I like your auc-tex elisp package for writing LaTeX files - I am especially impressed by the help with error correction.

`<goncal@cnmvax.uab.es>'

Thanks so much!

`<bond@sce.carleton.ca>'

I >really< like the macro, particularly the hooks for previewing and the error parsing!

`<ascott@gara.une.oz.au>'

All in all I am pleased with your package. Thanks a lot.

Key Index

"

- "_

$

- $

C

- C-c %
- C-c ;
- C-c]
- C-c ^
- C-c `
- C-c C-c
- C-c C-e
- C-c C-f
- C-c C-f C-b
- C-c C-f C-c
- C-c C-f C-e
- C-c C-f C-i
- C-c C-f C-r
- C-c C-f C-s
- C-c C-f C-t
- C-c C-k
- C-c C-l
- C-c C-m
- C-c C-n
- C-c C-o
- C-c C-o C-a
- C-c C-o C-b
- C-c C-o C-c
- C-c C-o C-e
- C-c C-o C-f
- C-c C-o C-h

l

m

t

Function Index

a

- auto-indent-mode
- auto-ispell-mode

c

- convert-character-query

h

- hide-body
- hide-entry
- hide-leaves
- hide-subtree

i

- indent-line-function
- indent-relative
- ispell-complete-word
- ispell-word

k

- keyboard-query-mapping

l

- LaTeX-add-bibliographies
- LaTeX-add-environments
- LaTeX-add-labels
- LaTeX-close-environment
- LaTeX-dead-mode
- LaTeX-environment
- LaTeX-format-environment
- LaTeX-format-paragraph
- LaTeX-format-region

o

s

t

- TeX-recenter-output-buffer
- TeX-toggle-debug-bad-boxes

Variable Index

l

- language-encodings
- LaTeX-auto-label-regexp-list
- LaTeX-auto-minimal-regexp-list
- LaTeX-auto-regexp-list
- LaTeX-dead-keys
- LaTeX-default-environment
- LaTeX-figure-label
- LaTeX-float
- LaTeX-indent-level
- LaTeX-item-indent
- LaTeX-math-abbrev-prefix
- LaTeX-mathlist
- LaTeX-section-hooks
- LaTeX-section-label
- LaTeX-table-label

o

- outline-level-function
- outline-prefix-char
- outline-regexp

p

- plain-TeX-auto-regexp-list

t

- TeX-auto-cleanup-hooks
- TeX-auto-full-regexp-list
- TeX-auto-global
- TeX-auto-local

Concept Index

TOC

.

\

a

b

c

d

e

f

g

h

TOC

i

j

k

l

m

n

o

p

q

r

S

GNU Automake

For version 1.4, 10 January 1999

David MacKenzie and Tom Tromey

This file documents the GNU Automake package for creating GNU Standards-compliant Makefiles from template files. This edition documents version 1.4.

This document was generated on 20 May 1999 using the texi2html translator version 1.51a.

Go to the first, previous, next, last section, table of contents.

Introduction

Automake is a tool for automatically generating `Makefile.in's from files called `Makefile.am'. Each `Makefile.am' is basically a series of `make' macro definitions (with rules being thrown in occasionally). The generated `Makefile.in's are compliant with the GNU Makefile standards.

The GNU Makefile Standards Document (see section `Makefile Conventions' in The GNU Coding Standards) is long, complicated, and subject to change. The goal of Automake is to remove the burden of Makefile maintenance from the back of the individual GNU maintainer (and put it on the back of the Automake maintainer).

The typical Automake input file is simply a series of macro definitions. Each such file is processed to create a `Makefile.in'. There should generally be one `Makefile.am' per directory of a project.

Automake does constrain a project in certain ways; for instance it assumes that the project uses Autoconf (see section `Introduction' in The Autoconf Manual), and enforces certain restrictions on the `configure.in' contents.

Automake requires `perl' in order to generate the `Makefile.in's. However, the distributions created by Automake are fully GNU standards-compliant, and do not require `perl' in order to be built.

Mail suggestions and bug reports for Automake to bug-automake@gnu.org.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

General ideas

The following sections cover a few basic ideas that will help you understand how Automake works.

General Operation

Automake works by reading a `Makefile.am' and generating a `Makefile.in'. Certain macros and targets defined in the `Makefile.am' instruct Automake to generate more specialized code; for instance, a `bin_PROGRAMS' macro definition will cause targets for compiling and linking programs to be generated.

The macro definitions and targets in the `Makefile.am' are copied verbatim into the generated file. This allows you to add arbitrary code into the generated `Makefile.in'. For instance the Automake distribution includes a non-standard `cvs-dist` target, which the Automake maintainer uses to make distributions from his source control system.

Note that GNU make extensions are not recognized by Automake. Using such extensions in a `Makefile.am' will lead to errors or confusing behavior.

Automake tries to group comments with adjoining targets and macro definitions in an intelligent way.

A target defined in `Makefile.am' generally overrides any such target of a similar name that would be automatically generated by `automake`. Although this is a supported feature, it is generally best to avoid making use of it, as sometimes the generated rules are very particular.

Similarly, a macro defined in `Makefile.am' will override any definition of the macro that `automake` would ordinarily create. This feature is more often useful than the ability to override a target definition. Be warned that many of the macros generated by `automake` are considered to be for internal use only, and their names might change in future releases.

When examining a macro definition, Automake will recursively examine macros referenced in the definition. For example, if Automake is looking at the content of `foo_SOURCES` in this snippet

```
xs = a.c b.c
foo_SOURCES = c.c $(xs)
```

it would use the files `a.c', `b.c', and `c.c' as the contents of `foo_SOURCES`.

Automake also allows a form of comment which is *not* copied into the output; all lines beginning with `##' are completely ignored by Automake.

It is customary to make the first line of `Makefile.am' read:

```
## Process this file with automake to produce Makefile.in
```

Depth

`automake` supports three kinds of directory hierarchy: `flat', `shallow', and `deep'.

A **flat** package is one in which all the files are in a single directory. The `` `Makefile.am' `` for such a package by definition lacks a `SUBDIRS` macro. An example of such a package is `termutils`.

A **deep** package is one in which all the source lies in subdirectories; the top level directory contains mainly configuration information. GNU `cpio` is a good example of such a package, as is GNU `tar`. The top level `` `Makefile.am' `` for a deep package will contain a `SUBDIRS` macro, but no other macros to define objects which are built.

A **shallow** package is one in which the primary source resides in the top-level directory, while various parts (typically libraries) reside in subdirectories. Automake is one such package (as is GNU `make`, which does not currently use `automake`).

Strictness

While Automake is intended to be used by maintainers of GNU packages, it does make some effort to accommodate those who wish to use it, but do not want to use all the GNU conventions.

To this end, Automake supports three levels of **strictness**---the strictness indicating how stringently Automake should check standards conformance.

The valid strictness levels are:

`foreign'

> Automake will check for only those things which are absolutely required for proper operations. For instance, whereas GNU standards dictate the existence of a `` `NEWS' `` file, it will not be required in this mode. The name comes from the fact that Automake is intended to be used for GNU programs; these relaxed rules are not the standard mode of operation.

`gnu'

> Automake will check--as much as possible--for compliance to the GNU standards for packages. This is the default.

`gnits'

> Automake will check for compliance to the as-yet-unwritten **Gnits standards**. These are based on the GNU standards, but are even more detailed. Unless you are a Gnits standards contributor, it is recommended that you avoid this option until such time as the Gnits standard is actually published.

For more information on the precise implications of the strictness level, see section The effect of `--gnu` and `--gnits`.

The Uniform Naming Scheme

Automake macros (from here on referred to as *variables*) generally follow a **uniform naming scheme** that makes it easy to decide how programs (and other derived objects) are built, and how they are installed. This scheme also supports `configure` time determination of what should be built.

At `make` time, certain variables are used to determine which objects are to be built. These variables are called **primary variables**. For instance, the primary variable `PROGRAMS` holds a list of programs which are to be compiled and linked.

A different set of variables is used to decide where the built objects should be installed. These variables are named after the primary variables, but have a prefix indicating which standard directory should be used as the installation directory. The standard directory names are given in the GNU standards (see section `Directory Variables' in The GNU Coding Standards). Automake extends this list with `pkglibdir`, `pkgincludedir`, and `pkgdatadir`; these are the same as the non-`pkg' versions, but with `@PACKAGE@' appended. For instance, `pkglibdir` is defined as `$(datadir)/@PACKAGE@`.

For each primary, there is one additional variable named by prepending `EXTRA_' to the primary name. This variable is used to list objects which may or may not be built, depending on what `configure` decides. This variable is required because Automake must statically know the entire list of objects that may be built in order to generate a `Makefile.in' that will work in all cases.

For instance, `cpio` decides at configure time which programs are built. Some of the programs are installed in `bindir`, and some are installed in `sbindir`:

```
EXTRA_PROGRAMS = mt rmt
bin_PROGRAMS = cpio pax
sbin_PROGRAMS = @PROGRAMS@
```

Defining a primary variable without a prefix (e.g. `PROGRAMS`) is an error.

Note that the common `dir' suffix is left off when constructing the variable names; thus one writes `bin_PROGRAMS' and not `bindir_PROGRAMS'.

Not every sort of object can be installed in every directory. Automake will flag those attempts it finds in error. Automake will also diagnose obvious misspellings in directory names.

Sometimes the standard directories--even as augmented by Automake--- are not enough. In particular it is sometimes useful, for clarity, to install objects in a subdirectory of some predefined directory. To this end, Automake allows you to extend the list of possible installation directories. A given prefix (e.g. `zar') is valid if a variable of the same name with `dir' appended is defined (e.g. `zardir`).

For instance, until HTML support is part of Automake, you could use this to install raw HTML documentation:

```
htmldir = $(prefix)/html
html_DATA = automake.html
```

The special prefix `noinst' indicates that the objects in question should not be installed at all.

The special prefix `check' indicates that the objects in question should not be built until the `make check` command is run.

Possible primary names are `PROGRAMS', `LIBRARIES', `LISP', `SCRIPTS', `DATA', `HEADERS', `MANS', and `TEXINFOS'.

How derived variables are named

Sometimes a Makefile variable name is derived from some text the user supplies. For instance, program names are rewritten into Makefile macro names. Automake canonicalizes this text, so that it does not have to follow Makefile macro naming rules. All characters in the name except for letters, numbers, and the underscore are turned into underscores when making macro references. For example, if your program is named `sniff-glue`, the derived variable name would be `sniff_glue_SOURCES`, not `sniff-glue_SOURCES`.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Some example packages

A simple example, start to finish

Let's suppose you just finished writing `zardoz`, a program to make your head float from vortex to vortex. You've been using Autoconf to provide a portability framework, but your `` `Makefile.in' ``s have been ad-hoc. You want to make them bulletproof, so you turn to Automake.

The first step is to update your `` `configure.in' `` to include the commands that `automake` needs. The simplest way to do this is to add an `AM_INIT_AUTOMAKE` call just after `AC_INIT`:

```
AM_INIT_AUTOMAKE(zardoz, 1.0)
```

Since your program doesn't have any complicating factors (e.g., it doesn't use `gettext`, it doesn't want to build a shared library), you're done with this part. That was easy!

Now you must regenerate `` `configure' ``. But to do that, you'll need to tell `autoconf` how to find the new macro you've used. The easiest way to do this is to use the `aclocal` program to generate your `` `aclocal.m4' `` for you. But wait... you already have an `` `aclocal.m4' ``, because you had to write some hairy macros for your program. The `aclocal` program lets you put your own macros into `` `acinclude.m4' ``, so simply rename and then run:

```
mv aclocal.m4 acinclude.m4
aclocal
autoconf
```

Now it is time to write your `` `Makefile.am' `` for `zardoz`. Since `zardoz` is a user program, you want to install it where the rest of the user programs go. Additionally, `zardoz` has some Texinfo documentation. Your `` `configure.in' `` script uses `AC_REPLACE_FUNCS`, so you need to link against `@LIBOBJS@'. So here's what you'd write:

```
bin_PROGRAMS = zardoz
zardoz_SOURCES = main.c head.c float.c vortex9.c gun.c
zardoz_LDADD = @LIBOBJS@

info_TEXINFOS = zardoz.texi
```

Now you can run `automake --add-missing` to generate your `` `Makefile.in' `` and grab any auxiliary files you might need, and you're done!

A classic program

GNU hello is renowned for its classic simplicity and versatility. This section shows how Automake could be used with the GNU Hello package. The examples below are from the latest beta version of GNU Hello, but with all of the maintainer-only code stripped out, as well as all copyright comments.

Of course, GNU Hello is somewhat more featureful than your traditional two-liner. GNU Hello is internationalized, does option processing, and has a manual and a test suite. GNU Hello is a deep package.

Here is the `configure.in' from GNU Hello:

```
dnl Process this file with autoconf to produce a configure script.
AC_INIT(src/hello.c)
AM_INIT_AUTOMAKE(hello, 1.3.11)
AM_CONFIG_HEADER(config.h)

dnl Set of available languages.
ALL_LINGUAS="de fr es ko nl no pl pt sl sv"

dnl Checks for programs.
AC_PROG_CC
AC_ISC_POSIX

dnl Checks for libraries.

dnl Checks for header files.
AC_STDC_HEADERS
AC_HAVE_HEADERS(string.h fcntl.h sys/file.h sys/param.h)

dnl Checks for library functions.
AC_FUNC_ALLOCA

dnl Check for st_blksize in struct stat
AC_ST_BLKSIZE

dnl internationalization macros
AM_GNU_GETTEXT
AC_OUTPUT([Makefile doc/Makefile intl/Makefile po/Makefile.in \
           src/Makefile tests/Makefile tests/hello],
   [chmod +x tests/hello])
```

The `AM_' macros are provided by Automake (or the Gettext library); the rest are standard Autoconf macros.

The top-level `Makefile.am':


```
EXTRA_DIST = BUGS ChangeLog.O
SUBDIRS = doc intl po src tests
```

As you can see, all the work here is really done in subdirectories.

The `po' and `intl' directories are automatically generated using `gettextize'; they will not be discussed here.

In `doc/Makefile.am' we see:

```
info_TEXINFOS = hello.texi
hello_TEXINFOS = gpl.texi
```

This is sufficient to build, install, and distribute the GNU Hello manual.

Here is `tests/Makefile.am':

```
TESTS = hello
EXTRA_DIST = hello.in testdata
```

The script `hello' is generated by `configure', and is the only test case. `make check' will run this test.

Last we have `src/Makefile.am', where all the real work is done:

```
bin_PROGRAMS = hello
hello_SOURCES = hello.c version.c getopt.c getopt1.c getopt.h system.h
hello_LDADD = @INTLLIBS@ @ALLOCA@
localedir = $(datadir)/locale
INCLUDES = -I../intl -DLOCALEDIR=\"$(localedir)\"
```

Building etags and ctags

Here is another, trickier example. It shows how to generate two programs (`ctags' and `etags') from the same source file (`etags.c'). The difficult part is that each compilation of `etags.c' requires different `cpp' flags.

```
bin_PROGRAMS = etags ctags
ctags_SOURCES =
ctags_LDADD = ctags.o

etags.o: etags.c
        $(COMPILE) -DETAGS_REGEXPS -c etags.c

ctags.o: etags.c
        $(COMPILE) -DCTAGS -o ctags.o -c etags.c
```

Note that `ctags_SOURCES` is defined to be empty--that way no implicit value is substituted. The implicit value, however, is used to generate `etags` from `` `etags.o' ``.

`ctags_LDADD` is used to get `` `ctags.o' `` into the link line. `ctags_DEPENDENCIES` is generated by Automake.

The above rules won't work if your compiler doesn't accept both `-c' and `-o'. The simplest fix for this is to introduce a bogus dependency (to avoid problems with a parallel `make`):

```
etags.o: etags.c ctags.o
        $(COMPILE) -DETAGS_REGEXPS -c etags.c

ctags.o: etags.c
        $(COMPILE) -DCTAGS -c etags.c && mv etags.o ctags.o
```

Also, these explicit rules do not work if the de-ANSI-fication feature is used (see section Automatic de-ANSI-fication). Supporting de-ANSI-fication requires a little more work:

```
etags._o: etags._c ctags.o
        $(COMPILE) -DETAGS_REGEXPS -c etags.c

ctags._o: etags._c
        $(COMPILE) -DCTAGS -c etags.c && mv etags._o ctags.o
```

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Creating a `Makefile.in'

To create all the `Makefile.in's for a package, run the `automake` program in the top level directory, with no arguments. `automake` will automatically find each appropriate `Makefile.am' (by scanning `configure.in'; see section Scanning `configure.in') and generate the corresponding `Makefile.in'. Note that `automake` has a rather simplistic view of what constitutes a package; it assumes that a package has only one `configure.in', at the top. If your package has multiple `configure.in's, then you must run `automake` in each directory holding a `configure.in'.

You can optionally give `automake` an argument; `.am' is appended to the argument and the result is used as the name of the input file. This feature is generally only used to automatically rebuild an out-of-date `Makefile.in'. Note that `automake` must always be run from the topmost directory of a project, even if being used to regenerate the `Makefile.in' in some subdirectory. This is necessary because `automake` must scan `configure.in', and because `automake` uses the knowledge that a `Makefile.in' is in a subdirectory to change its behavior in some cases.

`automake` accepts the following options:

`-a'

`--add-missing'

> Automake requires certain common files to exist in certain situations; for instance `config.guess' is required if `configure.in' runs `AC_CANONICAL_HOST`. Automake is distributed with several of these files; this option will cause the missing ones to be automatically added to the package, whenever possible. In general if Automake tells you a file is missing, try using this option. By default Automake tries to make a symbolic link pointing to its own copy of the missing file; this can be changed with `--copy`.

`--amdir=dir'

> Look for Automake data files in directory dir instead of in the installation directory. This is typically used for debugging.

`--build-dir=dir'

> Tell Automake where the build directory is. This option is used when including dependencies into a `Makefile.in' generated by `make dist`; it should not be used otherwise.

`-c'

`--copy'

> When used with `--add-missing`, causes installed files to be copied. The default is to make a symbolic link.

`--cygnus'

> Causes the generated `Makefile.in's to follow Cygnus rules, instead of GNU or Gnits rules.

For more information, see section The effect of `--cygnus`.

`--foreign'

Set the global strictness to `foreign'. For more information, see section Strictness.

`--gnits'

Set the global strictness to `gnits'. For more information, see section The effect of `--gnu` and `--gnits`.

`--gnu'

Set the global strictness to `gnu'. For more information, see section The effect of `--gnu` and `--gnits`. This is the default strictness.

`--help'

Print a summary of the command line options and exit.

`-i'

`--include-deps'

Include all automatically generated dependency information (see section Automatic dependency tracking) in the generated `Makefile.in'. This is generally done when making a distribution; see section What Goes in a Distribution.

`--generate-deps'

Generate a file concatenating all automatically generated dependency information (see section Automatic dependency tracking) into one file, `.dep_segment'. This is generally done when making a distribution; see section What Goes in a Distribution. It is useful when maintaining a `SMakefile' or makefiles for other platforms (`Makefile.DOS', etc.) It can only be used in conjunction with `--include-deps', `--srcdir-name', and `--build-dir'. Note that if this option is given, no other processing is done.

`--no-force'

Ordinarily `automake` creates all `Makefile.in's mentioned in `configure.in'. This option causes it to only update those `Makefile.in's which are out of date with respect to one of their dependents.

`-o dir'

`--output-dir=dir'

Put the generated `Makefile.in' in the directory dir. Ordinarily each `Makefile.in' is created in the directory of the corresponding `Makefile.am'. This option is used when making distributions.

`--srcdir-name=dir'

Tell Automake the name of the source directory associated with the current build. This option is used when including dependencies into a `Makefile.in' generated by `make dist`; it should not be used otherwise.

`-v'

`--verbose'

Cause Automake to print information about which files are being read or created.

`--version'

Print the version number of Automake and exit.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Scanning `configure.in'

Automake scans the package's `configure.in' to determine certain information about the package. Some autoconf macros are required and some variables must be defined in `configure.in'. Automake will also use information from `configure.in' to further tailor its output.

Automake also supplies some Autoconf macros to make the maintenance easier. These macros can automatically be put into your `aclocal.m4' using the aclocal program.

Configuration requirements

The simplest way to meet the basic Automake requirements is to use the macro AM_INIT_AUTOMAKE (see section Autoconf macros supplied with Automake). But if you prefer, you can do the required steps by hand:

- Define the variables PACKAGE and VERSION with AC_SUBST. PACKAGE should be the name of the package as it appears when bundled for distribution. For instance, Automake defines PACKAGE to be `automake'. VERSION should be the version number of the release that is being developed. We recommend that you make `configure.in' the only place in your package where the version number is defined; this makes releases simpler. Automake doesn't do any interpretation of PACKAGE or VERSION, except in `Gnits' mode (see section The effect of --gnu and --gnits).
- Use the macro AC_ARG_PROGRAM if a program or script is installed. See section `Transforming Program Names When Installing' in The Autoconf.
- Use AC_PROG_MAKE_SET if the package is not flat. See section `Creating Output Files' in The Autoconf Manual.
- Use AM_SANITY_CHECK to make sure the build environment is sane.
- Call AC_PROG_INSTALL (see section `Particular Program Checks' in The Autoconf Manual).
- Use AM_MISSING_PROG to see whether the programs aclocal, autoconf, automake, autoheader, and makeinfo are in the build environment. Here is how this is done:

```
missing_dir=`cd $ac_aux_dir && pwd`
AM_MISSING_PROG(ACLOCAL, aclocal, $missing_dir)
AM_MISSING_PROG(AUTOCONF, autoconf, $missing_dir)
AM_MISSING_PROG(AUTOMAKE, automake, $missing_dir)
AM_MISSING_PROG(AUTOHEADER, autoheader, $missing_dir)
AM_MISSING_PROG(MAKEINFO, makeinfo, $missing_dir)
```

Here are the other macros which Automake requires but which are not run by AM_INIT_AUTOMAKE:

AC_OUTPUT

Automake uses this to determine which files to create (see section `Creating Output Files' in The Autoconf Manual). Listed files named `Makefile` are treated as `Makefile'`s. Other listed files are treated differently. Currently the only difference is that a `Makefile'` is removed by `make distclean`, while other files are removed by `make clean`.

Other things Automake recognizes

Automake will also recognize the use of certain macros and tailor the generated `Makefile.in'` appropriately. Currently recognized macros and their effects are:

AC_CONFIG_HEADER

Automake requires the use of AM_CONFIG_HEADER, which is similar to AC_CONFIG_HEADER (see section `Configuration Header Files' in The Autoconf Manual), but does some useful Automake-specific work.

AC_CONFIG_AUX_DIR

Automake will look for various helper scripts, such as `mkinstalldirs'`, in the directory named in this macro invocation. If not seen, the scripts are looked for in their `standard' locations (either the top source directory, or in the source directory corresponding to the current `Makefile.am'`, whichever is appropriate). See section `Finding `configure' Input' in The Autoconf Manual. FIXME: give complete list of things looked for in this directory

AC_PATH_XTRA

Automake will insert definitions for the variables defined by AC_PATH_XTRA into each `Makefile.in'` that builds a C program or library. See section `System Services' in The Autoconf Manual.

AC_CANONICAL_HOST

AC_CHECK_TOOL

Automake will ensure that `config.guess'` and `config.sub'` exist. Also, the `Makefile'` variables `host_alias' and `host_triplet' are introduced. See both section `Getting the Canonical System Type' in The Autoconf Manual, and section `Generic Program Checks' in The Autoconf Manual.

AC_CANONICAL_SYSTEM

This is similar to AC_CANONICAL_HOST, but also defines the `Makefile'` variables `build_alias' and `target_alias'. See section `Getting the Canonical System Type' in The Autoconf Manual.

AC_FUNC_ALLOCA

AC_FUNC_GETLOADAVG

AC_FUNC_MEMCMP

AC_STRUCT_ST_BLOCKS

AC_FUNC_FNMATCH

AM_FUNC_STRTOD

`AC_REPLACE_FUNCS`

`AC_REPLACE_GNU_GETOPT`

`AM_WITH_REGEX`

Automake will ensure that the appropriate dependencies are generated for the objects corresponding to these macros. Also, Automake will verify that the appropriate source files are part of the distribution. Note that Automake does not come with any of the C sources required to use these macros, so `automake -a` will not install the sources. See section Building a library, for more information. Also, see section `Particular Function Checks' in The Autoconf Manual.

`LIBOBJS`

Automake will detect statements which put `.o' files into `LIBOBJS`, and will treat these additional files as if they were discovered via `AC_REPLACE_FUNCS`. See section `Generic Function Checks' in The Autoconf Manual.

`AC_PROG_RANLIB`

This is required if any libraries are built in the package. See section `Particular Program Checks' in The Autoconf Manual.

`AC_PROG_CXX`

This is required if any C++ source is included. See section `Particular Program Checks' in The Autoconf Manual.

`AC_PROG_F77`

This is required if any Fortran 77 source is included. This macro is distributed with Autoconf version 2.13 and later. See section `Particular Program Checks' in The Autoconf Manual.

`AC_F77_LIBRARY_LDFLAGS`

This is required for programs and shared libraries that are a mixture of languages that include Fortran 77 (see section Mixing Fortran 77 With C and C++). See section Autoconf macros supplied with Automake.

`AM_PROG_LIBTOOL`

Automake will turn on processing for `libtool` (see section `Introduction' in The Libtool Manual).

`AC_PROG_YACC`

If a Yacc source file is seen, then you must either use this macro or define the variable `YACC' in `configure.in'. The former is preferred (see section `Particular Program Checks' in The Autoconf Manual).

`AC_DECL_YYTEXT`

This macro is required if there is Lex source in the package. See section `Particular Program Checks' in The Autoconf Manual.

`AC_PROG_LEX`

If a Lex source file is seen, then this macro must be used. See section `Particular Program Checks' in The Autoconf Manual.

`ALL_LINGUAS`

If Automake sees that this variable is set in `configure.in', it will check the `po' direc to ensure that all the named `.po' files exist, and that all the `.po' files that exist are named.

AM_C_PROTOTYPES

This is required when using automatic de-ANSI-fication; see section Automatic de-ANSI-fication.

AM_GNU_GETTEXT

This macro is required for packages which use GNU gettext (see section Gettext). It is distributed with gettext. If Automake sees this macro it ensures that the package meets some of gettext's requirements.

AM_MAINTAINER_MODE

This macro adds a `--enable-maintainer-mode' option to `configure'. If this is used, `automake' will cause `maintainer-only' rules to be turned off by default in the generated `Makefile.in's. This macro is disallowed in `Gnits' mode (see section The effect of `--gnu' and `--gnits'). This macro defines the `MAINTAINER_MODE' conditional, which you can use in your own `Makefile.am'.

AC_SUBST

AC_CHECK_TOOL

AC_CHECK_PROG

AC_CHECK_PROGS

AC_PATH_PROG

AC_PATH_PROGS

For each of these macros, the first argument is automatically defined as a variable in each generated `Makefile.in'. See section `Setting Output Variables' in The Autoconf Manual, and section `Generic Program Checks' in The Autoconf Manual.

Auto-generating aclocal.m4

Automake includes a number of Autoconf macros which can be used in your package; some of them are actually required by Automake in certain situations. These macros must be defined in your `aclocal.m4'; otherwise they will not be seen by `autoconf'.

The `aclocal' program will automatically generate `aclocal.m4' files based on the contents of `configure.in'. This provides a convenient way to get Automake-provided macros, without having to search around. Also, the `aclocal' mechanism is extensible for use by other packages.

At startup, `aclocal' scans all the `.m4' files it can find, looking for macro definitions. Then it scans `configure.in'. Any mention of one of the macros found in the first step causes that macro, and any macros it in turn requires, to be put into `aclocal.m4'.

The contents of `acinclude.m4', if it exists, are also automatically included in `aclocal.m4'. This is useful for incorporating local macros into `configure'.

`aclocal' accepts the following options:

`--acdir=dir`

Look for the macro files in dir instead of the installation directory. This is typically used for debugging.

`--help`

Print a summary of the command line options and exit.

`-I dir`

Add the directory dir to the list of directories searched for `` `.m4' `` files.

`--output=file`

Cause the output to be put into file instead of `` `aclocal.m4' ``.

`--print-ac-dir`

Prints the name of the directory which `aclocal` will search to find the `` `.m4' `` files. When this option is given, normal processing is suppressed. This option can be used by a package to determine where to install a macro file.

`--verbose`

Print the names of the files it examines.

`--version`

Print the version number of Automake and exit.

Autoconf macros supplied with Automake

`AM_CONFIG_HEADER`

Automake will generate rules to automatically regenerate the config header. If you do use this macro, you must create the file `` `stamp-h.in' `` in your source directory. It can be empty.

`AM_ENABLE_MULTILIB`

This is used when a "multilib" library is being built. A **multilib** library is one that is built multiple times, once per target flag combination. This is only useful when the library is intended to be cross-compiled. The first optional argument is the name of the `` `Makefile' `` being generated; it defaults to `Makefile'. The second option argument is used to find the top source directory; it defaults to the empty string (generally this should not be used unless you are familiar with the internals).

`AM_FUNC_STRTOD`

If the `strtod` function is not available, or does not work correctly (like the one on SunOS 5.4), add `` `strtod.o' `` to output variable `LIBOBJS`.

`AM_FUNC_ERROR_AT_LINE`

If the function `error_at_line` is not found, then add `` `error.o' `` to `LIBOBJS`.

`AM_FUNC_MKTIME`

Check for a working `mktime` function. If not found, add `` `mktime.o' `` to `LIBOBJS'.

`AM_FUNC_OBSTACK`

Check for the GNU obstacks code; if not found, add `` `obstack.o' `` to `LIBOBJS'.

`AM_C_PROTOTYPES`

Check to see if function prototypes are understood by the compiler. If so, define `PROTOTYPES' and set the output variables `U' and `ANSI2KNR' to the empty string. Otherwise, set `U' to `_' and `ANSI2KNR' to `./ansi2knr'. Automake uses these values to implement automatic de-ANSI-fication.

`AM_HEADER_TIOCGWINSZ_NEEDS_SYS_IOCTL`

If the use of `TIOCGWINSZ` requires `<sys/ioctl.h>', then define `GWINSZ_IN_SYS_IOCTL`. Otherwise `TIOCGWINSZ` can be found in `<termios.h>'.

`AM_INIT_AUTOMAKE`

Runs many macros that most `configure.in`'s need. This macro has two required arguments, the package and the version number. By default this macro `AC_DEFINE`'s `PACKAGE' and `VERSION'. This can be avoided by passing in a non-empty third argument.

`AM_PATH_LISPDIR`

Searches for the program `emacs`, and, if found, sets the output variable `lispdir` to the full path to Emacs' site-lisp directory.

`AM_PROG_CC_STDC`

If the C compiler in not in ANSI C mode by default, try to add an option to output variable `CC` to make it so. This macro tries various options that select ANSI C on some system or another. It considers the compiler to be in ANSI C mode if it handles function prototypes correctly. If you use this macro, you should check after calling it whether the C compiler has been set to accept ANSI C; if not, the shell variable `am_cv_prog_cc_stdc` is set to `no'. If you wrote your source code in ANSI C, you can make an un-ANSIfied copy of it by using the `ansi2knr` option (see section Automatic de-ANSI-fication).

`AM_PROG_LEX`

Like `AC_PROG_LEX` with `AC_DECL_YYTEXT` (see section `Particular Program Checks' in The Autoconf Manual), but uses the `missing` script on systems that do not have `lex`. `HP-UX 10' is one such system.

`AM_SANITY_CHECK`

This checks to make sure that a file created in the build directory is newer than a file in the source directory. This can fail on systems where the clock is set incorrectly. This macro is automatically run from `AM_INIT_AUTOMAKE`.

`AM_SYS_POSIX_TERMIOS`

Check to see if POSIX termios headers and functions are available on the system. If so, set the shell variable `am_cv_sys_posix_termios` to `yes'. If not, set the variable to `no'.

`AM_TYPE_PTRDIFF_T`

Define `HAVE_PTRDIFF_T' if the type `ptrdiff_t' is defined in `<stddef.h>'.

`AM_WITH_DMALLOC`

Add support for the `dmalloc` package. If the user configures with `--with-dmalloc', then define `WITH_DMALLOC` and add `-ldmalloc' to `LIBS`.

`AM_WITH_REGEX`

Adds `--with-regex' to the `configure` command line. If specified (the default), then the `reg
regular expression library is used, `regex.o' is put into `LIBOBJS', and `WITH_REGEX' i
defined.. If `--without-regex' is given, then the `rx' regular expression library is used, and `rx.o'
is put into `LIBOBJS'.

Writing your own aclocal macros

The `aclocal` program doesn't have any built-in knowledge of any macros, so it is easy to extend it with your own macros.

This is mostly used for libraries which want to supply their own Autoconf macros for use by other programs. For instance the `gettext` library supplies a macro `AM_GNU_GETTEXT` which should be used by any package using `gettext`. When the library is installed, it installs this macro so that `aclocal` will find it.

A file of macros should be a series of `AC_DEFUN`'s. The `aclocal` programs also understands `AC_REQUIRE`, so it is safe to put each macro in a separate file. See section `Prerequisite Macros' in The Autoconf Manual, and section `Macro Definitions' in The Autoconf Manual.

A macro file's name should end in `.m4'. Such files should be installed in `$(datadir)/aclocal'.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

The top-level `Makefile.am'

In non-flat packages, the top level `Makefile.am' must tell Automake which subdirectories are to be built. This is done via the SUBDIRS variable.

The SUBDIRS macro holds a list of subdirectories in which building of various sorts can occur. Many targets (e.g. all) in the generated `Makefile' will run both locally and in all specified subdirectories. Note that the directories listed in SUBDIRS are not required to contain `Makefile.am's; only `Makefile's (after configuration). This allows inclusion of libraries from packages which do not use Automake (such as gettext). The directories mentioned in SUBDIRS must be direct children of the current directory. For instance, you cannot put `src/subdir' into SUBDIRS.

In a deep package, the top-level `Makefile.am' is often very short. For instance, here is the `Makefile.am' from the GNU Hello distribution:

```
EXTRA_DIST = BUGS ChangeLog.O README-alpha
SUBDIRS = doc intl po src tests
```

It is possible to override the SUBDIRS variable if, like in the case of GNU Inetutils, you want to only build a subset of the entire package. In your `Makefile.am' include:

```
SUBDIRS = @SUBDIRS@
```

Then in your `configure.in' you can specify:

```
SUBDIRS = "src doc lib po"
AC_SUBST(SUBDIRS)
```

The upshot of this is that Automake is tricked into building the package to take the subdirs, but doesn't actually bind that list until configure is run.

Although the SUBDIRS macro can contain configure substitutions (e.g. `@DIRS@'); Automake itself does not actually examine the contents of this variable.

If SUBDIRS is defined, then your `configure.in' must include AC_PROG_MAKE_SET.

The use of SUBDIRS is not restricted to just the top-level `Makefile.am'. Automake can be used to construct packages of arbitrary depth.

By default, Automake generates `Makefiles' which work depth-first (`postfix'). However, it is possible to change this ordering. You can do this by putting `.' into SUBDIRS. For instance, putting `.' first will cause a `prefix' ordering of directories.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Building Programs and Libraries

A large part of Automake's functionality is dedicated to making it easy to build programs and libraries.

Building a program

In a directory containing source that gets built into a program (as opposed to a library), the `PROGRAMS' primary is used. Programs can be installed in `bindir`, `sbindir`, `libexecdir`, `pkglibdir`, or not at all (`noinst').

For instance:

```
bin_PROGRAMS = hello
```

In this simple case, the resulting `Makefile.in' will contain code to generate a program named `hello`. The variable `hello_SOURCES` is used to specify which source files get built into an executable:

```
hello_SOURCES = hello.c version.c getopt.c getopt1.c getopt.h system.h
```

This causes each mentioned `.c' file to be compiled into the corresponding `.o'. Then all are linked to produce `hello'.

If `prog_SOURCES' is needed, but not specified, then it defaults to the single file `prog.c'.

Multiple programs can be built in a single directory. Multiple programs can share a single source file, which must be listed in each `_SOURCES' definition.

Header files listed in a `_SOURCES' definition will be included in the distribution but otherwise ignored. In case it isn't obvious, you should not include the header file generated by `configure' in an `_SOURCES' variable; this file should not be distributed. Lex (`.l') and Yacc (`.y') files can also be listed; see section Yacc and Lex support.

Automake must know all the source files that could possibly go into a program, even if not all the files are built in every circumstance. Any files which are only conditionally built should be listed in the appropriate `EXTRA_' variable. For instance, if `hello-linux.c' were conditionally included in `hello`, the `Makefile.am' would contain:

```
EXTRA_hello_SOURCES = hello-linux.c
```

Similarly, sometimes it is useful to determine the programs that are to be built at configure time. For instance, GNU `cpio` only builds `mt` and `rmt` under special circumstances.

In this case, you must notify Automake of all the programs that can possibly be built, but at the same cause the generated `Makefile.in' to use the programs specified by `configure'. This is done by having `configure' substitute values into each `_PROGRAMS' definition, while listing all optionally built programs in `EXTRA_PROGRAMS'.

If you need to link against libraries that are not found by `configure', you can use `LDADD' to do so. This variable actually can be used to add any options to the linker command line.

Sometimes, multiple programs are built in one directory but do not share the same link-time requirements. In this case, you can use the `prog_LDADD' variable (where prog is the name of the program as it appears in some `_PROGRAMS' variable, and usually written in lowercase) to override the global `LDADD'. If this variable exists for a given program, then that program is not linked using `LDADD'.

For instance, in GNU cpio, `pax', `cpio' and `mt' are linked against the library `libcpio.a'. However, `rmt' is built in the same directory, and has no such link requirement. Also, `mt' and `rmt' are only built on certain architectures. Here is what cpio's `src/Makefile.am' looks like (abridged):

```
bin_PROGRAMS = cpio pax @MT@
libexec_PROGRAMS = @RMT@
EXTRA_PROGRAMS = mt rmt

LDADD = ../lib/libcpio.a @INTLLIBS@
rmt_LDADD =

cpio_SOURCES = ...
pax_SOURCES = ...
mt_SOURCES = ...
rmt_SOURCES = ...
```

`prog_LDADD' is inappropriate for passing program-specific linker flags (except for `-l' and `-L'). So, use the `prog_LDFLAGS' variable for this purpose.

It is also occasionally useful to have a program depend on some other target which is not actually part of that program. This can be done using the `prog_DEPENDENCIES' variable. Each program depends on the contents of such a variable, but no further interpretation is done.

If `prog_DEPENDENCIES' is not supplied, it is computed by Automake. The automatically-assigned value is the contents of `prog_LDADD', with most configure substitutions, `-l', and `-L' options removed. The configure substitutions that are left in are only `@LIBOBJS@' and `@ALLOCA@'; these are left because it is known that they will not cause an invalid value for `prog_DEPENDENCIES' to be generated.

Building a library

Building a library is much like building a program. In this case, the name of the primary is `LIBRARIES'. Libraries can be installed in `libdir' or `pkglibdir'.

See section Building a Shared Library, for information on how to build shared libraries using Libtool and

the `LTLIBRARIES' primary.

Each `_LIBRARIES' variable is a list of the libraries to be built. For instance to create a library named `libcpio.a', but not install it, you would write:

```
noinst_LIBRARIES = libcpio.a
```

The sources that go into a library are determined exactly as they are for programs, via the `_SOURCES' variables. Note that the library name is canonicalized (see section How derived variables are named), so the `_SOURCES' variable corresponding to `liblob.a' is `liblob_a_SOURCES', not `liblob.a_SOURCES'.

Extra objects can be added to a library using the `library_LIBADD' variable. This should be used for objects determined by `configure'. Again from `cpio':

```
libcpio_a_LIBADD = @LIBOBJS@ @ALLOCA@
```

Special handling for LIBOBJS and ALLOCA

Automake explicitly recognizes the use of `@LIBOBJS@' and `@ALLOCA@', and uses this information, plus the list of `LIBOBJS' files derived from `configure.in' to automatically include the appropriate source files in the distribution (see section What Goes in a Distribution). These source files are also automatically handled in the dependency-tracking scheme; see See section Automatic dependency tracking.

`@LIBOBJS@' and `@ALLOCA@' are specially recognized in any `_LDADD' or `_LIBADD' variable.

Building a Shared Library

Building shared libraries is a relatively complex matter. For this reason, GNU Libtool (see section `Introduction' in The Libtool Manual) was created to help build shared libraries in a platform-independent way.

Automake uses Libtool to build libraries declared with the `LTLIBRARIES' primary. Each `_LTLIBRARIES' variable is a list of shared libraries to build. For instance, to create a library named `libgettext.a' and its corresponding shared libraries, and install them in `libdir', write:

```
lib_LTLIBRARIES = libgettext.la
```

Note that shared libraries *must* be installed, so `check_LTLIBRARIES' is not allowed. However, `noinst_LTLIBRARIES' is allowed. This feature should be used for libtool "convenience libraries".

For each library, the `library_LIBADD' variable contains the names of extra libtool objects (`.lo' files) to add to the shared library. The `library_LDFLAGS' variable contains any additional libtool flags, such as `-version-info' or `-static'.

Where an ordinary library might include `@LIBOBJS@`, a libtool library must use `@LTLIBOBJS@`. Th required because the object files that libtool operates on do not necessarily end in `.o'`. The libtool manual contains more details on this topic.

For libraries installed in some directory, Automake will automatically supply the appropriate `-rpath' option. However, for libraries determined at configure time (and thus mentioned in `EXTRA_LTLIBRARIES`), Automake does not know the eventual installation directory; for such libraries you must add the `-rpath' option to the appropriate `_LDFLAGS' variable by hand.

See section `The Libtool Manual' in The Libtool Manual, for more information.

Variables used when building a program

Occasionally it is useful to know which `Makefile'` variables Automake uses for compilations; for instance you might need to do your own compilation in some special cases.

Some variables are inherited from Autoconf; these are `CC`, `CFLAGS`, `CPPFLAGS`, `DEFS`, `LDFLAGS`, and `LIBS`.

There are some additional variables which Automake itself defines:

`INCLUDES`

A list of `-I' options. This can be set in your `Makefile.am'` if you have special directories you want to look in. Automake already provides some `-I' options automatically. In particular it generates `-I$(srcdir)' and a `-I' pointing to the directory holding `config.h'` (if you've used `AC_CONFIG_HEADER` or `AM_CONFIG_HEADER`). `INCLUDES` can actually be used for other `cpp` options besides `-I'. For instance, it is sometimes used to pass arbitrary `-D' options to the compiler.

`COMPILE`

This is the command used to actually compile a C source file. The filename is appended to form the complete command line.

`LINK`

This is the command used to actually link a C program.

Yacc and Lex support

Automake has somewhat idiosyncratic support for Yacc and Lex.

Automake assumes that the `.c'` file generated by `yacc` (or `lex`) should be named using the basename of the input file. That is, for a yacc source file `foo.y'`, Automake will cause the intermediate file to be named `foo.c'` (as opposed to `y.tab.c'`, which is more traditional).

The extension of a yacc source file is used to determine the extension of the resulting `C' or `C++' file. Files with the extension `.y' will be turned into `.c' files; likewise, `.yy' will become `.cc'; `.y++', `c++'; and `.yxx', `.cxx'.

Likewise, lex source files can be used to generate `C' or `C++'; the extensions `.l', `.ll', `.l++', and `.lx` recognized.

You should never explicitly mention the intermediate (`C' or `C++') file in any `SOURCES' variable; only list the source file.

The intermediate files generated by `yacc` (or `lex`) will be included in any distribution that is made. That way the user doesn't need to have `yacc` or `lex`.

If a `yacc` source file is seen, then your `configure.in' must define the variable `YACC'. This is most easily done by invoking the macro `AC_PROG_YACC' (see section `Particular Program Checks' in The Autoconf Manual).

Similarly, if a `lex` source file is seen, then your `configure.in' must define the variable `LEX'. You can use `AC_PROG_LEX' to do this (see section `Particular Program Checks' in The Autoconf Manual). Automake's `lex` support also requires that you use the `AC_DECL_YYTEXT' macro--automake needs to know the value of `LEX_OUTPUT_ROOT'. This is all handled for you if you use the `AM_PROG_LEX` macro (see section Autoconf macros supplied with Automake).

Automake makes it possible to include multiple `yacc` (or `lex`) source files in a single program. Automake uses a small program called `ylwrap` to run `yacc` (or `lex`) in a subdirectory. This is necessary because yacc's output filename is fixed, and a parallel make could conceivably invoke more than one instance of `yacc` simultaneously. The `ylwrap` program is distributed with Automake. It should appear in the directory specified by `AC_CONFIG_AUX_DIR' (see section `Finding `configure' Input' in The Autoconf Manual), or the current directory if that macro is not used in `configure.in'.

For `yacc`, simply managing locking is insufficient. The output of `yacc` always uses the same symbol names internally, so it isn't possible to link two `yacc` parsers into the same executable.

We recommend using the following renaming hack used in `gdb`:

```
#define yymaxdepth c_maxdepth
#define yyparse c_parse
#define yylex   c_lex
#define yyerror c_error
#define yylval  c_lval
#define yychar  c_char
#define yydebug c_debug
#define yypact  c_pact
#define yyr1    c_r1
#define yyr2    c_r2
#define yydef   c_def
#define yychk   c_chk
#define yypgo   c_pgo
#define yyact   c_act
#define yyexca  c_exca
#define yyerrflag c_errflag
#define yynerrs c_nerrs
```


```
#define yyps    c_ps
#define yypv    c_pv
#define yys     c_s
#define yy_yys  c_yys
#define yystate c_state
#define yytmp   c_tmp
#define yyv     c_v
#define yy_yyv  c_yyv
#define yyval   c_val
#define yylloc  c_lloc
#define yyreds  c_reds
#define yytoks  c_toks
#define yylhs   c_yylhs
#define yylen   c_yylen
#define yydefred c_yydefred
#define yydgoto c_yydgoto
#define yysindex c_yysindex
#define yyrindex c_yyrindex
#define yygindex c_yygindex
#define yytable  c_yytable
#define yycheck  c_yycheck
#define yyname   c_yyname
#define yyrule   c_yyrule
```

For each define, replace the `c_' prefix with whatever you like. These defines work for `bison`, `byacc`, and traditional `yacc`s. If you find a parser generator that uses a symbol not covered here, please report the new name so it can be added to the list.

C++ Support

Automake includes full support for C++.

Any package including C++ code must define the output variable `CXX' in `configure.in'; the simplest way to do this is to use the `AC_PROG_CXX` macro (see section `Particular Program Checks' in The Autoconf Manual).

A few additional variables are defined when a C++ source file is seen:

`CXX`

The name of the C++ compiler.

`CXXFLAGS`

Any flags to pass to the C++ compiler.

`CXXCOMPILE`

The command used to actually compile a C++ source file. The file name is appended to form the complete command line.

`CXXLINK`

The command used to actually link a C++ program.

Fortran 77 Support

Automake includes full support for Fortran 77.

Any package including Fortran 77 code must define the output variable `F77' in `configure.in'; the simplest way to do this is to use the `AC_PROG_F77` macro (see section `Particular Program Checks' in The Autoconf Manual). See section Fortran 77 and Autoconf.

A few additional variables are defined when a Fortran 77 source file is seen:

`F77`

The name of the Fortran 77 compiler.

`FFLAGS`

Any flags to pass to the Fortran 77 compiler.

`RFLAGS`

Any flags to pass to the Ratfor compiler.

`F77COMPILE`

The command used to actually compile a Fortran 77 source file. The file name is appended to form the complete command line.

`FLINK`

The command used to actually link a pure Fortran 77 program or shared library.

Automake can handle preprocessing Fortran 77 and Ratfor source files in addition to compiling them(1). Automake also contains some support for creating programs and shared libraries that are a mixture of Fortran 77 and other languages (see section Mixing Fortran 77 With C and C++).

These issues are covered in the following sections.

Preprocessing Fortran 77

`N.f' is made automatically from `N.F' or `N.r'. This rule runs just the preprocessor to convert a preprocessable Fortran 77 or Ratfor source file into a strict Fortran 77 source file. The precise command used is as follows:

`.F'

```
$(F77) -F $(DEFS) $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS)
$(AM_FFLAGS) $(FFLAGS)
```

`.r'

```
$(F77) -F $(AM_FFLAGS) $(FFLAGS) $(AM_RFLAGS) $(RFLAGS)
```

Compiling Fortran 77 Files

`N.o' is made automatically from `N.f', `N.F' or `N.r' by running the Fortran 77 compiler. The precise command used is as follows:

```
`.f'
    $(F77) -c $(AM_FFLAGS) $(FFLAGS)
`.F'
    $(F77) -c $(DEFS) $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS)
    $(AM_FFLAGS) $(FFLAGS)
`.r'
    $(F77) -c $(AM_FFLAGS) $(FFLAGS) $(AM_RFLAGS) $(RFLAGS)
```

Mixing Fortran 77 With C and C++

Automake currently provides *limited* support for creating programs and shared libraries that are a mixture of Fortran 77 and C and/or C++. However, there are many other issues related to mixing Fortran 77 with other languages that are *not* (currently) handled by Automake, but that are handled by other packages(2).

Automake can help in two ways:

1. Automatic selection of the linker depending on which combinations of source code.
2. Automatic selection of the appropriate linker flags (e.g. `-L' and `-l') to pass to the automatically selected linker in order to link in the appropriate Fortran 77 intrinsic and run-time libraries. These extra Fortran 77 linker flags are supplied in the output variable FLIBS by the AC_F77_LIBRARY_LDFLAGS Autoconf macro supplied with newer versions of Autoconf (Autoconf version 2.13 and later). See section `Fortran 77 Compiler Characteristics' in The Autoconf.

If Automake detects that a program or shared library (as mentioned in some _PROGRAMS or _LTLIBRARIES primary) contains source code that is a mixture of Fortran 77 and C and/or C++, then it requires that the macro AC_F77_LIBRARY_LDFLAGS be called in `configure.in', and that either $(FLIBS) or @FLIBS@ appear in the appropriate _LDADD (for programs) or _LIBADD (for shared libraries) variables. It is the responsibility of the person writing the `Makefile.am' to make sure that $(FLIBS) or @FLIBS@ appears in the appropriate _LDADD or _LIBADD variable.

For example, consider the following `Makefile.am':

```
bin_PROGRAMS = foo
foo_SOURCES  = main.cc foo.f
foo_LDADD    = libfoo.la @FLIBS@

pkglib_LTLIBRARIES = libfoo.la
libfoo_la_SOURCES  = bar.f baz.c zardoz.cc
libfoo_la_LIBADD   = $(FLIBS)
```

In this case, Automake will insist that AC_F77_LIBRARY_LDFLAGS is mentioned in

`configure.in'. Also, if @FLIBS@ hadn't been mentioned in foo_LDADD and libfoo_la_LIBADD, then Automake would have issued a warning.

How the Linker is Chosen

The following diagram demonstrates under what conditions a particular linker is chosen by Automake.

For example, if Fortran 77, C and C++ source code were to be compiled into a program, then the C++ linker will be used. In this case, if the C or Fortran 77 linkers required any special libraries that weren't included by the C++ linker, then they must be manually added to an _LDADD or _LIBADD variable by the user writing the `Makefile.am'.

source code \ Linker	C	C++	Fortran
C	x		
C++		x	
Fortran			x
C + C++		x	
C + Fortran			x
C++ + Fortran		x	
C + C++ + Fortran		x	

Fortran 77 and Autoconf

The current Automake support for Fortran 77 requires a recent enough version Autoconf that also includes support for Fortran 77. Full Fortran 77 support was added to Autoconf 2.13, so you will want to use that version of Autoconf or later.

Support for Other Languages

Automake currently only includes full support for C, C++ (see section C++ Support)and Fortran 77 (see section Fortran 77 Support). There is only rudimentary support for other languages, support for which will be improved based on user demand.

Automatic de-ANSI-fication

Although the GNU standards allow the use of ANSI C, this can have the effect of limiting portability of a package to some older compilers (notably SunOS).

Automake allows you to work around this problem on such machines by **de-ANSI-fying** each source file before the actual compilation takes place.

If the `Makefile.am' variable AUTOMAKE_OPTIONS (see section Changing Automake's Behavior) contains the option ansi2knr then code to handle de-ANSI-fication is inserted into the generated `Makefile.in'.

This causes each C source file in the directory to be treated as ANSI C. If an ANSI C compiler is available, it is used. If no ANSI C compiler is available, the ansi2knr program is used to convert the source files into K&R C, which is then compiled.

The ansi2knr program is simple-minded. It assumes the source code will be formatted in a particular way; see the ansi2knr man page for details.

Support for de-ANSI-fication requires the source files `ansi2knr.c' and `ansi2knr.1' to be in the same package as the ANSI C source; these files are distributed with Automake. Also, the package `configure.in' must call the macro AM_C_PROTOTYPES (see section Autoconf macros supplied with Automake).

Automake also handles finding the ansi2knr support files in some other directory in the current package. This is done by prepending the relative path to the appropriate directory to the ansi2knr option. For instance, suppose the package has ANSI C code in the `src' and `lib' subdirs. The files `ansi2knr.c' and `ansi2knr.1' appear in `lib'. Then this could appear in `src/Makefile.am':

```
AUTOMAKE_OPTIONS = ../lib/ansi2knr
```

If no directory prefix is given, the files are assumed to be in the current directory.

Files mentioned in `LIBOBJS` which need de-ANSI-fication will not be automatically handled. That's because `configure` will generate an object name like `` `regex.o' ``, while `make` will be looking for `` `regex_.o' `` (when de-ANSI-fying). Eventually this problem will be fixed via `autoconf` magic, but for now you must put this code into your `` `configure.in' ``, just before the `AC_OUTPUT` call:

```
# This is necessary so that .o files in LIBOBJS are also built via
# the ANSI2KNR-filtering rules.
LIBOBJS=`echo $LIBOBJS|sed 's/\.o /\$U.o /g;s/\.o$/\$U.o/'`
```

Automatic dependency tracking

As a developer it is often painful to continually update the `` `Makefile.in' `` whenever the include-file dependencies change in a project. Automake supplies a way to automatically track dependency changes, and distribute the dependencies in the generated `` `Makefile.in' ``.

Currently this support requires the use of GNU `make` and `gcc`. It might become possible in the future to supply a different dependency generating program, if there is enough demand. In the meantime, this mode is enabled by default if any C program or library is defined in the current directory, so you may get a `Must be a separator' error from non-GNU make.

When you decide to make a distribution, the `dist` target will re-run `automake` with `--include-deps' and other options. See section Creating a `` `Makefile.in' ``, and section Changing Automake's Behavior. This will cause the previously generated dependencies to be inserted into the generated `` `Makefile.in' ``, and thus into the distribution. This step also turns off inclusion of the dependency generation code, so that those who download your distribution but don't use GNU `make` and `gcc` will not get errors.

When added to the `` `Makefile.in' ``, the dependencies have all system-specific dependencies automatically removed. This can be done by listing the files in `OMIT_DEPENDENCIES'. For instance all references to system header files are removed by Automake. Sometimes it is useful to specify that a certain header file should be removed. For instance if your `` `configure.in' `` uses `AM_WITH_REGEX', then any dependency on `` `rx.h' `` or `` `regex.h' `` should be removed, because the correct one cannot be known until the user configures the package.

As it turns out, Automake is actually smart enough to handle the particular case of the regular expression header. It will also automatically omit `` `libintl.h' `` if `AM_GNU_GETTEXT' is used.

Automatic dependency tracking can be suppressed by putting `no-dependencies` in the variable `AUTOMAKE_OPTIONS`.

If you unpack a distribution made by `make dist`, and you want to turn on the dependency-tracking code again, simply re-run `automake`.

The actual dependency files are put under the build directory, in a subdirectory named `` `.deps' ``. These dependencies are machine specific. It is safe to delete them if you like; they will be automatically recreated during the next build.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Other Derived Objects

Automake can handle derived objects which are not C programs. Sometimes the support for actually building such objects must be explicitly supplied, but Automake will still automatically handle installation and distribution.

Executable Scripts

It is possible to define and install programs which are scripts. Such programs are listed using the `SCRIPTS' primary name. Automake doesn't define any dependencies for scripts; the `Makefile.am' should include the appropriate rules.

Automake does not assume that scripts are derived objects; such objects must be deleted by hand (see section What Gets Cleaned).

The automake program itself is a Perl script that is generated at configure time from `automake.in'. Here is how this is handled:

```
bin_SCRIPTS = automake
```

Since automake appears in the AC_OUTPUT macro, a target for it is automatically generated.

Script objects can be installed in bindir, sbindir, libexecdir, or pkgdatadir.

Header files

Header files are specified by the `HEADERS' family of variables. Generally header files are not installed, so the noinst_HEADERS variable will be the most used.

All header files must be listed somewhere; missing ones will not appear in the distribution. Often it is clearest to list uninstalled headers with the rest of the sources for a program. See section Building a program. Headers listed in a `_SOURCES' variable need not be listed in any `_HEADERS' variable.

Headers can be installed in includedir, oldincludedir, or pkgincludedir.

Architecture-independent data files

Automake supports the installation of miscellaneous data files using the `DATA' family of variables.

Such data can be installed in the directories datadir, sysconfdir, sharedstatedir,

`localstatedir`, or `pkgdatadir`.

By default, data files are *not* included in a distribution.

Here is how Automake installs its auxiliary data files:

```
pkgdata_DATA = clean-kr.am clean.am ...
```

Built sources

Occasionally a file which would otherwise be called `source' (e.g. a C `.h' file) is actually derived from some other file. Such files should be listed in the `BUILT_SOURCES` variable.

Built sources are also not compiled by default. You must explicitly mention them in some other `_SOURCES' variable for this to happen.

Note that, in some cases, `BUILT_SOURCES` will work in somewhat surprising ways. In order to get the built sources to work with automatic dependency tracking, the `` `Makefile' `` must depend on `$(BUILT_SOURCES)`. This can cause these sources to be rebuilt at what might seem like funny times.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Other GNU Tools

Since Automake is primarily intended to generate `Makefile.in's for use in GNU programs, it tries hard to interoperate with other GNU tools.

Emacs Lisp

Automake provides some support for Emacs Lisp. The `LISP' primary is used to hold a list of `.el' files. Possible prefixes for this primary are `lisp_' and `noinst_'. Note that if `lisp_LISP' is defined, then `configure.in' must run `AM_PATH_LISPDIR' (see section Autoconf macros supplied with Automake).

By default Automake will byte-compile all Emacs Lisp source files using the Emacs found by `AM_PATH_LISPDIR'. If you wish to avoid byte-compiling, simply define the variable `ELCFILES' to be empty. Byte-compiled Emacs Lisp files are not portable among all versions of Emacs, so it makes sense to turn this off if you expect sites to have more than one version of Emacs installed. Furthermore, many packages don't actually benefit from byte-compilation. Still, we recommend that you leave it enabled by default. It is probably better for sites with strange setups to cope for themselves than to make the installation less nice for everybody else.

Gettext

If `AM_GNU_GETTEXT' is seen in `configure.in', then Automake turns on support for GNU gettext, a message catalog system for internationalization (see section `GNU Gettext' in GNU gettext utilities).

The `gettext' support in Automake requires the addition of two subdirectories to the package, `intl' and `po'. Automake insures that these directories exist and are mentioned in `SUBDIRS'.

Furthermore, Automake checks that the definition of `ALL_LINGUAS' in `configure.in' corresponds to all the valid `.po' files, and nothing more.

Guile

Automake provides some automatic support for writing Guile modules. Automake will turn on Guile support if the `AM_INIT_GUILE_MODULE' macro is used in `configure.in'.

Right now Guile support just means that the `AM_INIT_GUILE_MODULE' macro is understood to mean:

- `AM_INIT_AUTOMAKE' is run.

- `AC_CONFIG_AUX_DIR` is run, with a path of `..'.

As the Guile module code matures, no doubt the Automake support will grow as well.

Libtool

Automake provides support for GNU Libtool (see section `Introduction' in The Libtool Manual) with the `LTLIBRARIES' primary. See section Building a Shared Library.

Java

Automake provides some minimal support for Java compilation with the `JAVA' primary.

Any `.java' files listed in a `_JAVA' variable will be compiled with `JAVAC` at build time. By default, `.class' files are not included in the distribution.

Currently Automake enforces the restriction that only one `_JAVA' primary can be used in a given `Makefile.am'. The reason for this restriction is that, in general, it isn't possible to know which `.class' files were generated from which `.java' files -- so it would be impossible to know which files to install where.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Building documentation

Currently Automake provides support for Texinfo and man pages.

Texinfo

If the current directory contains Texinfo source, you must declare it with the `TEXINFOS' primary. Generally Texinfo files are converted into info, and thus the `info_TEXINFOS` macro is most commonly used here. Note that any Texinfo source file must end in the `.texi' or `.texinfo' extension.

If the `.texi' file `@includes` `version.texi', then that file will be automatically generated. The file `version.texi' defines three Texinfo macros you can reference: `EDITION`, `VERSION`, and `UPDATED`. The first two hold the version number of your package (but are kept separate for clarity); the last is the date the primary file was last modified. The `version.texi' support requires the `mdate-sh` program; this program is supplied with Automake and automatically included when `automake` is invoked with the `--add-missing` option.

Sometimes an info file actually depends on more than one `.texi' file. For instance, in GNU Hello, `hello.texi' includes the file `gpl.texi'. You can tell Automake about these dependencies using the `texi_TEXINFOS` variable. Here is how GNU Hello does it:

```
info_TEXINFOS = hello.texi
hello_TEXINFOS = gpl.texi
```

By default, Automake requires the file `texinfo.tex' to appear in the same directory as the Texinfo source. However, if you used `AC_CONFIG_AUX_DIR` in `configure.in' (see section `Finding `configure' Input' in The Autoconf Manual), then `texinfo.tex' is looked for there. Automake supplies `texinfo.tex' if `--add-missing' is given.

If your package has Texinfo files in many directories, you can use the variable `TEXINFO_TEX` to tell Automake where to find the canonical `texinfo.tex' for your package. The value of this variable should be the relative path from the current `Makefile.am' to `texinfo.tex':

```
TEXINFO_TEX = ../doc/texinfo.tex
```

The option `no-texinfo.tex' can be used to eliminate the requirement for `texinfo.tex'. Use of the variable `TEXINFO_TEX` is preferable, however, because that allows the `dvi` target to still work.

Automake generates an `install-info` target; some people apparently use this. By default, info pages are installed by `make install'. This can be prevented via the `no-installinfo` option.

Man pages

A package can also include man pages (but see the GNU standards on this matter, section `Man Pages' in The GNU Coding Standards.) Man pages are declared using the `MANS' primary. Generally the `man_MANS` macro is used. Man pages are automatically installed in the correct subdirectory of `mandir`, based on the file extension. They are not automatically included in the distribution.

By default, man pages are installed by `make install'. However, since the GNU project does not require man pages, many maintainers do not expend effort to keep the man pages up to date. In these cases, the `no-installman` option will prevent the man pages from being installed by default. The user can still explicitly install them via `make install-man'.

Here is how the documentation is handled in GNU `cpio` (which includes both Texinfo documentation and man pages):

```
info_TEXINFOS = cpio.texi
man_MANS = cpio.1 mt.1
EXTRA_DIST = $(man_MANS)
```

Texinfo source and info pages are all considered to be source for the purposes of making a distribution.

Man pages are not currently considered to be source, because it is not uncommon for man pages to be automatically generated. For the same reason, they are not automatically included in the distribution.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

What Gets Installed

Naturally, Automake handles the details of actually installing your program once it has been built. All `PROGRAMS`, `SCRIPTS`, `LIBRARIES`, `LISP`, `DATA` and `HEADERS` are automatically installed in the appropriate places.

Automake also handles installing any specified info and man pages.

Automake generates separate `install-data` and `install-exec` targets, in case the installer is installing on multiple machines which share directory structure--these targets allow the machine-independent parts to be installed only once. The `install` target depends on both of these targets.

Automake also generates an `uninstall` target, an `installdirs` target, and an `install-strip` target.

It is possible to extend this mechanism by defining an `install-exec-local` or `install-data-local` target. If these targets exist, they will be run at `make install' time.

Variables using the standard directory prefixes `data', `info', `man', `include', `oldinclude', `pkgdata', or `pkginclude' (e.g. `data_DATA') are installed by `install-data'.

Variables using the standard directory prefixes `bin', `sbin', `libexec', `sysconf', `localstate', `lib', or `pkglib' (e.g. `bin_PROGRAMS') are installed by `install-exec'.

Any variable using a user-defined directory prefix with `exec' in the name (e.g. `myexecbin_PROGRAMS' is installed by `install-exec'. All other user-defined prefixes are installed by `install-data'.

Automake generates support for the `DESTDIR' variable in all install rules. `DESTDIR' is used during the `make install' step to relocate install objects into a staging area. Each object and path is prefixed with the value of `DESTDIR' before being copied into the install area. Here is an example of typical DESTDIR usage:

```
make DESTDIR=/tmp/staging install
```

This places install objects in a directory tree built under `/tmp/staging'. If `/gnu/bin/foo' and `/gnu/share/aclocal/foo.m4' are to be installed, the above command would install `/tmp/staging/gnu/bin/foo' and `/tmp/staging/gnu/share/aclocal/foo.m4'.

This feature is commonly used to build install images and packages. For more information, see section `Makefile Conventions' in The GNU Coding Standards.

Go to the first, previous, next, last section, table of contents.

TOC

Go to the first, previous, next, last section, table of contents.

What Gets Cleaned

The GNU Makefile Standards specify a number of different clean rules. Generally the files that can be cleaned are determined automatically by Automake. Of course, Automake also recognizes some variables that can be defined to specify additional files to clean. These variables are `MOSTLYCLEANFILES`, `CLEANFILES`, `DISTCLEANFILES`, and `MAINTAINERCLEANFILES`.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

What Goes in a Distribution

The `dist` target in the generated `Makefile.in' can be used to generate a gzip'd `tar` file for distribution. The tar file is named based on the `PACKAGE' and `VERSION' variables; more precisely it is named `package-version.tar.gz'. You can use the `make` variable `GZIP_ENV' to control how gzip is run. The default setting is `--best'.

For the most part, the files to distribute are automatically found by Automake: all source files are automatically included in a distribution, as are all `Makefile.am's and `Makefile.in's. Automake also has a built-in list of commonly used files which, if present in the current directory, are automatically included. This list is printed by `automake --help'. Also, files which are read by `configure` (i.e. the source files corresponding to the files specified in the `AC_OUTPUT` invocation) are automatically distributed.

Still, sometimes there are files which must be distributed, but which are not covered in the automatic rules. These files should be listed in the `EXTRA_DIST` variable. You can mention files from subdirectories in `EXTRA_DIST`. You can also mention a directory there; in this case the entire directory will be recursively copied into the distribution.

If you define `SUBDIRS`, Automake will recursively include the subdirectories in the distribution. If `SUBDIRS` is defined conditionally (see section Conditionals), Automake will normally include all directories that could possibly appear in `SUBDIRS` in the distribution. If you need to specify the set of directories conditionally, you can set the variable `DIST_SUBDIRS` to the exact list of subdirectories to include in the distribution.

Occasionally it is useful to be able to change the distribution before it is packaged up. If the `dist-hook` target exists, it is run after the distribution directory is filled, but before the actual tar (or shar) file is created. One way to use this is for distributing files in subdirectories for which a new `Makefile.am' is overkill:

```
dist-hook:
        mkdir $(distdir)/random
        cp -p $(srcdir)/random/a1 $(srcdir)/random/a2 $(distdir)/random
```

Automake also generates a `distcheck` target which can be help to ensure that a given distribution will actually work. `distcheck` makes a distribution, and then tries to do a `VPATH` build.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Support for test suites

Automake supports two forms of test suites.

If the variable `TESTS` is defined, its value is taken to be a list of programs to run in order to do the testing. The programs can either be derived objects or source objects; the generated rule will look both in `srcdir` and `.'. Programs needing data files should look for them in `srcdir` (which is both an environment variable and a make variable) so they work when building in a separate directory (see section `Build Directories' in The Autoconf Manual), and in particular for the `distcheck` target (see section What Goes in a Distribution).

The number of failures will be printed at the end of the run. If a given test program exits with a status of 77, then its result is ignored in the final count. This feature allows non-portable tests to be ignored in environments where they don't make sense.

The variable `TESTS_ENVIRONMENT` can be used to set environment variables for the test run; the environment variable `srcdir` is set in the rule. If all your test programs are scripts, you can also set `TESTS_ENVIRONMENT` to an invocation of the shell (e.g. `$(SHELL) -x'); this can be useful for debugging the tests.

If `dejagnu' appears in `AUTOMAKE_OPTIONS`, then a `dejagnu`-based test suite is assumed. The value of the variable `DEJATOOL` is passed as the `--tool` argument to `runtest`; it defaults to the name of the package.

The variable `RUNTESTDEFAULTFLAGS` holds the `--tool` and `--srcdir` flags that are passed to dejagnu by default; this can be overridden if necessary.

The variables `EXPECT`, `RUNTEST` and `RUNTESTFLAGS` can also be overridden to provide project-specific values. For instance, you will need to do this if you are testing a compiler toolchain, because the default values do not take into account host and target names.

In either case, the testing is done via `make check'.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Changing Automake's Behavior

Various features of Automake can be controlled by options in the `Makefile.am'. Such options are listed in a special variable named AUTOMAKE_OPTIONS. Currently understood options are:

`gnits`

`gnu`

`foreign`

`cygnus`

> Set the strictness as appropriate. The `gnits` option also implies `readme-alpha` and `check-news`.

`ansi2knr`

`path/ansi2knr`

> Turn on automatic de-ANSI-fication. See section Automatic de-ANSI-fication. If preceded by a path, the generated `Makefile.in' will look in the specified directory to find the `ansi2knr' program. Generally the path should be a relative path to another directory in the same distribution (though Automake currently does not check this).

`check-news`

> Cause `make dist` to fail unless the current version number appears in the first few lines of the `NEWS' file.

`dejagnu`

> Cause `dejagnu`-specific rules to be generated. See section Support for test suites.

`dist-shar`

> Generate a `dist-shar` target as well as the ordinary `dist` target. This new target will create a shar archive of the distribution.

`dist-zip`

> Generate a `dist-zip` target as well as the ordinary `dist` target. This new target will create a zip archive of the distribution.

`dist-tarZ`

> Generate a `dist-tarZ` target as well as the ordinary `dist` target. This new target will create a compressed tar archive of the distribution; a traditional `tar` and `compress` will be assumed. Warning: if you are actually using `GNU tar`, then the generated archive might contain nonportable constructs.

`no-dependencies`

> This is similar to using `--include-deps' on the command line, but is useful for those situations where you don't have the necessary bits to make automatic dependency tracking work See section

Automatic dependency tracking. In this case the effect is to effectively disable automatic dependency tracking.

`no-installinfo`

The generated `Makefile.in' will not cause info pages to be built or installed by default. However, `info` and `install-info` targets will still be available. This option is disallowed at `GNU' strictness and above.

`no-installman`

The generated `Makefile.in' will not cause man pages to be installed by default. However, an `install-man` target will still be available for optional installation. This option is disallowed at `GNU' strictness and above.

`no-texinfo.tex`

Don't require `texinfo.tex', even if there are texinfo files in this directory.

`readme-alpha`

If this release is an alpha release, and the file `README-alpha' exists, then it will be added to the distribution. If this option is given, version numbers are expected to follow one of two forms. The first form is `MAJOR.MINOR.ALPHA', where each element is a number; the final period and number should be left off for non-alpha releases. The second form is `MAJOR.MINORALPHA', where ALPHA is a letter; it should be omitted for non-alpha releases.

version

A version number (e.g. `0.30') can be specified. If Automake is not newer than the version specified, creation of the `Makefile.in' will be suppressed.

Unrecognized options are diagnosed by `automake`.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Miscellaneous Rules

There are a few rules and variables that didn't fit anywhere else.

Interfacing to `etags`

Automake will generate rules to generate `TAGS' files for use with GNU Emacs under some circumstances.

If any C, C++ or Fortran 77 source code or headers are present, then `tags` and `TAGS` targets will be generated for the directory.

At the topmost directory of a multi-directory package, a `tags` target file will be generated which, when run, will generate a `TAGS' file that includes by reference all `TAGS' files from subdirectories.

Also, if the variable `ETAGS_ARGS` is defined, a `tags` target will be generated. This variable is intended for use in directories which contain taggable source that `etags` does not understand.

Here is how Automake generates tags for its source, and for nodes in its Texinfo file:

```
ETAGS_ARGS = automake.in --lang=none \
 --regex='/^@node[ \t]+\([^,]+\)/\1/' automake.texi
```

If you add filenames to `ETAGS_ARGS', you will probably also want to set `TAGS_DEPENDENCIES'. The contents of this variable are added directly to the dependencies for the `tags` target.

Automake will also generate an `ID` target which will run `mkid` on the source. This is only supported on a directory-by-directory basis.

Handling new file extensions

It is sometimes useful to introduce a new implicit rule to handle a file type that Automake does not know about. If this is done, you must notify GNU Make of the new suffixes. This can be done by putting a list of new suffixes in the `SUFFIXES` variable.

For instance, currently Automake does not provide any Java support. If you wrote a macro to generate `.class' files from `.java' source files, you would also need to add these suffixes to the list:

```
SUFFIXES = .java .class
```

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Include

To include another file (perhaps for common rules), the following syntax is supported:

include ($(srcdir)|$(top_srcdir))/filename

Using files in the current directory:

```
include $(srcdir)/Makefile.extra
```

```
include Makefile.generated
```

Using a file in the top level directory:

```
include $(top_srcdir)/filename
```

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Conditionals

Automake supports a simple type of conditionals.

Before using a conditional, you must define it by using `AM_CONDITIONAL` in the `configure.in` file (see section Autoconf macros supplied with Automake). The `AM_CONDITIONAL` macro takes two arguments.

The first argument to `AM_CONDITIONAL` is the name of the conditional. This should be a simple string starting with a letter and containing only letters, digits, and underscores.

The second argument to `AM_CONDITIONAL` is a shell condition, suitable for use in a shell `if` statement. The condition is evaluated when `configure` is run.

Conditionals typically depend upon options which the user provides to the `configure` script. Here is an example of how to write a conditional which is true if the user uses the `--enable-debug' option.

```
AC_ARG_ENABLE(debug,
[  --enable-debug    Turn on debugging],
[case "${enableval}" in
  yes) debug=true ;;
  no)  debug=false ;;
  *) AC_MSG_ERROR(bad value ${enableval} for --enable-debug) ;;
esac],[debug=false])
AM_CONDITIONAL(DEBUG, test x$debug = xtrue)
```

Here is an example of how to use that conditional in `Makefile.am':

```
if DEBUG
DBG = debug
else
DBG =
endif
noinst_PROGRAMS = $(DBG)
```

This trivial example could also be handled using EXTRA_PROGRAMS (see section Building a program).

You may only test a single variable in an `if` statement. The `else` statement may be omitted. Conditionals may be nested to any depth.

Note that conditionals in Automake are not the same as conditionals in GNU Make. Automake conditionals are checked at configure time by the `configure' script, and affect the translation from

`Makefile.in' to `Makefile'. They are based on options passed to `configure' and on results that `configure' has discovered about the host system. GNU Make conditionals are checked at `make' time, and are based on variables passed to the make program or defined in the `Makefile'.

Automake conditionals will work with any make program.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

The effect of `--gnu` and `--gnits`

The `--gnu' option (or `gnu' in the `AUTOMAKE_OPTIONS' variable) causes `automake` to check the following:

- The files `INSTALL', `NEWS', `README', `COPYING', `AUTHORS', and `ChangeLog' are required at the topmost directory of the package.
- The options `no-installman' and `no-installinfo' are prohibited.

Note that this option will be extended in the future to do even more checking; it is advisable to be familiar with the precise requirements of the GNU standards. Also, `--gnu' can require certain non-standard GNU programs to exist for use by various maintainer-only targets; for instance in the future `pathchk` might be required for `make dist'.

The `--gnits' option does everything that `--gnu' does, and checks the following as well:

- `make dist' will check to make sure the `NEWS' file has been updated to the current version.
- The file `COPYING.LIB' is prohibited. The LGPL is apparently considered a failed experiment.
- `VERSION' is checked to make sure its format complies with Gnits standards.
- If `VERSION' indicates that this is an alpha release, and the file `README-alpha' appears in the topmost directory of a package, then it is included in the distribution. This is done in `--gnits' mode, and no other, because this mode is the only one where version number formats are constrained, and hence the only mode where Automake can automatically determine whether `README-alpha' should be included.
- The file `THANKS' is required.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

The effect of `--cygnus`

Cygnus Solutions has slightly different rules for how a `Makefile.in' is to be constructed. Passing `--cygnus' to `automake` will cause any generated `Makefile.in' to comply with Cygnus rules.

Here are the precise effects of `--cygnus':

- Info files are always created in the build directory, and not in the source directory.
- `texinfo.tex' is not required if a Texinfo source file is specified. The assumption is that the file will be supplied, but in a place that Automake cannot find. This assumption is an artifact of how Cygnus packages are typically bundled.
- `make dist' will look for files in the build directory as well as the source directory. This is required to support putting info files into the build directory.
- Certain tools will be searched for in the build tree as well as in the user's `PATH'. These tools are `runtest`, `expect`, `makeinfo` and `texi2dvi`.
- `--foreign` is implied.
- The options `no-installinfo' and `no-dependencies' are implied.
- The macros `AM_MAINTAINER_MODE' and `AM_CYGWIN32' are required.
- The `check` target doesn't depend on `all`.

GNU maintainers are advised to use `gnu' strictness in preference to the special Cygnus mode.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

When Automake Isn't Enough

Automake's implicit copying semantics means that many problems can be worked around by simply adding some `make` targets and rules to `` `Makefile.in' ``. Automake will ignore these additions.

There are some caveats to doing this. Although you can overload a target already used by Automake, it is often inadvisable, particularly in the topmost directory of a non-flat package. However, various useful targets have a `-local' version you can specify in your `` `Makefile.in' ``. Automake will supplement the standard target with these user-supplied targets.

The targets that support a local version are `all`, `info`, `dvi`, `check`, `install-data`, `install-exec`, `uninstall`, and the various `clean` targets (`mostlyclean`, `clean`, `distclean`, and `maintainer-clean`). Note that there are no `uninstall-exec-local` or `uninstall-data-local` targets; just use `uninstall-local`. It doesn't make sense to uninstall just data or just executables.

For instance, here is one way to install a file in `` `/etc' ``:

```
install-data-local:
        $(INSTALL_DATA) $(srcdir)/afile /etc/afile
```

Some targets also have a way to run another target, called a **hook**, after their work is done. The hook is named after the principal target, with `-hook' appended. The targets allowing hooks are `install-data`, `install-exec`, `dist`, and `distcheck`.

For instance, here is how to create a hard link to an installed program:

```
install-exec-hook:
        ln $(bindir)/program $(bindir)/proglink
```

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Distributing `Makefile.in's

Automake places no restrictions on the distribution of the resulting `Makefile.in's. We still encourage software authors to distribute their work under terms like those of the GPL, but doing so is not required to use Automake.

Some of the files that can be automatically installed via the `--add-missing` switch do fall under the GPL; examine each file to see.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

TOC

Some ideas for the future

Here are some things that might happen in the future:

- HTML support.
- The output will be cleaned up. For instance, only variables which are actually used will appear in the generated `Makefile.in'.
- There will be support for automatically recoding a distribution. The intent is to allow a maintainer to use whatever character set is most convenient locally, but for all distributions to be Unicode or ISO 10646 with the UTF-8 encoding.
- Rewrite in Guile. This won't happen in the near future, but it will eventually happen.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Macro and Variable Index

#

-

@

_

a

b

C

d

e

f

g

TOC

h

i

j

l

m

n

o

p

r

S

t

u

v

y

z

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

General Index

#

-

@

_

a

b

C

d

e

f

g

h

i

j

l

m

n

o

p

r

S

t

u

v

y

z

Go to the first, previous, next, last section, table of contents.

GNU Automake

For version 1.4, 10 January 1999

David MacKenzie and Tom Tromey

---

(1)

Much, if not most, of the information in the following sections pertaining to preprocessing Fortran 77 programs was taken almost verbatim from section `Catalogue of Rules' in The GNU Make Manual.

(2)

For example, `the cfortran package` addresses all of these inter-language issues, and runs under nearly all Fortran 77, C and C++ compilers on nearly all platforms. However, `cfortran` is not yet Free Software, but it will be in the next major release.

This document was generated on 20 May 1999 using the texi2html translator version 1.51a.

Bash Features

Bash Features

Bourne Shell Style Features

Bash is an acronym for Bourne Again SHell. The Bourne shell is the traditional Unix shell written by Stephen Bourne. All of the Bourne shell builtin commands are available in Bash, and the rules for evaluation and quoting are taken from the Posix 1003 specification for the `standard' Unix shell.

The shell builtin control features are briefly discussed here.

Looping Constructs

Note that wherever you see an `;' in the description of a command's syntax, it can be replaced indiscriminately with newlines.

Bash supports the following looping constructs.

`until`

The syntax of the `until` command is:

```
until test-commands; do consequent-commands; done
```

Execute consequent-commands as long as the final command in test-commands has an exit status which is not zero.

`while`

The syntax of the `while` command is:

```
while test-commands; do consequent-commands; done
```

Execute consequent-commands as long as the final command in test-commands has an exit status of zero.

`for`

The syntax of the for command is:

```
for name [in words ...]; do commands; done
```

Execute commands for each member in words, with name bound to the current member. If "`in words`" is not present, "`in "$@"`" is assumed.

Conditional Constructs

`if`

The syntax of the `if` command is:

```
if test-commands; then
  consequent-commands;
```


```
[else alternate-consequents;]
fi
```

Execute consequent-commands only if the final command in test-commands has an exit status of zero. If "`else alternate-consequents`" is present, and the final command in test-commands has a non-zero exit status, then execute alternate-consequents.

`case`

The syntax of the `case` command is:

```
case word in [pattern [| pattern]...) commands ;;]... esac
```

Selectively execute commands based upon word matching pattern. The `|' is used to separate multiple patterns.

Here is an example using `case` in a script that could be used to describe an interesting feature of an animal:

```
echo -n "Enter the name of an animal:"
read ANIMAL
echo -n "The $ANIMAL has "
case $ANIMAL in
  horse | dog | cat) echo -n "four";;
  man | kangaroo ) echo -n "two";;
  *) echo -n "an unknown number of";;
esac
echo "legs."
```

(T)C-Shell Style Features

The C-Shell `csh` was created by Bill Joy at UC Berkeley. It is generally considered to have better features for interactive use than the Bourne shell. Some of the `csh` features present in Bash include job control, history expansion, `protected' redirection, and several variables for controlling the interactive behaviour of the shell (e.g. `ignoreeof`).

For details on history expansion, @xref{Using History Interactively}.

Bash has tilde (~) expansion, similar, but not identical, to that of `csh`. The following table shows what unquoted words beginning with a tilde expand to.

`~`

The current value of `$HOME`.

`~/foo`

`$HOME`/foo

`~fred/foo`

The subdirectory `foo` of the home directory of the user named `fred`.

`~+/foo`

`$PWD`/foo

`~-`

`$OLDPWD`/foo

Here is a list of the commands and variables whose meanings were taken from `csh`.

`pushd`


```
pushd [dir | +n]
```

Save the current directory on a list and then CD to DIR. With no arguments, exchanges the top two directories.

`+n`

Brings the nth directory to the top of the list by rotating.

`dir`

Makes the current working directory be the top of the stack, and then cd's to DIR. You can see the saved directory list with the `dirs' command.

- popd

```
popd [+n]
```

Pops the directory stack, and cd's to the new top directory. The elements are numbered from 0 starting at the first directory listed with `dirs`; i.e. `popd` is equivalent to `popd 0`.

- dirs

```
dirs
```

Display the list of currently remembered directories. Directories find their way onto the list with the `pushd` command; you can get back up through the list with the `popd` command.

- history

```
history [n] [ [-w -r] [filename]]
```

Display the history list with line numbers. Lines listed with with a `*` have been modified. Argument of n says to list only the last n lines. Argument `-w` means write out the current history file. `-r` means to read it instead. If filename is given, then use that file, else if `$HISTFILE` has a value, use that, else use `` `~/.bash_history' ``.

- ignoreeof If this variable is set, it represents the number of consecutive `EOF`s Bash will read before exiting. By default, Bash will exit upon reading an `EOF` character.

Korn Shell Style Features

```
fc
```

```
fc [-e ename] [-nlr] [first] [last]
fc -s [pat=rep] [command]
```

Fix Command. In the first form, a range of commands from first to last is selected from the history list. First and/or last may be specified as a string (to locate the most recent command beginning with that string) or as a number (an index into the history list, where a negative number is used as an offset from the current command number). If last is not specified it is set to first. If first is not specified it is set to the previous command for editing and -16 for listing. If the `-l` flag is given, the commands are listed on standard output. The `-n` flag suppresses the command numbers when listing. The `-r` flag reverses the order of the listing. Otherwise, the editor given by ename is invoked on a file containing those commands. If ename is not given, the value of the following variable expansion is used: `${FCEDIT:-${EDITOR:-vi}}`. This says to use the value of the `FCEDIT` variable if set, or the value of the `EDITOR` variable if that is set, or `vi` if neither is set. When editing is complete, the edited commands are echoed and executed.

In the second form, command is re-executed after the substitution old=new is performed.

A useful alias to use with the `fc` command is `r='fc -s'`, so that typing `r cc` runs the last command beginning with `cc` and typing `r` re-executes the last command.

`typeset`

The `typeset` command is supplied for compatibility with the Korn shell; however, it has been made obsolete by the presence of the `declare` command, documented with the Bash specific features.

`type`

Bash's `type` command is a superset of the `type` found in Korn shells; See section Bash Builtin Commands for details.

Bash Specific Features

Shell Command Line Options

Along with the single character shell command-line options (See section The Set Builtin) there are several other options that you can use. These options must appear on the command line before the single character command options to be recognized.

`-norc`

Don't load ~/.bashrc init file. (Default if shell name is `sh').

`-rcfile filename`

Load filename init file (instead `~/.bashrc').

`-noprofile`

Don't load `~/.bash_profile' (nor `/etc/profile').

`-version`

Display the version number of this shell.

`-login`

Make this shell act as if it were directly invoked from login. This is equivalent to "exec - bash" but can be issued from another shell, such as csh. If you wanted to replace your current login shell with a bash login shell, you would say "exec bash -login".

`-nobraceexpansion`

Do not perform curly brace expansion (foo{a,b} -> fooa foob).

`-nolinediting`

Do not use the GNU Readline library to read interactive text lines.

The Set Builtin

This builtin is so overloaded that it deserves its own section. So here it is.

`set`

```
set [-aefhknotuvxldH] [arg ...]
-a
```

Mark variables which are modified or created for export.

`-e`

Exit immediately if a command exits with a non-zero status.

`-f`

Disable file name generation (globbing).

`-k`

All keyword arguments are placed in the environment for a command, not just those that precede the command name.

`-m`

Job control is enabled.

`-n`

Read commands but do not execute them.

`-o option-name`

Set the variable corresponding to option-name:

`allexport`

same as -a.

`braceexpand`

the shell will perform brace expansion.

`emacs`

use an emacs-style line editing interface.

`errexit`

same as -e.

`histexpand`

same as -H.

`ignoreeof`

the shell will not exit upon reading EOF.

`monitor`

same as -m.

`noclobber`

disallow redirection to existing files.

`noexec`

same as -n.

`noglob`

same as -f.

`nohash`

same as -d.

`notify`

notify of job termination immediately.

`nounset`

same as -u.

`verbose`

same as -v.

`vi`
: use a vi-style line editing interface.

`xtrace`
: same as -x.

- -t Exit after reading and executing one command.
- -u Treat unset variables as an error when substituting.
- -v Print shell input lines as they are read.
- -x Print commands and their arguments as they are executed.
- -l Save and restore the binding of the name in a `for` command.
- -d Disable the hashing of commands that are looked up for execution. Normally, commands are remembered in a hash table, and once found, do not have to be looked up again.
- -H Enable ! style history substitution. This flag is on by default.

Using `+' rather than `-' causes these flags to be turned off. The flags can also be used upon invocation of the shell. The current set of flags may be found in `$-`. The remaining args are positional parameters and are assigned, in order, to `$1`, `$2`, .. `$9`. If no args are given, all shell variables are printed.

Is This Shell Interactive?

You may wish to determine within a startup script whether Bash is running interactively or not. To do this, you examine the variable `$PS1`; it is unset in non-interactive shells, and set in interactive shells. Thus:

```
if [ "$PS1" = "" ]; then
  echo "This shell is not interactive"
else
  echo "This shell is interactive"
fi
```

You can ask an interactive Bash to not run your `` `~/.bashrc' `` file with the `-norc` flag. You can change the name of the `` `~/.bashrc' `` file to any other file name with `-rcfile filename`. You can ask Bash to not run your `` `~/.bash_profile' `` file with the `-noprofile` flag.

Controlling the Prompt

The value of the variable `$PROMPT_COMMAND` is examined just before Bash prints each toplevel prompt. If it is set and non-null, then the value is executed just as if you had typed it on the command line.

In addition, the following table describes the special characters which can appear in the `PS1` variable:

`\t`
: the time.

`\d`
: the date.

`\n`
: CRLF.

`\s`
: the name of the shell.

`\w`
: the current working directory.

`\W`
: the last element of PWD.

`\u`
: your username.

`\h`
: the hostname.

`\#`
: the command number of this command.

`\!`
: the history number of this command.

`\<octal>`
: the character code in octal.

`\\`
: a backslash.

Bash Startup Files

When and how Bash executes `` `~/.bash_profile' ``, `` `~/.bashrc' ``, and `` `~/.bash_logout' ``.

```
For Login shells:

    On logging in:
       If `/etc/profile' exists, then source it.

       If `~/.bash_profile' exists, then source it,
          else if `~/.bash_login' exists, then source it,
             else if `~/.profile' exists, then source it.

    On logging out:
       If `~/.bash_logout' exists, source it.

For non-login interactive shells:
    On starting up:
       If `~/.bashrc' exists, then source it.

For non-interactive shells:
    On starting up:
       If the environment variable ENV is non-null, source the
       file mentioned there.
```

So, typically, your `~/.bash_profile` contains the line

```
if [ -f `~/.bashrc' ]; then source `~/.bashrc'; fi
```

after (or before) any login specific initializations.

Bash Builtin Commands

`builtin`

`builtin [shell-builtin [args]]`

Run a shell builtin. This is useful when you wish to rename a shell builtin to be a function, but need the functionality of the builtin within the function itself.

`declare`

`declare [-[frxi]] name[=value]`

Declare variables and/or give them attributes. If no names are given, then display the values of variables instead. `-f` means to use function names only. `-r` says to make names readonly. `-x` says to make names export. `-i` says that the variable is to be treated as an integer; arithmetic evaluation (see `let`) will be done when the variable is assigned to. Using `+` instead of `-` turns off the attribute instead. When used in a function, makes names local, as with the `local` command.

`type`

`type [-all] [-type | -path] [name ...]`

For each name, indicate how it would be interpreted if used as a command name.

If the `-type` flag is used, `type` returns a single word which is one of "alias", "function", "builtin", "file" or "", if name is an alias, shell function, shell builtin, disk file, or unfound, respectively.

If the `-path` flag is used, `type` either returns the name of the disk file that would be exec'ed, or nothing if `-type` wouldn't return "file".

If the `-all` flag is used, returns all of the places that contain an executable named file. This includes aliases and functions, if and only if the `-path` flag is not also used.

`enable`

`enable [-n] [name ...]`

Enable and disable builtin shell commands. This allows you to use a disk command which has the same name as a shell builtin. If `-n` is used, the names become disabled. Otherwise names are enabled. For example, to use the `test` found on your path instead of the shell builtin version, you type `enable -n test`.

`help`

`help [pattern]`

Display helpful information about builtin commands. If pattern is specified, gives detailed help on all commands matching pattern, otherwise a list of the builtins is printed.

`command`

`command [command [args ...]]`

Runs command with arg ignoring shell functions. If you have a shell function called `ls`, and you wish to call the command `ls`, you can say "command ls".

`hash`

`hash [-r] [name]`

For each name, the full pathname of the command is determined and remembered. The `-r` option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is presented.

`local`

`local name[=value]`

Create a local variable called name, and give it value. `local` can only be used within a function; it makes the variable name have a visible scope restricted to that function and its children.

`readonly`

`readonly [-f] [name ...]`

The given names are marked readonly and the values of these names may not be changed by subsequent assignment. If the -f option is given, the functions corresponding to the names are so marked. If no arguments are given, a list of all readonly names is printed.

`ulimit`

`ulimit [-acdmstfpn [limit]]`

Ulimit provides control over the resources available to processes started by the shell, on systems that allow such control. If an option is given, it is interpreted as follows:

`-a`
: all current limits are reported.

`-c`
: the maximum size of core files created.

`-d`
: the maximum size of a process's data segment.

`-m`
: the maximum resident set size.

`-s`
: the maximum stack size.

`-t`
: the maximum amount of cpu time in seconds.

`-f`
: the maximum size of files created by the shell.

`-p`
: the pipe buffer size.

`-n`
: the maximum number of open file descriptors.

If limit is given, it is the new value of the specified resource. Otherwise, the current value of the specified resource is printed. If no option is given, then `-f` is assumed. Values are in 1k increments, except for `-t`, which is in seconds, and `-p`, which is in increments of 512 bytes.

Bash Variables

`history_control`

Set to a value of "ignorespace", it means don't enter lines which begin with a SPC on the history list. Set to a value of "ignoredups", it means don't enter lines which match the last entered line. Unset, or any other value than those above mean to save all lines on the history list.

`HISTFILE`

The name of the file that the command history is saved in.

`HISTSIZE`

If set, this is the maximum number of commands to remember in the history.

`histchars`

Up to three characters which control history expansion, quick substitution, and tokenization. The first character is the history-expansion-char, that is, the character which signifies the start of a history expansion, normally `!'. The second character is the character which signifies `quick substitution' when seen as the first character on a line, normally `^'. The optional third character is the character which signifies the remainder of the line is a comment, when found as the first character of a word, usually `#'.

`hostname_completion_file`

Contains the name of a file in the same format as `` `/etc/hosts' `` that should be read when the shell needs to complete a hostname. You can change the file interactively; the next time you want to complete a hostname Bash will add the contents of the new file to the already existing database.

`MAILCHECK`

How often (in seconds) that the shell should check for mail in the file(s) specified in `MAILPATH`.

`MAILPATH`

Colon separated list of pathnames to check for mail in. You can also specify what message is printed by separating the pathname from the message with a `?'. `$_` stands for the name of the current mailfile. For example:

```
MAILPATH='/usr/spool/mail/bfox?"You have mail":~/shell-mail?"$_ has mail!"'
```

`ignoreeof`

`IGNOREEOF`

Controls the action of the shell on receipt of an `EOF` character as the sole input. If set, then the value of it is the number of `EOF` characters that can be seen in a row as sole input characters before the shell will exit. If the variable exists but does not have a numeric value (or has no value) then the default is 10. if the variable does not exist, then `EOF` signifies the end of input to the shell. This is only in effect for interactive shells.

`auto_resume`

This variable controls how the shell interacts with the user and job control. If this variable exists then single word simple commands without redirects are treated as candidates for resumption of an existing job. There is no ambiguity allowed; if you have more than one job beginning with the string that you have typed, then the most recently accessed job will be selected.

`no_exit_on_failed_exec`

If this variable exists, the shell will not exit in the case that it couldn't execute the file specified in the `exec` command.

`PROMPT_COMMAND`

If present, this contains a string which is a command to execute before the printing of each toplevel prompt.

`nolinks`

If present, says not to follow symbolic links when doing commands that change the current working direct
By default, bash follows the logical chain of directories when performing `cd` type commands.

For example, if `` `/usr/sys' `` is a link to `` `/usr/local/sys' `` then:

```
cd /usr/sys; echo $PWD -> /usr/sys
cd ..; pwd -> /usr
```

If `nolinks` exists, then:

```
cd /usr/sys; echo $PWD -> /usr/local/sys
cd ..; pwd -> /usr/local
```

Variable Index

a

- auto_resume

h

- histchars
- HISTFILE
- history_control
- HISTSIZE
- hostname_completion_file

i

- IGNOREEOF
- ignoreeof

m

- MAILCHECK
- MAILPATH

n

- no_exit_on_failed_exec
- nolinks

p

- PROMPT_COMMAND

Concept Index

b

- builtin

c

- case
- command

d

- declare

e

- enable

f

- fc
- for

h

- hash
- help
- History, how to use

i

- if

l

- local

p

- pushd

r

- Readline, how to use
- readonly

s

- set

t

- type
- typeset

u

- ulimit
- until

w

- while

Bison

The YACC-compatible Parser Generator

November 1995, Bison Version 1.25

by Charles Donnelly and Richard Stallman

Go to the next section.

Introduction

Bison is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar. Once you are proficient with Bison, you may use it to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.

Bison is upward compatible with Yacc: all properly-written Yacc grammars ought to work with Bison with no change. Anyone familiar with Yacc should be able to use Bison with little trouble. You need to be fluent in C programming in order to use Bison or to understand this manual.

We begin with tutorial chapters that explain the basic concepts of using Bison and show three explained examples, each building on the last. If you don't know Bison or Yacc, start by reading these chapters. Reference chapters follow which describe specific aspects of Bison in detail.

Bison was written primarily by Robert Corbett; Richard Stallman made it Yacc-compatible. Wilfred Hansen of Carnegie Mellon University added multicharacter string literals and other features.

This edition corresponds to version 1.25 of Bison.

Go to the next section.

Go to the previous, next section.

Conditions for Using Bison

As of Bison version 1.24, we have changed the distribution terms for `yyparse` to permit using Bison's output in non-free programs. Formerly, Bison parsers could be used only in programs that were free software.

The other GNU programming tools, such as the GNU C compiler, have never had such a requirement. They could always be used for non-free software. The reason Bison was different was not due to a special policy decision; it resulted from applying the usual General Public License to all of the Bison source code.

The output of the Bison utility--the Bison parser file--contains a verbatim copy of a sizable piece of Bison, which is the code for the `yyparse` function. (The actions from your grammar are inserted into this function at one point, but the rest of the function is not changed.) When we applied the GPL terms to the code for `yyparse`, the effect was to restrict the use of Bison output to free software.

We didn't change the terms because of sympathy for people who want to make software proprietary. **Software should be free.** But we concluded that limiting Bison's use to free software was doing little to encourage people to make other software free. So we decided to make the practical conditions for using Bison match the practical conditions for using the other GNU tools.

Go to the previous, next section.

Go to the previous, next section.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.
2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.
3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
   1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work

based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regard of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
    1. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
    2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
    3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

    The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

    If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the

Program (or any work based on the Program), you indicate your acceptance of this License to do and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted

by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.
```


```
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
`Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Go to the previous, next section.

Go to the previous, next section.

The Concepts of Bison

This chapter introduces many of the basic concepts without which the details of Bison will not make sense. If you do not already know how to use Bison or Yacc, we suggest you start by reading this chapter carefully.

Languages and Context-Free Grammars

In order for Bison to parse a language, it must be described by a context-free grammar. This means that you specify one or more syntactic groupings and give rules for constructing them from their parts. For example, in the C language, one kind of grouping is called an `expression'. One rule for making an expression might be, "An expression can be made of a minus sign and another expression". Another would be, "An expression can be an integer". As you can see, rules are often recursive, but there must be at least one rule which leads out of the recursion.

The most common formal system for presenting such rules for humans to read is Backus-Naur Form or "BNF", which was developed in order to specify the language Algol 60. Any grammar expressed in BNF is a context-free grammar. The input to Bison is essentially machine-readable BNF.

Not all context-free languages can be handled by Bison, only those that are LALR(1). In brief, this means that it must be possible to tell how to parse any portion of an input string with just a single token of look-ahead. Strictly speaking, that is a description of an LR(1) grammar, and LALR(1) involves additional restrictions that are hard to explain simply; but it is rare in actual practice to find an LR(1) grammar that fails to be LALR(1). See section Mysterious Reduce/Reduce Conflicts, for more information on this.

In the formal grammatical rules for a language, each kind of syntactic unit or grouping is named by a symbol. Those which are built by grouping smaller constructs according to grammatical rules are called nonterminal symbols; those which can't be subdivided are called terminal symbols or token types. We call a piece of input corresponding to a single terminal symbol a token, and a piece corresponding to a single nonterminal symbol a grouping.

We can use the C language as an example of what symbols, terminal and nonterminal, mean. The tokens of C are identifiers, constants (numeric and string), and the various keywords, arithmetic operators and punctuation marks. So the terminal symbols of a grammar for C include `identifier', `number', `string', plus one symbol for each keyword, operator or punctuation mark: `if', `return', `const', `static', `int', `char', `plus-sign', `open-brace', `close-brace', `comma' and many more. (These tokens can be subdivided into characters, but that is a matter of lexicography, not grammar.)

Here is a simple C function subdivided into tokens:

```
int             /* keyword `int' */
square (x)      /* identifier, open-paren, */
```


```
                /* identifier, close-paren */
    int x;      /* keyword `int', identifier, semicolon */
{               /* open-brace */
  return x * x; /* keyword `return', identifier, */
                /* asterisk, identifier, semicolon */
}               /* close-brace */
```

The syntactic groupings of C include the expression, the statement, the declaration, and the function definition. These are represented in the grammar of C by nonterminal symbols `expression', `statement', `declaration' and `function definition'. The full grammar uses dozens of additional language constructs, each with its own nonterminal symbol, in order to express the meanings of these four. The example above is a function definition; it contains one declaration, and one statement. In the statement, each `x' is an expression and so is `x * x'.

Each nonterminal symbol must have grammatical rules showing how it is made out of simpler constructs. For example, one kind of C statement is the `return` statement; this would be described with a grammar rule which reads informally as follows:

> A `statement' can be made of a `return' keyword, an `expression' and a `semicolon'.

There would be many other rules for `statement', one for each kind of statement in C.

One nonterminal symbol must be distinguished as the special one which defines a complete utterance in the language. It is called the start symbol. In a compiler, this means a complete input program. In the C language, the nonterminal symbol `sequence of definitions and declarations' plays this role.

For example, `1 + 2' is a valid C expression--a valid part of a C program--but it is not valid as an *entire* C program. In the context-free grammar of C, this follows from the fact that `expression' is not the start symbol.

The Bison parser reads a sequence of tokens as its input, and groups the tokens using the grammar rules. If the input is valid, the end result is that the entire token sequence reduces to a single grouping whose symbol is the grammar's start symbol. If we use a grammar for C, the entire input must be a `sequence of definitions and declarations'. If not, the parser reports a syntax error.

From Formal Rules to Bison Input

A formal grammar is a mathematical construct. To define the language for Bison, you must write a file expressing the grammar in Bison syntax: a Bison grammar file. See section Bison Grammar Files.

A nonterminal symbol in the formal grammar is represented in Bison input as an identifier, like an identifier in C. By convention, it should be in lower case, such as `expr`, `stmt` or `declaration`.

The Bison representation for a terminal symbol is also called a token type. Token types as well can be represented as C-like identifiers. By convention, these identifiers should be upper case to distinguish them from nonterminals: for example, `INTEGER`, `IDENTIFIER`, `IF` or `RETURN`. A terminal symbol that stands for a particular keyword in the language should be named after that keyword converted to upper case. The terminal symbol `error` is reserved for error recovery. See section Symbols, Terminal

and Nonterminal.

A terminal symbol can also be represented as a character literal, just like a C character constant. You should do this whenever a token is just a single character (parenthesis, plus-sign, etc.): use that same character in a literal as the terminal symbol for that token.

A third way to represent a terminal symbol is with a C string constant containing several characters. See section Symbols, Terminal and Nonterminal, for more information.

The grammar rules also have an expression in Bison syntax. For example, here is the Bison rule for a C `return` statement. The semicolon in quotes is a literal character token, representing part of the C syntax for the statement; the naked semicolon, and the colon, are Bison punctuation used in every rule.

```
stmt:   RETURN expr ';'
        ;
```

See section Syntax of Grammar Rules.

Semantic Values

A formal grammar selects tokens only by their classifications: for example, if a rule mentions the terminal symbol `integer constant', it means that *any* integer constant is grammatically valid in that position. The precise value of the constant is irrelevant to how to parse the input: if `x+4' is grammatical then `x+1' or `x+3989' is equally grammatical.

But the precise value is very important for what the input means once it is parsed. A compiler is useless if it fails to distinguish between 4, 1 and 3989 as constants in the program! Therefore, each token in a Bison grammar has both a token type and a semantic value. See section Defining Language Semantics, for details.

The token type is a terminal symbol defined in the grammar, such as `INTEGER`, `IDENTIFIER` or `','`. It tells everything you need to know to decide where the token may validly appear and how to group it with other tokens. The grammar rules know nothing about tokens except their types.

The semantic value has all the rest of the information about the meaning of the token, such as the value of an integer, or the name of an identifier. (A token such as `','` which is just punctuation doesn't need to have any semantic value.)

For example, an input token might be classified as token type `INTEGER` and have the semantic value 4. Another input token might have the same token type `INTEGER` but value 3989. When a grammar rule says that `INTEGER` is allowed, either of these tokens is acceptable because each is an `INTEGER`. When the parser accepts the token, it keeps track of the token's semantic value.

Each grouping can also have a semantic value as well as its nonterminal symbol. For example, in a calculator, an expression typically has a semantic value that is a number. In a compiler for a programming language, an expression typically has a semantic value that is a tree structure describing the meaning of the expression.

Semantic Actions

In order to be useful, a program must do more than parse input; it must also produce some output based on the input. In a Bison grammar, a grammar rule can have an action made up of C statements. Each time the parser recognizes a match for that rule, the action is executed. See section Actions. Most of the time, the purpose of an action is to compute the semantic value of the whole construct from the semantic values of its parts. For example, suppose we have a rule which says an expression can be the sum of two expressions. When the parser recognizes such a sum, each of the subexpressions has a semantic value which describes how it was built up. The action for this rule should create a similar sort of value for the newly recognized larger expression.

For example, here is a rule that says an expression can be the sum of two subexpressions:

```
expr: expr '+' expr   { $$ = $1 + $3; }
        ;
```

The action says how to produce the semantic value of the sum expression from the values of the two subexpressions.

Bison Output: the Parser File

When you run Bison, you give it a Bison grammar file as input. The output is a C source file that parses the language described by the grammar. This file is called a Bison parser. Keep in mind that the Bison utility and the Bison parser are two distinct programs: the Bison utility is a program whose output is the Bison parser that becomes part of your program.

The job of the Bison parser is to group tokens into groupings according to the grammar rules--for example, to build identifiers and operators into expressions. As it does this, it runs the actions for the grammar rules it uses.

The tokens come from a function called the lexical analyzer that you must supply in some fashion (such as by writing it in C). The Bison parser calls the lexical analyzer each time it wants a new token. It doesn't know what is "inside" the tokens (though their semantic values may reflect this). Typically the lexical analyzer makes the tokens by parsing characters of text, but Bison does not depend on this. See section The Lexical Analyzer Function `yylex`.

The Bison parser file is C code which defines a function named `yyparse` which implements that grammar. This function does not make a complete C program: you must supply some additional functions. One is the lexical analyzer. Another is an error-reporting function which the parser calls to report an error. In addition, a complete C program must start with a function called `main`; you have to provide this, and arrange for it to call `yyparse` or the parser will never run. See section Parser C-Language Interface.

Aside from the token type names and the symbols in the actions you write, all variable and function names used in the Bison parser file begin with `yy' or `YY'. This includes interface functions such as the

lexical analyzer function `yylex`, the error reporting function `yyerror` and the parser function `yyparse` itself. This also includes numerous identifiers used for internal purposes. Therefore, you should avoid using C identifiers starting with `yy' or `YY' in the Bison grammar file except for the ones defined in this manual.

Stages in Using Bison

The actual language-design process using Bison, from grammar specification to a working compiler or interpreter, has these parts:

1. Formally specify the grammar in a form recognized by Bison (see section Bison Grammar Files). For each grammatical rule in the language, describe the action that is to be taken when an instance of that rule is recognized. The action is described by a sequence of C statements.
2. Write a lexical analyzer to process input and pass tokens to the parser. The lexical analyzer may be written by hand in C (see section The Lexical Analyzer Function `yylex`). It could also be produced using Lex, but the use of Lex is not discussed in this manual.
3. Write a controlling function that calls the Bison-produced parser.
4. Write error-reporting routines.

To turn this source code as written into a runnable program, you must follow these steps:

1. Run Bison on the grammar to produce the parser.
2. Compile the code output by Bison, as well as any other source files.
3. Link the object files to produce the finished product.

The Overall Layout of a Bison Grammar

The input file for the Bison utility is a Bison grammar file. The general form of a Bison grammar file is as follows:

```
%{
C declarations
%}

Bison declarations

%%
Grammar rules
%%
Additional C code
```

The `%%', `%{' and `%}' are punctuation that appears in every Bison grammar file to separate the sections.

The C declarations may define types and variables used in the actions. You can also use preprocessor

commands to define macros used there, and use `#include` to include header files that do any of these things.

The Bison declarations declare the names of the terminal and nonterminal symbols, and may also describe operator precedence and the data types of semantic values of various symbols.

The grammar rules define how to construct each nonterminal symbol from its parts.

The additional C code can contain any C code you want to use. Often the definition of the lexical analyzer `yylex` goes here, plus subroutines called by the actions in the grammar rules. In a simple program, all the rest of the program can go here.

Go to the previous, next section.

Go to the previous, next section.

Examples

Now we show and explain three sample programs written using Bison: a reverse polish notation calculator, an algebraic (infix) notation calculator, and a multi-function calculator. All three have been tested under BSD Unix 4.3; each produces a usable, though limited, interactive desk-top calculator.

These examples are simple, but Bison grammars for real programming languages are written the same way.

Reverse Polish Notation Calculator

The first example is that of a simple double-precision reverse polish notation calculator (a calculator using postfix operators). This example provides a good starting point, since operator precedence is not an issue. The second example will illustrate how operator precedence is handled.

The source code for this calculator is named `` `rpcalc.y' ``. The `.y' extension is a convention used for Bison input files.

Declarations for `rpcalc`

Here are the C and Bison declarations for the reverse polish notation calculator. As in C, comments are placed between `/*...*/'.

```
/* Reverse polish notation calculator. */

%{
#define YYSTYPE double
#include <math.h>
%}

%token NUM

%% /* Grammar rules and actions follow */
```

The C declarations section (see section The C Declarations Section) contains two preprocessor directives.

The `#define` directive defines the macro `YYSTYPE`, thus specifying the C data type for semantic values of both tokens and groupings (see section Data Types of Semantic Values). The Bison parser will use whatever type `YYSTYPE` is defined as; if you don't define it, `int` is the default. Because we specify `double`, each token and each expression has an associated value, which is a floating point number.

The `#include` directive is used to declare the exponentiation function `pow`.

The second section, Bison declarations, provides information to Bison about the token types (see sec The Bison Declarations Section). Each terminal symbol that is not a single-character literal must be declared here. (Single-character literals normally don't need to be declared.) In this example, all the arithmetic operators are designated by single-character literals, so the only terminal symbol that needs to be declared is `NUM`, the token type for numeric constants.

Grammar Rules for `rpcalc`

Here are the grammar rules for the reverse polish notation calculator.

```
input:    /* empty */
        | input line
;

line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;

exp:      NUM             { $$ = $1;         }
        | exp exp '+'     { $$ = $1 + $2;    }
        | exp exp '-'     { $$ = $1 - $2;    }
        | exp exp '*'     { $$ = $1 * $2;    }
        | exp exp '/'     { $$ = $1 / $2;    }
      /* Exponentiation */
        | exp exp '^'     { $$ = pow ($1, $2); }
      /* Unary minus    */
        | exp 'n'         { $$ = -$1;        }
;
%%
```

The groupings of the rpcalc "language" defined here are the expression (given the name `exp`), the line of input (`line`), and the complete input transcript (`input`). Each of these nonterminal symbols has several alternate rules, joined by the `|' punctuator which is read as "or". The following sections explain what these rules mean.

The semantics of the language is determined by the actions taken when a grouping is recognized. The actions are the C code that appears inside braces. See section Actions.

You must specify these actions in C, but Bison provides the means for passing semantic values between the rules. In each action, the pseudo-variable `$$` stands for the semantic value for the grouping that the rule is going to construct. Assigning a value to `$$` is the main job of most actions. The semantic values of the components of the rule are referred to as `$1`, `$2`, and so on.

Explanation of `input`

Consider the definition of `input`:


```
input:    /* empty */
        | input line
;
```

This definition reads as follows: "A complete input is either an empty string, or a complete input followed by an input line". Notice that "complete input" is defined in terms of itself. This definition is said to be left recursive since `input` appears always as the leftmost symbol in the sequence. See section Recursive Rules.

The first alternative is empty because there are no symbols between the colon and the first `|'; this means that `input` can match an empty string of input (no tokens). We write the rules this way because it is legitimate to type Ctrl-d right after you start the calculator. It's conventional to put an empty alternative first and write the comment `/* empty */' in it.

The second alternate rule (`input line`) handles all nontrivial input. It means, "After reading any number of lines, read one more line if possible." The left recursion makes this rule into a loop. Since the first alternative matches empty input, the loop can be executed zero or more times.

The parser function `yyparse` continues to process input until a grammatical error is seen or the lexical analyzer says there are no more input tokens; we will arrange for the latter to happen at end of file.

Explanation of `line`

Now consider the definition of `line`:

```
line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;
```

The first alternative is a token which is a newline character; this means that rpcalc accepts a blank line (and ignores it, since there is no action). The second alternative is an expression followed by a newline. This is the alternative that makes rpcalc useful. The semantic value of the `exp` grouping is the value of `$1` because the `exp` in question is the first symbol in the alternative. The action prints this value, which is the result of the computation the user asked for.

This action is unusual because it does not assign a value to `$$`. As a consequence, the semantic value associated with the `line` is uninitialized (its value will be unpredictable). This would be a bug if that value were ever used, but we don't use it: once rpcalc has printed the value of the user's input line, that value is no longer needed.

Explanation of `expr`

The `exp` grouping has several rules, one for each kind of expression. The first rule handles the simplest expressions: those that are just numbers. The second handles an addition-expression, which looks like two expressions followed by a plus-sign. The third handles subtraction, and so on.


```
exp:      NUM
        | exp exp '+'     { $$ = $1 + $2;    }
        | exp exp '-'     { $$ = $1 - $2;    }
        ...
        ;
```

We have used `|' to join all the rules for `exp', but we could equally well have written them separately:

```
exp:      NUM ;
exp:      exp exp '+'     { $$ = $1 + $2;    } ;
exp:      exp exp '-'     { $$ = $1 - $2;    } ;
        ...
```

Most of the rules have actions that compute the value of the expression in terms of the value of its parts. For example, in the rule for addition, `$1' refers to the first component `exp' and `$2' refers to the second one. The third component, `'+'', has no meaningful associated semantic value, but if it had one you could refer to it as `$3'. When `yyparse' recognizes a sum expression using this rule, the sum of the two subexpressions' values is produced as the value of the entire expression. See section Actions.

You don't have to give an action for every rule. When a rule has no action, Bison by default copies the value of `$1' into `$$'. This is what happens in the first rule (the one that uses `NUM').

The formatting shown here is the recommended convention, but Bison does not require it. You can add or change whitespace as much as you wish. For example, this:

```
exp   : NUM | exp exp '+' {$$ = $1 + $2; } | ...
```

means the same thing as this:

```
exp:      NUM
        | exp exp '+'    { $$ = $1 + $2; }
        | ...
```

The latter, however, is much more readable.

The `rpcalc' Lexical Analyzer

The lexical analyzer's job is low-level parsing: converting characters or sequences of characters into tokens. The Bison parser gets its tokens by calling the lexical analyzer. See section The Lexical Analyzer Function `yylex'.

Only a simple lexical analyzer is needed for the RPN calculator. This lexical analyzer skips blanks and tabs, then reads in numbers as `double' and returns them as `NUM' tokens. Any other character that isn't part of a number is a separate token. Note that the token-code for such a single-character token is the character itself.

The return value of the lexical analyzer function is a numeric code which represents a token type. The

same text used in Bison rules to stand for this token type is also a C expression for the numeric code for the type. This works in two ways. If the token type is a character literal, then its numeric code is the ASCII code for that character; you can use the same character literal in the lexical analyzer to express the number. If the token type is an identifier, that identifier is defined by Bison as a C macro whose definition is the appropriate number. In this example, therefore, `NUM` becomes a macro for `yylex` to use.

The semantic value of the token (if it has one) is stored into the global variable `yylval`, which is where the Bison parser will look for it. (The C data type of `yylval` is `YYSTYPE`, which was defined at the beginning of the grammar; see section Declarations for `rpcalc`.)

A token type code of zero is returned if the end-of-file is encountered. (Bison recognizes any nonpositive value as indicating the end of the input.)

Here is the code for the lexical analyzer:

```
/* Lexical analyzer returns a double floating point
   number on the stack and the token NUM, or the ASCII
   character read if not a number.  Skips all blanks
   and tabs, returns 0 for EOF. */

#include <ctype.h>

yylex ()
{
  int c;

  /* skip white space  */
  while ((c = getchar ()) == ' ' || c == '\t')
    ;
  /* process numbers   */
  if (c == '.' || isdigit (c))
    {
      ungetc (c, stdin);
      scanf ("%lf", &yylval);
      return NUM;
    }
  /* return end-of-file  */
  if (c == EOF)
    return 0;
  /* return single chars */
  return c;
}
```

The Controlling Function

In keeping with the spirit of this example, the controlling function is kept to the bare minimum. The only requirement is that it call `yyparse` to start the process of parsing.

```
main ()
{
  yyparse ();
}
```

The Error Reporting Routine

When `yyparse` detects a syntax error, it calls the error reporting function `yyerror` to print an error message (usually but not always `"parse error"`). It is up to the programmer to supply `yyerror` (see section Parser C-Language Interface), so here is the definition we will use:

```
#include <stdio.h>

yyerror (s)  /* Called by yyparse on error */
     char *s;
{
  printf ("%s\n", s);
}
```

After `yyerror` returns, the Bison parser may recover from the error and continue parsing if the grammar contains a suitable error rule (see section Error Recovery). Otherwise, `yyparse` returns nonzero. We have not written any error rules in this example, so any invalid input will cause the calculator program to exit. This is not clean behavior for a real calculator, but it is adequate in the first example.

Running Bison to Make the Parser

Before running Bison to produce a parser, we need to decide how to arrange all the source code in one or more source files. For such a simple example, the easiest thing is to put everything in one file. The definitions of `yylex`, `yyerror` and `main` go at the end, in the "additional C code" section of the file (see section The Overall Layout of a Bison Grammar).

For a large project, you would probably have several source files, and use `make` to arrange to recompile them.

With all the source in a single file, you use the following command to convert it into a parser file:

```
bison file_name.y
```

In this example the file was called `` `rpcalc.y' `` (for "Reverse Polish CALCulator"). Bison produces a

file named `file_name.tab.c', removing the `.y' from the original file name. The file output by Bison contains the source code for `yyparse`. The additional functions in the input file (`yylex`, `yyerror` and `main`) are copied verbatim to the output.

Compiling the Parser File

Here is how to compile and run the parser file:

```
# List files in current directory.
% ls
rpcalc.tab.c  rpcalc.y

# Compile the Bison parser.
# `-lm' tells compiler to search math library for pow.
% cc rpcalc.tab.c -lm -o rpcalc

# List files again.
% ls
rpcalc  rpcalc.tab.c  rpcalc.y
```

The file `rpcalc' now contains the executable code. Here is an example session using `rpcalc`.

```
% rpcalc
4 9 +
13
3 7 + 3 4 5 *+-
-13
3 7 + 3 4 5 * + - n                  Note the unary minus, `n'
13
5 6 / 4 n +
-3.166666667
3 4 ^                                Exponentiation
81
^D                                   End-of-file indicator
%
```

Infix Notation Calculator: `calc`

We now modify rpcalc to handle infix operators instead of postfix. Infix notation involves the concept of operator precedence and the need for parentheses nested to arbitrary depth. Here is the Bison code for `calc.y', an infix desk-top calculator.

```
/* Infix notation calculator--calc */
```


```
%{
#define YYSTYPE double
#include <math.h>
%}

/* BISON Declarations */
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG     /* negation--unary minus */
%right '^'    /* exponentiation        */

/* Grammar follows */
%%
input:    /* empty string */
        | input line
;

line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;

exp:      NUM                { $$ = $1;         }
        | exp '+' exp        { $$ = $1 + $3;    }
        | exp '-' exp        { $$ = $1 - $3;    }
        | exp '*' exp        { $$ = $1 * $3;    }
        | exp '/' exp        { $$ = $1 / $3;    }
        | '-' exp  %prec NEG { $$ = -$2;        }
        | exp '^' exp        { $$ = pow ($1, $3); }
        | '(' exp ')'        { $$ = $2;         }
;
%%
```

The functions `yylex`, `yyerror` and `main` can be the same as before.

There are two important new features shown in this code.

In the second section (Bison declarations), `%left` declares token types and says they are left-associative operators. The declarations `%left` and `%right` (right associativity) take the place of `%token` which is used to declare a token type name without associativity. (These tokens are single-character literals, which ordinarily don't need to be declared. We declare them here to specify the associativity.)

Operator precedence is determined by the line ordering of the declarations; the higher the line number of the declaration (lower on the page or screen), the higher the precedence. Hence, exponentiation has the highest precedence, unary minus (`NEG`) is next, followed by `*' and `/', and so on. See section Operator Precedence.

The other important new feature is the `%prec` in the grammar section for the unary minus operator. The `%prec` simply instructs Bison that the rule `| '-' exp' has the same precedence as `NEG`---in this case the next-to-highest. See section Context-Dependent Precedence.

Here is a sample run of `calc.y':

```
% calc
4 + 4.5 - (34/(8*3+-3))
6.880952381
-56 + 2
-54
3 ^ 2
9
```

Simple Error Recovery

Up to this point, this manual has not addressed the issue of error recovery---how to continue parsing after the parser detects a syntax error. All we have handled is error reporting with `yyerror`. Recall that by default `yyparse` returns after calling `yyerror`. This means that an erroneous input line causes the calculator program to exit. Now we show how to rectify this deficiency.

The Bison language itself includes the reserved word `error`, which may be included in the grammar rules. In the example below it has been added to one of the alternatives for `line`:

```
line:     '\n'
        | exp '\n'   { printf ("\t%.10g\n", $1); }
        | error '\n' { yyerrok;                  }
;
```

This addition to the grammar allows for simple error recovery in the event of a parse error. If an expression that cannot be evaluated is read, the error will be recognized by the third rule for `line`, and parsing will continue. (The `yyerror` function is still called upon to print its message as well.) The action executes the statement `yyerrok`, a macro defined automatically by Bison; its meaning is that error recovery is complete (see section Error Recovery). Note the difference between `yyerrok` and `yyerror`; neither one is a misprint.

This form of error recovery deals with syntax errors. There are other kinds of errors; for example, division by zero, which raises an exception signal that is normally fatal. A real calculator program must handle this signal and use `longjmp` to return to `main` and resume parsing input lines; it would also have to discard the rest of the current line of input. We won't discuss this issue further because it is not specific to Bison programs.

Multi-Function Calculator: `mfcalc`

Now that the basics of Bison have been discussed, it is time to move on to a more advanced problem. The above calculators provided only five functions, `+', `-', `*', `/' and `^'. It would be nice to have a calculator that provides other mathematical functions such as `sin`, `cos`, etc.

It is easy to add new operators to the infix calculator as long as they are only single-character literals. The lexical analyzer `yylex` passes back all non-number characters as tokens, so new grammar rules suffice for adding a new operator. But we want something more flexible: built-in functions whose syntax has this form:

```
function_name (argument)
```

At the same time, we will add memory to the calculator, by allowing you to create named variables, store values in them, and use them later. Here is a sample session with the multi-function calculator:

```
% mfcalc
pi = 3.141592653589
3.1415926536
sin(pi)
0.0000000000
alpha = beta1 = 2.3
2.3000000000
alpha
2.3000000000
ln(alpha)
0.8329091229
exp(ln(beta1))
2.3000000000
%
```

Note that multiple assignment and nested function calls are permitted.

Declarations for `mfcalc`

Here are the C and Bison declarations for the multi-function calculator.

```
%{
#include <math.h>  /* For math functions, cos(), sin(), etc. */
#include "calc.h"  /* Contains definition of `symrec'        */
%}
%union {
double     val;  /* For returning numbers.                   */
symrec  *tptr;   /* For returning symbol-table pointers      */
}
```


```
%token <val>  NUM        /* Simple double precision number   */
%token <tptr> VAR FNCT   /* Variable and Function            */
%type  <val>  exp

%right '='
%left '-' '+'
%left '*' '/'
%left NEG     /* Negation--unary minus */
%right '^'    /* Exponentiation        */

/* Grammar follows */

%%
```

The above grammar introduces only two new features of the Bison language. These features allow semantic values to have various data types (see section More Than One Value Type).

The `%union` declaration specifies the entire list of possible types; this is instead of defining `YYSTYPE`. The allowable types are now double-floats (for `exp` and `NUM`) and pointers to entries in the symbol table. See section The Collection of Value Types.

Since values can now have various types, it is necessary to associate a type with each grammar symbol whose semantic value is used. These symbols are `NUM`, `VAR`, `FNCT`, and `exp`. Their declarations are augmented with information about their data type (placed between angle brackets).

The Bison construct `%type` is used for declaring nonterminal symbols, just as `%token` is used for declaring token types. We have not used `%type` before because nonterminal symbols are normally declared implicitly by the rules that define them. But `exp` must be declared explicitly so we can specify its value type. See section Nonterminal Symbols.

Grammar Rules for `mfcalc`

Here are the grammar rules for the multi-function calculator. Most of them are copied directly from `calc`; three rules, those which mention `VAR` or `FNCT`, are new.

```
input:   /* empty */
        | input line
;

line:
          '\n'
        | exp '\n'   { printf ("\t%.10g\n", $1); }
        | error '\n' { yyerrok;                  }
;
```


```
exp:      NUM                { $$ = $1;                         }
        | VAR                { $$ = $1->value.var;              }
        | VAR '=' exp        { $$ = $3; $1->value.var = $3;     }
        | FNCT '(' exp ')'   { $$ = (*($1->value.fnctptr))($3); }
        | exp '+' exp        { $$ = $1 + $3;                    }
        | exp '-' exp        { $$ = $1 - $3;                    }
        | exp '*' exp        { $$ = $1 * $3;                    }
        | exp '/' exp        { $$ = $1 / $3;                    }
        | '-' exp  %prec NEG { $$ = -$2;                        }
        | exp '^' exp        { $$ = pow ($1, $3);               }
        | '(' exp ')'        { $$ = $2;                         }
;
/* End of grammar */
%%
```

The `mfcalc` Symbol Table

The multi-function calculator requires a symbol table to keep track of the names and meanings of variables and functions. This doesn't affect the grammar rules (except for the actions) or the Bison declarations, but it requires some additional C functions for support.

The symbol table itself consists of a linked list of records. Its definition, which is kept in the header `` `calc.h' ``, is as follows. It provides for either functions or variables to be placed in the table.

```
/* Data type for links in the chain of symbols.      */
struct symrec
{
  char *name;  /* name of symbol                     */
  int type;    /* type of symbol: either VAR or FNCT */
  union {
    double var;           /* value of a VAR         */
    double (*fnctptr)();  /* value of a FNCT        */
  } value;
  struct symrec *next;    /* link field             */
};

typedef struct symrec symrec;

/* The symbol table: a chain of `struct symrec'.     */
extern symrec *sym_table;

symrec *putsym ();
symrec *getsym ();
```

The new version of `main` includes a call to `init_table`, a function that initializes the symbol table. Here it is, and `init_table` as well:


```
#include <stdio.h>

main ()
{
  init_table ();
  yyparse ();
}

yyerror (s)  /* Called by yyparse on error */
     char *s;
{
  printf ("%s\n", s);
}

struct init
{
  char *fname;
  double (*fnct)();
};

struct init arith_fncts[]
  = {
      "sin", sin,
      "cos", cos,
      "atan", atan,
      "ln", log,
      "exp", exp,
      "sqrt", sqrt,
      0, 0
    };

/* The symbol table: a chain of `struct symrec'.  */
symrec *sym_table = (symrec *)0;

init_table ()  /* puts arithmetic functions in table. */
{
  int i;
  symrec *ptr;
  for (i = 0; arith_fncts[i].fname != 0; i++)
    {
      ptr = putsym (arith_fncts[i].fname, FNCT);
      ptr->value.fctptr = arith_fncts[i].fnct;
    }
}
```

By simply editing the initialization list and adding the necessary include files, you can add additional functions to the calculator.

Two important functions allow look-up and installation of symbols in the symbol table. The function `putsym` is passed a name and the type (VAR or FNCT) of the object to be installed. The object is linked to the front of the list, and a pointer to the object is returned. The function `getsym` is passed the name of the symbol to look up. If found, a pointer to that symbol is returned; otherwise zero is returned.

```
symrec *
putsym (sym_name,sym_type)
     char *sym_name;
     int sym_type;
{
  symrec *ptr;
  ptr = (symrec *) malloc (sizeof (symrec));
  ptr->name = (char *) malloc (strlen (sym_name) + 1);
  strcpy (ptr->name,sym_name);
  ptr->type = sym_type;
  ptr->value.var = 0; /* set value to 0 even if fctn.  */
  ptr->next = (struct symrec *)sym_table;
  sym_table = ptr;
  return ptr;
}

symrec *
getsym (sym_name)
     char *sym_name;
{
  symrec *ptr;
  for (ptr = sym_table; ptr != (symrec *) 0;
       ptr = (symrec *)ptr->next)
    if (strcmp (ptr->name,sym_name) == 0)
      return ptr;
  return 0;
}
```

The function `yylex` must now recognize variables, numeric values, and the single-character arithmetic operators. Strings of alphanumeric characters with a leading nondigit are recognized as either variables or functions depending on what the symbol table says about them.

The string is passed to `getsym` for look up in the symbol table. If the name appears in the table, a pointer to its location and its type (VAR or FNCT) is returned to `yyparse`. If it is not already in the table, then it is installed as a VAR using `putsym`. Again, a pointer and its type (which must be VAR) is returned to `yyparse`.

No change is needed in the handling of numeric values and arithmetic operators in `yylex`.


```
#include <ctype.h>
yylex ()
{
  int c;

  /* Ignore whitespace, get first nonwhite character.  */
  while ((c = getchar ()) == ' ' || c == '\t');

  if (c == EOF)
    return 0;

  /* Char starts a number => parse the number.         */
  if (c == '.' || isdigit (c))
    {
      ungetc (c, stdin);
      scanf ("%lf", &yylval.val);
      return NUM;
    }

  /* Char starts an identifier => read the name.       */
  if (isalpha (c))
    {
      symrec *s;
      static char *symbuf = 0;
      static int length = 0;
      int i;

      /* Initially make the buffer long enough
         for a 40-character symbol name.  */
      if (length == 0)
        length = 40, symbuf = (char *)malloc (length + 1);

      i = 0;
      do
        {
          /* If buffer is full, make it bigger.        */
          if (i == length)
            {
              length *= 2;
              symbuf = (char *)realloc (symbuf, length + 1);
            }
          /* Add this character to the buffer.         */
          symbuf[i++] = c;
          /* Get another character.                    */
          c = getchar ();
```


```
        }
      while (c != EOF && isalnum (c));

      ungetc (c, stdin);
      symbuf[i] = '\0';

      s = getsym (symbuf);
      if (s == 0)
        s = putsym (symbuf, VAR);
      yylval.tptr = s;
      return s->type;
    }

  /* Any other character is a token by itself.        */
  return c;
}
```

This program is both powerful and flexible. You may easily add new functions, and it is a simple job to modify this code to install predefined variables such as `pi` or `e` as well.

Exercises

1. Add some new functions from `` `math.h' `` to the initialization list.
2. Add another array that contains constants and their values. Then modify `init_table` to add these constants to the symbol table. It will be easiest to give the constants type `VAR`.
3. Make the program report an error if the user refers to an uninitialized variable in any way except to store a value in it.

Go to the previous, next section.

Go to the previous, next section.

Bison Grammar Files

Bison takes as input a context-free grammar specification and produces a C-language function that recognizes correct instances of the grammar.

The Bison grammar input file conventionally has a name ending in `.y'.

Outline of a Bison Grammar

A Bison grammar file has four main sections, shown here with the appropriate delimiters:

```
%{
C declarations
%}

Bison declarations

%%
Grammar rules
%%

Additional C code
```

Comments enclosed in `/* ... */' may appear in any of the sections.

The C Declarations Section

The C declarations section contains macro definitions and declarations of functions and variables that are used in the actions in the grammar rules. These are copied to the beginning of the parser file so that they precede the definition of `yyparse`. You can use `#include' to get the declarations from a header file. If you don't need any C declarations, you may omit the `%{' and `%}' delimiters that bracket this section.

The Bison Declarations Section

The Bison declarations section contains declarations that define terminal and nonterminal symbols, specify precedence, and so on. In some simple grammars you may not need any declarations. See section Bison Declarations.

The Grammar Rules Section

The grammar rules section contains one or more Bison grammar rules, and nothing else. See section Syntax of Grammar Rules.

There must always be at least one grammar rule, and the first `%%' (which precedes the grammar rules) may never be omitted even if it is the first thing in the file.

The Additional C Code Section

The additional C code section is copied verbatim to the end of the parser file, just as the C declarations section is copied to the beginning. This is the most convenient place to put anything that you want to have in the parser file but which need not come before the definition of `yyparse`. For example, the definitions of `yylex` and `yyerror` often go here. See section Parser C-Language Interface.

If the last section is empty, you may omit the `%%' that separates it from the grammar rules.

The Bison parser itself contains many static variables whose names start with `yy' and many macros whose names start with `YY'. It is a good idea to avoid using any such names (except those documented in this manual) in the additional C code section of the grammar file.

Symbols, Terminal and Nonterminal

Symbols in Bison grammars represent the grammatical classifications of the language.

A terminal symbol (also known as a token type) represents a class of syntactically equivalent tokens. You use the symbol in grammar rules to mean that a token in that class is allowed. The symbol is represented in the Bison parser by a numeric code, and the `yylex` function returns a token type code to indicate what kind of token has been read. You don't need to know what the code value is; you can use the symbol to stand for it.

A nonterminal symbol stands for a class of syntactically equivalent groupings. The symbol name is used in writing grammar rules. By convention, it should be all lower case.

Symbol names can contain letters, digits (not at the beginning), underscores and periods. Periods make sense only in nonterminals.

There are three ways of writing terminal symbols in the grammar:

- A named token type is written with an identifier, like an identifier in C. By convention, it should be all upper case. Each such name must be defined with a Bison declaration such as `%token`. See section Token Type Names.
- A character token type (or literal character token) is written in the grammar using the same syntax used in C for character constants; for example, `'+'` is a character token type. A character token type doesn't need to be declared unless you need to specify its semantic value data type (see section Data Types of Semantic Values), associativity, or precedence (see section Operator Precedence).

By convention, a character token type is used only to represent a token that consists of that particular character. Thus, the token type `'+'` is used to represent the character `+' as a token. Nothing enforces this convention, but if you depart from it, your program will confuse other readers.

All the usual escape sequences used in character literals in C can be used in Bison as well, but you must not use the null character as a character literal because its ASCII code, zero, is the code `yylex` returns for end-of-input (see section Calling Convention for `yylex`}).

- A literal string token is written like a C string constant; for example, `"<="` is a literal string token. A literal string token doesn't need to be declared unless you need to specify its semantic value data type (see section Data Types of Semantic Values), associativity, precedence (see section Operator Precedence).

  You can associate the literal string token with a symbolic name as an alias, using the `%token` declaration (see section Token Type Names). If you don't do that, the lexical analyzer has to retrieve the token number for the literal string token from the `yytname` table (see section Calling Convention for `yylex`).

  WARNING: literal string tokens do not work in Yacc.

  By convention, a literal string token is used only to represent a token that consists of that particular string. Thus, you should use the token type `"<="` to represent the string `<=' as a token. Bison does not enforces this convention, but if you depart from it, people who read your program will be confused.

  All the escape sequences used in string literals in C can be used in Bison as well. A literal string token must contain two or more characters; for a token containing just one character, use a character token (see above).

How you choose to write a terminal symbol has no effect on its grammatical meaning. That depends only on where it appears in rules and on when the parser function returns that symbol.

The value returned by `yylex` is always one of the terminal symbols (or 0 for end-of-input). Whichever way you write the token type in the grammar rules, you write it the same way in the definition of `yylex`. The numeric code for a character token type is simply the ASCII code for the character, so `yylex` can use the identical character constant to generate the requisite code. Each named token type becomes a C macro in the parser file, so `yylex` can use the name to stand for the code. (This is why periods don't make sense in terminal symbols.) See section Calling Convention for `yylex`.

If `yylex` is defined in a separate file, you need to arrange for the token-type macro definitions to be available there. Use the `-d' option when you run Bison, so that it will write these macro definitions into a separate header file `` `name.tab.h' `` which you can include in the other source files that need it. See section Invoking Bison.

The symbol `error` is a terminal symbol reserved for error recovery (see section Error Recovery); you shouldn't use it for any other purpose. In particular, `yylex` should never return this value.

Syntax of Grammar Rules

A Bison grammar rule has the following general form:

```
result: components...
        ;
```

where result is the nonterminal symbol that this rule describes and components are various terminal and nonterminal symbols that are put together by this rule (see section Symbols, Terminal and Nonterminal).

For example,

```
exp:      exp '+' exp
        ;
```

says that two groupings of type `exp`, with a `+' token in between, can be combined into a larger grouping of type `exp`.

Whitespace in rules is significant only to separate symbols. You can add extra whitespace as you wish.

Scattered among the components can be actions that determine the semantics of the rule. An action looks like this:

```
{C statements}
```

Usually there is only one action and it follows the components. See section Actions.

Multiple rules for the same result can be written separately or can be joined with the vertical-bar character `|' as follows:

```
result:    rule1-components...
        | rule2-components...
        ...
        ;
```

They are still considered distinct rules even when joined in this way.

If components in a rule is empty, it means that result can match the empty string. For example, here is how to define a comma-separated sequence of zero or more `exp` groupings:

```
expseq:   /* empty */
        | expseq1
        ;

expseq1:  exp
        | expseq1 ',' exp
        ;
```

It is customary to write a comment `/* empty */' in each rule with no components.

Recursive Rules

A rule is called recursive when its result nonterminal appears also on its right hand side. Nearly all Bison grammars need to use recursion, because that is the only way to define a sequence of any number of somethings. Consider this recursive definition of a comma-separated sequence of one or more expressions:

```
expseq1:  exp
        | expseq1 ',' exp
        ;
```

Since the recursive use of `expseq1` is the leftmost symbol in the right hand side, we call this left recursion. By contrast, here the same construct is defined using right recursion:

```
expseq1:  exp
        | exp ',' expseq1
        ;
```

Any kind of sequence can be defined using either left recursion or right recursion, but you should always use left recursion, because it can parse a sequence of any number of elements with bounded stack space. Right recursion uses up space on the Bison stack in proportion to the number of elements in the sequence, because all the elements must be shifted onto the stack before the rule can be applied even once. See section The Bison Parser Algorithm, for further explanation of this.

Indirect or mutual recursion occurs when the result of the rule does not appear directly on its right hand side, but does appear in rules for other nonterminals which do appear on its right hand side.

For example:

```
expr:     primary
        | primary '+' primary
        ;

primary:  constant
        | '(' expr ')'
        ;
```

defines two mutually-recursive nonterminals, since each refers to the other.

Defining Language Semantics

The grammar rules for a language determine only the syntax. The semantics are determined by the semantic values associated with various tokens and groupings, and by the actions taken when various groupings are recognized.

For example, the calculator calculates properly because the value associated with each expression is the proper number; it adds properly because the action for the grouping `x + y' is to add the numbers associated with x and y.

Data Types of Semantic Values

In a simple program it may be sufficient to use the same data type for the semantic values of all language constructs. This was true in the RPN and infix calculator examples (see section Reverse Polish Notation Calculator).

Bison's default is to use type `int` for all semantic values. To specify some other type, define `YYSTYPE` as a macro, like this:

```
#define YYSTYPE double
```

This macro definition must go in the C declarations section of the grammar file (see section Outline of a Bison Grammar).

More Than One Value Type

In most programs, you will need different data types for different kinds of tokens and groupings. For example, a numeric constant may need type `int` or `long`, while a string constant needs type `char *`, and an identifier might need a pointer to an entry in the symbol table.

To use more than one data type for semantic values in one parser, Bison requires you to do two things:

- Specify the entire collection of possible data types, with the `%union` Bison declaration (see section The Collection of Value Types).
- Choose one of those types for each symbol (terminal or nonterminal) for which semantic values are used. This is done for tokens with the `%token` Bison declaration (see section Token Type Names) and for groupings with the `%type` Bison declaration (see section Nonterminal Symbols).

Actions

An action accompanies a syntactic rule and contains C code to be executed each time an instance of that rule is recognized. The task of most actions is to compute a semantic value for the grouping built by the rule from the semantic values associated with tokens or smaller groupings.

An action consists of C statements surrounded by braces, much like a compound statement in C. It can be placed at any position in the rule; it is executed at that position. Most rules have just one action at the end

of the rule, following all the components. Actions in the middle of a rule are tricky and used only for special purposes (see section Actions in Mid-Rule).

The C code in an action can refer to the semantic values of the components matched by the rule with the construct `$n`, which stands for the value of the nth component. The semantic value for the grouping being constructed is `$$`. (Bison translates both of these constructs into array element references when it copies the actions into the parser file.)

Here is a typical example:

```
exp:    ...
        | exp '+' exp
            { $$ = $1 + $3; }
```

This rule constructs an `exp` from two smaller `exp` groupings connected by a plus-sign token. In the action, `$1` and `$3` refer to the semantic values of the two component `exp` groupings, which are the first and third symbols on the right hand side of the rule. The sum is stored into `$$` so that it becomes the semantic value of the addition-expression just recognized by the rule. If there were a useful semantic value associated with the `+' token, it could be referred to as `$2`.

If you don't specify an action for a rule, Bison supplies a default: `$$ = $1`. Thus, the value of the first symbol in the rule becomes the value of the whole rule. Of course, the default rule is valid only if the two data types match. There is no meaningful default action for an empty rule; every empty rule must have an explicit action unless the rule's value does not matter.

`$n` with n zero or negative is allowed for reference to tokens and groupings on the stack *before* those that match the current rule. This is a very risky practice, and to use it reliably you must be certain of the context in which the rule is applied. Here is a case in which you can use this reliably:

```
foo:      expr bar '+' expr  { ... }
        | expr bar '-' expr  { ... }
        ;

bar:      /* empty */
        { previous_expr = $0; }
        ;
```

As long as `bar` is used only in the fashion shown here, `$0` always refers to the `expr` which precedes `bar` in the definition of `foo`.

Data Types of Values in Actions

If you have chosen a single data type for semantic values, the `$$` and `$n` constructs always have that data type.

If you have used `%union` to specify a variety of data types, then you must declare a choice among these types for each terminal or nonterminal symbol that can have a semantic value. Then each time you use

$$ or $n, its data type is determined by which symbol it refers to in the rule. In this example,

```
exp:    ...
        | exp '+' exp
            { $$ = $1 + $3; }
```

$1 and $3 refer to instances of exp, so they all have the data type declared for the nonterminal symbol exp. If $2 were used, it would have the data type declared for the terminal symbol '+', whatever that might be.

Alternatively, you can specify the data type when you refer to the value, by inserting `<type>' after the `$' at the beginning of the reference. For example, if you have defined types as shown here:

```
%union {
  int itype;
  double dtype;
}
```

then you can write $<itype>1 to refer to the first subunit of the rule as an integer, or $<dtype>1 to refer to it as a double.

Actions in Mid-Rule

Occasionally it is useful to put an action in the middle of a rule. These actions are written just like usual end-of-rule actions, but they are executed before the parser even recognizes the following components.

A mid-rule action may refer to the components preceding it using $n, but it may not refer to subsequent components because it is run before they are parsed.

The mid-rule action itself counts as one of the components of the rule. This makes a difference when there is another action later in the same rule (and usually there is another at the end): you have to count the actions along with the symbols when working out which number n to use in $n.

The mid-rule action can also have a semantic value. The action can set its value with an assignment to $$, and actions later in the rule can refer to the value using $n. Since there is no symbol to name the action, there is no way to declare a data type for the value in advance, so you must use the `$<...>' construct to specify a data type each time you refer to this value.

There is no way to set the value of the entire rule with a mid-rule action, because assignments to $$ do not have that effect. The only way to set the value for the entire rule is with an ordinary action at the end of the rule.

Here is an example from a hypothetical compiler, handling a let statement that looks like `let (variable) statement' and serves to create a variable named variable temporarily for the duration of statement. To parse this construct, we must put variable into the symbol table while statement is parsed, then remove it afterward. Here is how it is done:

```
stmt:   LET '(' var ')'
```


```
                { $<context>$ = push_context ();
                  declare_variable ($3); }
        stmt    { $$ = $6;
                  pop_context ($<context>5); }
```

As soon as `let (variable)' has been recognized, the first action is run. It saves a copy of the current semantic context (the list of accessible variables) as its semantic value, using alternative `context` in the data-type union. Then it calls `declare_variable` to add the new variable to that list. Once the first action is finished, the embedded statement `stmt` can be parsed. Note that the mid-rule action is component number 5, so the `stmt' is component number 6.

After the embedded statement is parsed, its semantic value becomes the value of the entire `let`-statement. Then the semantic value from the earlier action is used to restore the prior list of variables. This removes the temporary `let`-variable from the list so that it won't appear to exist while the rest of the program is parsed.

Taking action before a rule is completely recognized often leads to conflicts since the parser must commit to a parse in order to execute the action. For example, the following two rules, without mid-rule actions, can coexist in a working parser because the parser can shift the open-brace token and look at what follows before deciding whether there is a declaration or not:

```
compound: '{' declarations statements '}'
        | '{' statements '}'
        ;
```

But when we add a mid-rule action as follows, the rules become nonfunctional:

```
compound: { prepare_for_local_variables (); }
          '{' declarations statements '}'
        | '{' statements '}'
        ;
```

Now the parser is forced to decide whether to run the mid-rule action when it has read no farther than the open-brace. In other words, it must commit to using one rule or the other, without sufficient information to do it correctly. (The open-brace token is what is called the look-ahead token at this time, since the parser is still deciding what to do about it. See section Look-Ahead Tokens.)

You might think that you could correct the problem by putting identical actions into the two rules, like this:

```
compound: { prepare_for_local_variables (); }
          '{' declarations statements '}'
        | { prepare_for_local_variables (); }
          '{' statements '}'
        ;
```

But this does not help, because Bison does not realize that the two actions are identical. (Bison never

tries to understand the C code in an action.)

If the grammar is such that a declaration can be distinguished from a statement by the first token (which is true in C), then one solution which does work is to put the action after the open-brace, like this:

```
compound: '{' { prepare_for_local_variables (); }
          declarations statements '}'
        | '{' statements '}'
        ;
```

Now the first token of the following declaration or statement, which would in any case tell Bison which rule to use, can still do so.

Another solution is to bury the action inside a nonterminal symbol which serves as a subroutine:

```
subroutine: /* empty */
          { prepare_for_local_variables (); }
        ;


compound: subroutine
          '{' declarations statements '}'
        | subroutine
          '{' statements '}'
        ;
```

Now Bison can execute the action in the rule for `subroutine` without deciding which rule for `compound` it will eventually use. Note that the action is now at the end of its rule. Any mid-rule action can be converted to an end-of-rule action in this way, and this is what Bison actually does to implement mid-rule actions.

Bison Declarations

The Bison declarations section of a Bison grammar defines the symbols used in formulating the grammar and the data types of semantic values. See section Symbols, Terminal and Nonterminal.

All token type names (but not single-character literal tokens such as `'+'` and `'*'`) must be declared. Nonterminal symbols must be declared if you need to specify which data type to use for the semantic value (see section More Than One Value Type).

The first rule in the file also specifies the start symbol, by default. If you want some other symbol to be the start symbol, you must declare it explicitly (see section Languages and Context-Free Grammars).

Token Type Names

The basic way to declare a token type name (terminal symbol) is as follows:

```
%token name
```

Bison will convert this into a `#define` directive in the parser, so that the function `yylex` (if it is in this file) can use the name name to stand for this token type's code.

Alternatively, you can use `%left`, `%right`, or `%nonassoc` instead of `%token`, if you wish to specify precedence. See section Operator Precedence.

You can explicitly specify the numeric code for a token type by appending an integer value in the field immediately following the token name:

```
%token NUM 300
```

It is generally best, however, to let Bison choose the numeric codes for all token types. Bison will automatically select codes that don't conflict with each other or with ASCII characters.

In the event that the stack type is a union, you must augment the `%token` or other token declaration to include the data type alternative delimited by angle-brackets (see section More Than One Value Type).

For example:

```
%union {              /* define stack type */
  double val;
  symrec *tptr;
}
%token <val> NUM      /* define token NUM and its type */
```

You can associate a literal string token with a token type name by writing the literal string at the end of a `%token` declaration which declares the name. For example:

```
%token arrow "=>"
```

For example, a grammar for the C language might specify these names with equivalent literal string tokens:

```
%token  <operator>  OR      "||"
%token  <operator>  LE 134  "<="
%left  OR  "<="
```

Once you equate the literal string and the token name, you can use them interchangeably in further declarations or the grammar rules. The `yylex` function can use the token name or the literal string to obtain the token type code number (see section Calling Convention for `yylex`).

Operator Precedence

Use the `%left`, `%right` or `%nonassoc` declaration to declare a token and specify its precedence and associativity, all at once. These are called precedence declarations. See section Operator Precedence, for general information on operator precedence.

The syntax of a precedence declaration is the same as that of `%token`: either

```
%left symbols...
```

or

```
%left <type> symbols...
```

And indeed any of these declarations serves the purposes of `%token`. But in addition, they specify the associativity and relative precedence for all the symbols:

- The associativity of an operator op determines how repeated uses of the operator nest: whether `x op y op z' is parsed by grouping x with y first or by grouping y with z first. `%left` specifies left-associativity (grouping x with y first) and `%right` specifies right-associativity (grouping y with z first). `%nonassoc` specifies no associativity, which means that `x op y op z' is considered a syntax error.
- The precedence of an operator determines how it nests with other operators. All the tokens declared in a single precedence declaration have equal precedence and nest together according to their associativity. When two tokens declared in different precedence declarations associate, the one declared later has the higher precedence and is grouped first.

The Collection of Value Types

The `%union` declaration specifies the entire collection of possible data types for semantic values. The keyword `%union` is followed by a pair of braces containing the same thing that goes inside a `union` in C.

For example:

```
%union {
  double val;
  symrec *tptr;
}
```

This says that the two alternative types are `double` and `symrec *`. They are given names `val` and `tptr`; these names are used in the `%token` and `%type` declarations to pick one of the types for a terminal or nonterminal symbol (see section Nonterminal Symbols).

Note that, unlike making a `union` declaration in C, you do not write a semicolon after the closing brace.

Nonterminal Symbols

When you use `%union` to specify multiple value types, you must declare the value type of each nonterminal symbol for which values are used. This is done with a `%type` declaration, like this:

```
%type <type> nonterminal...
```

Here nonterminal is the name of a nonterminal symbol, and type is the name given in the `%union` to the alternative that you want (see section The Collection of Value Types). You can give any number of nonterminal symbols in the same `%type` declaration, if they have the same value type. Use spaces to separate the symbol names.

You can also declare the value type of a terminal symbol. To do this, use the same `<type>` construction in a declaration for the terminal symbol. All kinds of token declarations allow `<type>`.

Suppressing Conflict Warnings

Bison normally warns if there are any conflicts in the grammar (see section Shift/Reduce Conflicts), but most real grammars have harmless shift/reduce conflicts which are resolved in a predictable way and would be difficult to eliminate. It is desirable to suppress the warning about these conflicts unless the number of conflicts changes. You can do this with the `%expect` declaration.

The declaration looks like this:

```
%expect n
```

Here n is a decimal integer. The declaration says there should be no warning if there are n shift/reduce conflicts and no reduce/reduce conflicts. The usual warning is given if there are either more or fewer conflicts, or if there are any reduce/reduce conflicts.

In general, using `%expect` involves these steps:

- Compile your grammar without `%expect`. Use the `-v' option to get a verbose list of where the conflicts occur. Bison will also print the number of conflicts.
- Check each of the conflicts to make sure that Bison's default resolution is what you really want. If not, rewrite the grammar and go back to the beginning.
- Add an `%expect` declaration, copying the number n from the number which Bison printed.

Now Bison will stop annoying you about the conflicts you have checked, but it will warn you again if changes in the grammar result in additional conflicts.

The Start-Symbol

Bison assumes by default that the start symbol for the grammar is the first nonterminal specified in the grammar specification section. The programmer may override this restriction with the `%start` declaration as follows:


```
%start symbol
```

A Pure (Reentrant) Parser

A reentrant program is one which does not alter in the course of execution; in other words, it consists entirely of pure (read-only) code. Reentrancy is important whenever asynchronous execution is possible; for example, a nonreentrant program may not be safe to call from a signal handler. In systems with multiple threads of control, a nonreentrant program must be called only within interlocks.

The Bison parser is not normally a reentrant program, because it uses statically allocated variables for communication with `yylex`. These variables include `yylval` and `yylloc`.

The Bison declaration `%pure_parser` says that you want the parser to be reentrant. It looks like this:

```
%pure_parser
```

The effect is that the two communication variables become local variables in `yyparse`, and a different calling convention is used for the lexical analyzer function `yylex`. See section Calling Conventions for Pure Parsers, for the details of this. The variable `yynerrs` also becomes local in `yyparse` (see section The Error Reporting Function `yyerror`). The convention for calling `yyparse` itself is unchanged.

Bison Declaration Summary

Here is a summary of all Bison declarations:

`%union`

> Declare the collection of data types that semantic values may have (see section The Collection of Value Types).

`%token`

> Declare a terminal symbol (token type name) with no precedence or associativity specified (see section Token Type Names).

`%right`

> Declare a terminal symbol (token type name) that is right-associative (see section Operator Precedence).

`%left`

> Declare a terminal symbol (token type name) that is left-associative (see section Operator Precedence).

`%nonassoc`

> Declare a terminal symbol (token type name) that is nonassociative (using it in a way that would be associative is a syntax error) (see section Operator Precedence).

`%type`

> Declare the type of semantic values for a nonterminal symbol (see section Nonterminal Symbols).

`%start`

Specify the grammar's start symbol (see section The Start-Symbol).

`%expect`

Declare the expected number of shift-reduce conflicts (see section Suppressing Conflict Warnings).

`%pure_parser`

Request a pure (reentrant) parser program (see section A Pure (Reentrant) Parser).

`%no_lines`

Don't generate any `#line` preprocessor commands in the parser file. Ordinarily Bison writes these commands in the parser file so that the C compiler and debuggers will associate errors and object code with your source file (the grammar file). This directive causes them to associate errors with the parser file, treating it an independent source file in its own right.

`%raw`

The output file `` `name.h' `` normally defines the tokens with Yacc-compatible token numbers. If this option is specified, the internal Bison numbers are used instead. (Yacc-compatible numbers start at 257 except for single character tokens; Bison assigns token numbers sequentially for all tokens starting at 3.)

`%token_table`

Generate an array of token names in the parser file. The name of the array is `yytname`; `yytname[i]` is the name of the token whose internal Bison token code number is i. The first three elements of `yytname` are always `"$"`, `"error"`, and `"$illegal"`; after these come the symbols defined in the grammar file.

For single-character literal tokens and literal string tokens, the name in the table includes the single-quote or double-quote characters: for example, `"'+'"` is a single-character literal and `"\"<=\""` is a literal string token. All the characters of the literal string token appear verbatim in the string found in the table; even double-quote characters are not escaped. For example, if the token consists of three characters `*"*', its string in `yytname` contains `"*"*"'. (In C, that would be written as `"\"*\"*\""`).

When you specify `%token_table`, Bison also generates macro definitions for macros `YYNTOKENS`, `YYNNTS`, and `YYNRULES`, and `YYNSTATES`:

`YYNTOKENS`

The highest token number, plus one.

`YYNNTS`

The number of non-terminal symbols.

`YYNRULES`

The number of grammar rules,

`YYNSTATES`

The number of parser states (see section Parser States).

Multiple Parsers in the Same Program

Most programs that use Bison parse only one language and therefore contain only one Bison parser. But what if you want to parse more than one language with the same program? Then you need to avoid a name conflict between different definitions of `yyparse`, `yylval`, and so on.

The easy way to do this is to use the option `-p prefix' (see section Invoking Bison). This renames the interface functions and variables of the Bison parser to start with prefix instead of `yy'. You can use this to give each parser distinct names that do not conflict.

The precise list of symbols renamed is `yyparse`, `yylex`, `yyerror`, `yynerrs`, `yylval`, `yychar` and `yydebug`. For example, if you use `-p c', the names become `cparse`, `clex`, and so on.

All the other variables and macros associated with Bison are not renamed. These others are not global; there is no conflict if the same name is used in different parsers. For example, YYSTYPE is not renamed, but defining this in different ways in different parsers causes no trouble (see section Data Types of Semantic Values).

The `-p' option works by adding macro definitions to the beginning of the parser source file, defining `yyparse` as `prefixparse`, and so on. This effectively substitutes one name for the other in the entire parser file.

Go to the previous, next section.

Go to the previous, next section.

Parser C-Language Interface

The Bison parser is actually a C function named `yyparse`. Here we describe the interface conventions of `yyparse` and the other functions that it needs to use.

Keep in mind that the parser uses many C identifiers starting with `yy' and `YY' for internal purposes. If you use such an identifier (aside from those in this manual) in an action or in additional C code in the grammar file, you are likely to run into trouble.

The Parser Function `yyparse`

You call the function `yyparse` to cause parsing to occur. This function reads tokens, executes actions, and ultimately returns when it encounters end-of-input or an unrecoverable syntax error. You can also write an action which directs `yyparse` to return immediately without reading further.

The value returned by `yyparse` is 0 if parsing was successful (return is due to end-of-input).

The value is 1 if parsing failed (return is due to a syntax error).

In an action, you can cause immediate return from `yyparse` by using these macros:

`YYACCEPT`

Return immediately with value 0 (to report success).

`YYABORT`

Return immediately with value 1 (to report failure).

The Lexical Analyzer Function `yylex`

The lexical analyzer function, `yylex`, recognizes tokens from the input stream and returns them to the parser. Bison does not create this function automatically; you must write it so that `yyparse` can call it. The function is sometimes referred to as a lexical scanner.

In simple programs, `yylex` is often defined at the end of the Bison grammar file. If `yylex` is defined in a separate source file, you need to arrange for the token-type macro definitions to be available there. To do this, use the `-d' option when you run Bison, so that it will write these macro definitions into a separate header file `name.tab.h' which you can include in the other source files that need it. See section Invoking Bison.

Calling Convention for `yylex`

The value that `yylex` returns must be the numeric code for the type of token it has just found, or 0 for end-of-input.

When a token is referred to in the grammar rules by a name, that name in the parser file becomes a C macro whose definition is the proper numeric code for that token type. So `yylex` can use the name to indicate that type. See section Symbols, Terminal and Nonterminal.

When a token is referred to in the grammar rules by a character literal, the numeric code for that character is the code for the token type. So `yylex` can simply return that character code. The null character must not be this way, because its code is zero and that is what signifies end-of-input.

Here is an example showing these things:

```
yylex ()
{
  ...
  if (c == EOF)     /* Detect end of file. */
    return 0;
  ...
  if (c == '+' || c == '-')
    return c;      /* Assume token type for `+' is '+'. */
  ...
  return INT;      /* Return the type of the token. */
  ...
}
```

This interface has been designed so that the output from the `lex` utility can be used without change as the definition of `yylex`.

If the grammar uses literal string tokens, there are two ways that `yylex` can determine the token type codes for them:

- If the grammar defines symbolic token names as aliases for the literal string tokens, `yylex` can use these symbolic names like all others. In this case, the use of the literal string tokens in the grammar file has no effect on `yylex`.
- `yylex` can find the multi-character token in the `yytname` table. The index of the token in the table is the token type's code. The name of a multi-character token is recorded in `yytname` with a double-quote, the token's characters, and another double-quote. The token's characters are not escaped in any way; they appear verbatim in the contents of the string in the table.

  Here's code for looking up a token in `yytname`, assuming that the characters of the token are stored in `token_buffer`.

  ```
  for (i = 0; i < YYNTOKENS; i++)
    {
      if (yytname[i] != 0
          && yytname[i][0] == '"'
          && strncmp (yytname[i] + 1, token_buffer, strlen (token_buffer))
          && yytname[i][strlen (token_buffer) + 1] == '"'
          && yytname[i][strlen (token_buffer) + 2] == 0)
        break;
    }
  ```

  The `yytname` table is generated only if you use the `%token_table` declaration. See section Bison Declaration Summary.

Semantic Values of Tokens

In an ordinary (nonreentrant) parser, the semantic value of the token must be stored into the global variable `yylval`. When you are using just one data type for semantic values, `yylval` has that type. Thus, if the type is `int` (the default), you might write this in `yylex`:

```
  ...
  yylval = value;  /* Put value onto Bison stack. */
  return INT;      /* Return the type of the token. */
  ...
```

When you are using multiple data types, `yylval`'s type is a union made from the `%union` declaration (see section The Collection of Value Types). So when you store a token's value, you must use the proper member of the union. If the `%union` declaration looks like this:

```
%union {
  int intval;
  double val;
  symrec *tptr;
}
```

then the code in `yylex` might look like this:

```
  ...
  yylval.intval = value; /* Put value onto Bison stack. */
  return INT;          /* Return the type of the token. */
  ...
```

Textual Positions of Tokens

If you are using the `@n'-feature (see section Special Features for Use in Actions) in actions to keep track of the textual locations of tokens and groupings, then you must provide this information in `yylex`. The function `yyparse` expects to find the textual location of a token just parsed in the global variable `yylloc`. So `yylex` must store the proper data in that variable. The value of `yylloc` is a structure and you need only initialize the members that are going to be used by the actions. The four members are called `first_line`, `first_column`, `last_line` and `last_column`. Note that the use of this feature makes the parser noticeably slower.

The data type of `yylloc` has the name `YYLTYPE`.

Calling Conventions for Pure Parsers

When you use the Bison declaration `%pure_parser` to request a pure, reentrant parser, the global communication variables `yylval` and `yylloc` cannot be used. (See section A Pure (Reentrant) Parser.) In such parsers the two global variables are replaced by pointers passed as arguments to `yylex`. You must declare them as shown here, and pass the information back by storing it through those pointers.

```
yylex (lvalp, llocp)
     YYSTYPE *lvalp;
```


```
     YYLTYPE *llocp;
{
  ...
  *lvalp = value;  /* Put value onto Bison stack.  */
  return INT;      /* Return the type of the token.  */
  ...
}
```

If the grammar file does not use the `@' constructs to refer to textual positions, then the type `YYLTYPE' will not be defined. In this case, omit the second argument; `yylex' will be called with only one argument.

If you use a reentrant parser, you can optionally pass additional parameter information to it in a reentrant way. To do so, define the macro `YYPARSE_PARAM' as a variable name. This modifies the `yyparse' function to accept one argument, of type `void *', with that name.

When you call `yyparse', pass the address of an object, casting the address to `void *'. The grammar actions can refer to the contents of the object by casting the pointer value back to its proper type and then dereferencing it. Here's an example. Write this in the parser:

```
%{
struct parser_control
{
  int nastiness;
  int randomness;
};

#define YYPARSE_PARAM parm
%}
```

Then call the parser like this:

```
struct parser_control
{
  int nastiness;
  int randomness;
};

...

{
  struct parser_control foo;
  ...  /* Store proper data in foo.  */
  value = yyparse ((void *) &foo);
  ...
}
```

In the grammar actions, use expressions like this to refer to the data:

```
((struct parser_control *) parm)->randomness
```

If you wish to pass the additional parameter data to `yylex', define the macro `YYLEX_PARAM' just like

YYPARSE_PARAM, as shown here:

```
%{
struct parser_control
{
  int nastiness;
  int randomness;
};

#define YYPARSE_PARAM parm
#define YYLEX_PARAM parm
%}
```

You should then define `yylex` to accept one additional argument--the value of `parm`. (This makes either two or three arguments in total, depending on whether an argument of type YYLTYPE is passed.) You can declare the argument as a pointer to the proper object type, or you can declare it as `void *` and access the contents as shown above.

You can use `%pure_parser' to request a reentrant parser without also using YYPARSE_PARAM. Then you should call `yyparse` with no arguments, as usual.

The Error Reporting Function `yyerror`

The Bison parser detects a parse error or syntax error whenever it reads a token which cannot satisfy any syntax rule. A action in the grammar can also explicitly proclaim an error, using the macro YYERROR (see section Special Features for Use in Actions).

The Bison parser expects to report the error by calling an error reporting function named `yyerror`, which you must supply. It is called by `yyparse` whenever a syntax error is found, and it receives one argument. For a parse error, the string is normally `"parse error"`.

If you define the macro YYERROR_VERBOSE in the Bison declarations section (see section The Bison Declarations Section), then Bison provides a more verbose and specific error message string instead of just plain `"parse error"`. It doesn't matter what definition you use for YYERROR_VERBOSE, just whether you define it.

The parser can detect one other kind of error: stack overflow. This happens when the input contains constructions that are very deeply nested. It isn't likely you will encounter this, since the Bison parser extends its stack automatically up to a very large limit. But if overflow happens, `yyparse` calls `yyerror` in the usual fashion, except that the argument string is `"parser stack overflow"`.

The following definition suffices in simple programs:

```
yyerror (s)
     char *s;
{
  fprintf (stderr, "%s\n", s);
}
```

After `yyerror` returns to `yyparse`, the latter will attempt error recovery if you have written suitable error

recovery grammar rules (see section Error Recovery). If recovery is impossible, `yyparse` will immediately return 1.

The variable `yynerrs` contains the number of syntax errors encountered so far. Normally this variable is global; but if you request a pure parser (see section A Pure (Reentrant) Parser) then it is a local variable which only the actions can access.

Special Features for Use in Actions

Here is a table of Bison constructs, variables and macros that are useful in actions.

`$$'

Acts like a variable that contains the semantic value for the grouping made by the current rule. See section Actions.

`$n'

Acts like a variable that contains the semantic value for the nth component of the current rule. See section Actions.

`$<typealt>$'

Like `$$` but specifies alternative typealt in the union specified by the `%union` declaration. See section Data Types of Values in Actions.

`$<typealt>n'

Like `$n` but specifies alternative typealt in the union specified by the `%union` declaration. See section Data Types of Values in Actions.

`YYABORT;'

Return immediately from `yyparse`, indicating failure. See section The Parser Function `yyparse`.

`YYACCEPT;'

Return immediately from `yyparse`, indicating success. See section The Parser Function `yyparse`.

`YYBACKUP (token, value);'

Unshift a token. This macro is allowed only for rules that reduce a single value, and only when there is no look-ahead token. It installs a look-ahead token with token type token and semantic value value; then it discards the value that was going to be reduced by this rule.

If the macro is used when it is not valid, such as when there is a look-ahead token already, then it reports a syntax error with a message `cannot back up' and performs ordinary error recovery.

In either case, the rest of the action is not executed.

`YYEMPTY'

Value stored in `yychar` when there is no look-ahead token.

`YYERROR;'

Cause an immediate syntax error. This statement initiates error recovery just as if the parser itself had detected an error; however, it does not call `yyerror`, and does not print any message. If you want to print an error message, call `yyerror` explicitly before the `YYERROR;' statement. See section Error Recovery.

`YYRECOVERING'

This macro stands for an expression that has the value 1 when the parser is recovering from a syntax error,

and 0 the rest of the time. See section Error Recovery.

`yychar'

Variable containing the current look-ahead token. (In a pure parser, this is actually a local variable within `yyparse`.) When there is no look-ahead token, the value `YYEMPTY` is stored in the variable. See section Look-Ahead Tokens.

`yyclearin;'

Discard the current look-ahead token. This is useful primarily in error rules. See section Error Recovery.

`yyerrok;'

Resume generating error messages immediately for subsequent syntax errors. This is useful primarily in error rules. See section Error Recovery.

`@n'

Acts like a structure variable containing information on the line numbers and column numbers of the nth component of the current rule. The structure has four members, like this:

```
struct {
  int first_line, last_line;
  int first_column, last_column;
};
```

Thus, to get the starting line number of the third component, use `@3.first_line'.

In order for the members of this structure to contain valid information, you must make `yylex` supply this information about each token. If you need only certain members, then `yylex` need only fill in those members.

The use of this feature makes the parser noticeably slower.

Go to the previous, next section.

Go to the previous, next section.

The Bison Parser Algorithm

As Bison reads tokens, it pushes them onto a stack along with their semantic values. The stack is called the parser stack. Pushing a token is traditionally called shifting.

For example, suppose the infix calculator has read `1 + 5 *', with a `3' to come. The stack will have four elements, one for each token that was shifted.

But the stack does not always have an element for each token read. When the last n tokens and groupings shifted match the components of a grammar rule, they can be combined according to that rule. This is called reduction. Those tokens and groupings are replaced on the stack by a single grouping whose symbol is the result (left hand side) of that rule. Running the rule's action is part of the process of reduction, because this is what computes the semantic value of the resulting grouping.

For example, if the infix calculator's parser stack contains this:

```
1 + 5 * 3
```

and the next input token is a newline character, then the last three elements can be reduced to 15 via the rule:

```
expr: expr '*' expr;
```

Then the stack contains just these three elements:

```
1 + 15
```

At this point, another reduction can be made, resulting in the single value 16. Then the newline token can be shifted.

The parser tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol (see section Languages and Context-Free Grammars).

This kind of parser is known in the literature as a bottom-up parser.

Look-Ahead Tokens

The Bison parser does *not* always reduce immediately as soon as the last n tokens and groupings match a rule. This is because such a simple strategy is inadequate to handle most languages. Instead, when a reduction is possible, the parser sometimes "looks ahead" at the next token in order to decide what to do.

When a token is read, it is not immediately shifted; first it becomes the look-ahead token, which is not on the stack. Now the parser can perform one or more reductions of tokens and groupings on the stack, while the look-ahead token remains off to the side. When no more reductions should take place, the

look-ahead token is shifted onto the stack. This does not mean that all possible reductions have been done; depending on the token type of the look-ahead token, some rules may choose to delay their application.

Here is a simple case where look-ahead is needed. These three rules define expressions which contain binary addition operators and postfix unary factorial operators (`!'), and allow parentheses for grouping.

```
expr:     term '+' expr
        | term
        ;

term:     '(' expr ')'
        | term '!'
        | NUMBER
        ;
```

Suppose that the tokens `1 + 2' have been read and shifted; what should be done? If the following token is `)', then the first three tokens must be reduced to form an `expr`. This is the only valid course, because shifting the `)' would produce a sequence of symbols `term ')'`, and no rule allows this.

If the following token is `!', then it must be shifted immediately so that `2 !' can be reduced to make a `term`. If instead the parser were to reduce before shifting, `1 + 2' would become an `expr`. It would then be impossible to shift the `!' because doing so would produce on the stack the sequence of symbols `expr '!'`. No rule allows that sequence.

The current look-ahead token is stored in the variable `yychar`. See section Special Features for Use in Actions.

Shift/Reduce Conflicts

Suppose we are parsing a language which has if-then and if-then-else statements, with a pair of rules like this:

```
if_stmt:
          IF expr THEN stmt
        | IF expr THEN stmt ELSE stmt
        ;
```

Here we assume that `IF`, `THEN` and `ELSE` are terminal symbols for specific keyword tokens.

When the `ELSE` token is read and becomes the look-ahead token, the contents of the stack (assuming the input is valid) are just right for reduction by the first rule. But it is also legitimate to shift the `ELSE`, because that would lead to eventual reduction by the second rule.

This situation, where either a shift or a reduction would be valid, is called a shift/reduce conflict. Bison is designed to resolve these conflicts by choosing to shift, unless otherwise directed by operator precedence declarations. To see the reason for this, let's contrast it with the other alternative.

Since the parser prefers to shift the `ELSE`, the result is to attach the else-clause to the innermost if-statement, making these two inputs equivalent:

```
if x then if y then win (); else lose;

if x then do; if y then win (); else lose; end;
```

But if the parser chose to reduce when possible rather than shift, the result would be to attach the else-clause to the outermost if-statement, making these two inputs equivalent:

```
if x then if y then win (); else lose;

if x then do; if y then win (); end; else lose;
```

The conflict exists because the grammar as written is ambiguous: either parsing of the simple nested if-statement is legitimate. The established convention is that these ambiguities are resolved by attaching the else-clause to the innermost if-statement; this is what Bison accomplishes by choosing to shift rather than reduce. (It would ideally be cleaner to write an unambiguous grammar, but that is very hard to do in this case.) This particular ambiguity was first encountered in the specifications of Algol 60 and is called the "dangling `else`" ambiguity.

To avoid warnings from Bison about predictable, legitimate shift/reduce conflicts, use the `%expect n` declaration. There will be no warning as long as the number of shift/reduce conflicts is exactly n. See section Suppressing Conflict Warnings.

The definition of `if_stmt` above is solely to blame for the conflict, but the conflict does not actually appear without additional rules. Here is a complete Bison input file that actually manifests the conflict:

```
%token IF THEN ELSE variable
%%
stmt:     expr
        | if_stmt
        ;

if_stmt:
          IF expr THEN stmt
        | IF expr THEN stmt ELSE stmt
        ;

expr:     variable
        ;
```

Operator Precedence

Another situation where shift/reduce conflicts appear is in arithmetic expressions. Here shifting is not always the preferred resolution; the Bison declarations for operator precedence allow you to specify when to shift and when to reduce.

When Precedence is Needed

Consider the following ambiguous grammar fragment (ambiguous because the input `1 - 2 * 3' can be parsed in two different ways):

```
expr:     expr '-' expr
        | expr '*' expr
        | expr '<' expr
        | '(' expr ')'
        ...
        ;
```

Suppose the parser has seen the tokens `1', `-' and `2'; should it reduce them via the rule for the addition operator? It depends on the next token. Of course, if the next token is `)', we must reduce; shifting is invalid because no single rule can reduce the token sequence `- 2 )' or anything starting with that. But if the next token is `*' or `<', we have a choice: either shifting or reduction would allow the parse to complete, but with different results.

To decide which one Bison should do, we must consider the results. If the next operator token op is shifted, then it must be reduced first in order to permit another opportunity to reduce the sum. The result is (in effect) `1 - (2 op 3)'. On the other hand, if the subtraction is reduced before shifting op, the result is `(1 - 2) op 3'. Clearly, then, the choice of shift or reduce should depend on the relative precedence of the operators `-' and op: `*' should be shifted first, but not `<'.

What about input such as `1 - 2 - 5'; should this be `(1 - 2) - 5' or should it be `1 - (2 - 5)'? For most operators we prefer the former, which is called left association. The latter alternative, right association, is desirable for assignment operators. The choice of left or right association is a matter of whether the parser chooses to shift or reduce when the stack contains `1 - 2' and the look-ahead token is `-': shifting makes right-associativity.

Specifying Operator Precedence

Bison allows you to specify these choices with the operator precedence declarations `%left` and `%right`. Each such declaration contains a list of tokens, which are operators whose precedence and associativity is being declared. The `%left` declaration makes all those operators left-associative and the `%right` declaration makes them right-associative. A third alternative is `%nonassoc`, which declares that it is a syntax error to find the same operator twice "in a row".

The relative precedence of different operators is controlled by the order in which they are declared. The first `%left` or `%right` declaration in the file declares the operators whose precedence is lowest, the

next such declaration declares the operators whose precedence is a little higher, and so on.

Precedence Examples

In our example, we would want the following declarations:

```
%left '<'
%left '-'
%left '*'
```

In a more complete example, which supports other operators as well, we would declare them in groups of equal precedence. For example, `'+'` is declared with `'-'`:

```
%left '<' '>' '=' NE LE GE
%left '+' '-'
%left '*' '/'
```

(Here `NE` and so on stand for the operators for "not equal" and so on. We assume that these tokens are more than one character long and therefore are represented by names, not character literals.)

How Precedence Works

The first effect of the precedence declarations is to assign precedence levels to the terminal symbols declared. The second effect is to assign precedence levels to certain rules: each rule gets its precedence from the last terminal symbol mentioned in the components. (You can also specify explicitly the precedence of a rule. See section Context-Dependent Precedence.)

Finally, the resolution of conflicts works by comparing the precedence of the rule being considered with that of the look-ahead token. If the token's precedence is higher, the choice is to shift. If the rule's precedence is higher, the choice is to reduce. If they have equal precedence, the choice is made based on the associativity of that precedence level. The verbose output file made by `-v' (see section Invoking Bison) says how each conflict was resolved.

Not all rules and not all tokens have precedence. If either the rule or the look-ahead token has no precedence, then the default is to shift.

Context-Dependent Precedence

Often the precedence of an operator depends on the context. This sounds outlandish at first, but it is really very common. For example, a minus sign typically has a very high precedence as a unary operator, and a somewhat lower precedence (lower than multiplication) as a binary operator.

The Bison precedence declarations, `%left`, `%right` and `%nonassoc`, can only be used once for a given token; so a token has only one precedence declared in this way. For context-dependent precedence, you need to use an additional mechanism: the `%prec` modifier for rules.

The `%prec` modifier declares the precedence of a particular rule by specifying a terminal symbol wh precedence should be used for that rule. It's not necessary for that symbol to appear otherwise in the The modifier's syntax is:

```
%prec terminal-symbol
```

and it is written after the components of the rule. Its effect is to assign the rule the precedence of terminal-symbol, overriding the precedence that would be deduced for it in the ordinary way. The altered rule precedence then affects how conflicts involving that rule are resolved (see section Operator Precedence).

Here is how `%prec` solves the problem of unary minus. First, declare a precedence for a fictitious terminal symbol named `UMINUS`. There are no tokens of this type, but the symbol serves to stand for its precedence:

```
...
%left '+' '-'
%left '*'
%left UMINUS
```

Now the precedence of `UMINUS` can be used in specific rules:

```
exp:    ...
        | exp '-' exp
        ...
        | '-' exp %prec UMINUS
```

Parser States

The function `yyparse` is implemented using a finite-state machine. The values pushed on the parser stack are not simply token type codes; they represent the entire sequence of terminal and nonterminal symbols at or near the top of the stack. The current state collects all the information about previous input which is relevant to deciding what to do next.

Each time a look-ahead token is read, the current parser state together with the type of look-ahead token are looked up in a table. This table entry can say, "Shift the look-ahead token." In this case, it also specifies the new parser state, which is pushed onto the top of the parser stack. Or it can say, "Reduce using rule number n." This means that a certain number of tokens or groupings are taken off the top of the stack, and replaced by one grouping. In other words, that number of states are popped from the stack, and one new state is pushed.

There is one other alternative: the table can say that the look-ahead token is erroneous in the current state. This causes error processing to begin (see section Error Recovery).

Reduce/Reduce Conflicts

A reduce/reduce conflict occurs if there are two or more rules that apply to the same sequence of input. This usually indicates a serious error in the grammar.

For example, here is an erroneous attempt to define a sequence of zero or more `word` groupings.

```
sequence: /* empty */
                { printf ("empty sequence\n"); }
        | maybeword
        | sequence word
                { printf ("added word %s\n", $2); }
        ;

maybeword: /* empty */
                { printf ("empty maybeword\n"); }
        | word
                { printf ("single word %s\n", $1); }
        ;
```

The error is an ambiguity: there is more than one way to parse a single `word` into a `sequence`. It could be reduced to a `maybeword` and then into a `sequence` via the second rule. Alternatively, nothing-at-all could be reduced into a `sequence` via the first rule, and this could be combined with the `word` using the third rule for `sequence`.

There is also more than one way to reduce nothing-at-all into a `sequence`. This can be done directly via the first rule, or indirectly via `maybeword` and then the second rule.

You might think that this is a distinction without a difference, because it does not change whether any particular input is valid or not. But it does affect which actions are run. One parsing order runs the second rule's action; the other runs the first rule's action and the third rule's action. In this example, the output of the program changes.

Bison resolves a reduce/reduce conflict by choosing to use the rule that appears first in the grammar, but it is very risky to rely on this. Every reduce/reduce conflict must be studied and usually eliminated. Here is the proper way to define `sequence`:

```
sequence: /* empty */
                { printf ("empty sequence\n"); }
        | sequence word
                { printf ("added word %s\n", $2); }
        ;
```

Here is another common error that yields a reduce/reduce conflict:

```
sequence: /* empty */
```


```
        | sequence words
        | sequence redirects
        ;

words:    /* empty */
        | words word
        ;

redirects:/* empty */
        | redirects redirect
        ;
```

The intention here is to define a sequence which can contain either `word` or `redirect` groupings. The individual definitions of `sequence`, `words` and `redirects` are error-free, but the three together make a subtle ambiguity: even an empty input can be parsed in infinitely many ways!

Consider: nothing-at-all could be a `words`. Or it could be two `words` in a row, or three, or any number. It could equally well be a `redirects`, or two, or any number. Or it could be a `words` followed by three `redirects` and another `words`. And so on.

Here are two ways to correct these rules. First, to make it a single level of sequence:

```
sequence: /* empty */
        | sequence word
        | sequence redirect
        ;
```

Second, to prevent either a `words` or a `redirects` from being empty:

```
sequence: /* empty */
        | sequence words
        | sequence redirects
        ;

words:    word
        | words word
        ;

redirects:redirect
        | redirects redirect
        ;
```

Mysterious Reduce/Reduce Conflicts

Sometimes reduce/reduce conflicts can occur that don't look warranted. Here is an example:

```
%token ID

%%
def:    param_spec return_spec ','
        ;
param_spec:
                type
        |       name_list ':' type
        ;
return_spec:
                type
        |       name ':' type
        ;
type:           ID
        ;
name:           ID
        ;
name_list:
                name
        |       name ',' name_list
        ;
```

It would seem that this grammar can be parsed with only a single token of look-ahead: when a `param_spec` is being read, an `ID` is a `name` if a comma or colon follows, or a `type` if another `ID` follows. In other words, this grammar is LR(1).

However, Bison, like most parser generators, cannot actually handle all LR(1) grammars. In this grammar, two contexts, that after an `ID` at the beginning of a `param_spec` and likewise at the beginning of a `return_spec`, are similar enough that Bison assumes they are the same. They appear similar because the same set of rules would be active--the rule for reducing to a `name` and that for reducing to a `type`. Bison is unable to determine at that stage of processing that the rules would require different look-ahead tokens in the two contexts, so it makes a single parser state for them both. Combining the two contexts causes a conflict later. In parser terminology, this occurrence means that the grammar is not LALR(1).

In general, it is better to fix deficiencies than to document them. But this particular deficiency is intrinsically hard to fix; parser generators that can handle LR(1) grammars are hard to write and tend to produce parsers that are very large. In practice, Bison is more useful as it is now.

When the problem arises, you can often fix it by identifying the two parser states that are being confused, and adding something to make them look distinct. In the above example, adding one rule to `return_spec` as follows makes the problem go away:


```
%token BOGUS
...
%%
...
return_spec:
          type
        |    name ':' type
        /* This rule is never used.  */
        |    ID BOGUS
        ;
```

This corrects the problem because it introduces the possibility of an additional active rule in the context after the `ID` at the beginning of `return_spec`. This rule is not active in the corresponding context in a `param_spec`, so the two contexts receive distinct parser states. As long as the token `BOGUS` is never generated by `yylex`, the added rule cannot alter the way actual input is parsed.

In this particular example, there is another way to solve the problem: rewrite the rule for `return_spec` to use `ID` directly instead of via `name`. This also causes the two confusing contexts to have different sets of active rules, because the one for `return_spec` activates the altered rule for `return_spec` rather than the one for `name`.

```
param_spec:
          type
        |    name_list ':' type
        ;
return_spec:
          type
        |    ID ':' type
        ;
```

Stack Overflow, and How to Avoid It

The Bison parser stack can overflow if too many tokens are shifted and not reduced. When this happens, the parser function `yyparse` returns a nonzero value, pausing only to call `yyerror` to report the overflow.

By defining the macro `YYMAXDEPTH`, you can control how deep the parser stack can become before a stack overflow occurs. Define the macro with a value that is an integer. This value is the maximum number of tokens that can be shifted (and not reduced) before overflow. It must be a constant expression whose value is known at compile time.

The stack space allowed is not necessarily allocated. If you specify a large value for `YYMAXDEPTH`, the parser actually allocates a small stack at first, and then makes it bigger by stages as needed. This increasing allocation happens automatically and silently. Therefore, you do not need to make `YYMAXDEPTH` painfully small merely to save space for ordinary inputs that do not need much stack.

The default value of YYMAXDEPTH, if you do not define it, is 10000.

You can control how much stack is allocated initially by defining the macro YYINITDEPTH. This value too must be a compile-time constant integer. The default is 200.

Go to the previous, next section.

Go to the previous, next section.

Error Recovery

It is not usually acceptable to have a program terminate on a parse error. For example, a compiler should recover sufficiently to parse the rest of the input file and check it for errors; a calculator should accept another expression.

In a simple interactive command parser where each input is one line, it may be sufficient to allow `yyparse` to return 1 on error and have the caller ignore the rest of the input line when that happens (and then call `yyparse` again). But this is inadequate for a compiler, because it forgets all the syntactic context leading up to the error. A syntax error deep within a function in the compiler input should not cause the compiler to treat the following line like the beginning of a source file.

You can define how to recover from a syntax error by writing rules to recognize the special token `error`. This is a terminal symbol that is always defined (you need not declare it) and reserved for error handling. The Bison parser generates an `error` token whenever a syntax error happens; if you have provided a rule to recognize this token in the current context, the parse can continue.

For example:

```
stmnts:  /* empty string */
        | stmnts '\n'
        | stmnts exp '\n'
        | stmnts error '\n'
```

The fourth rule in this example says that an error followed by a newline makes a valid addition to any `stmnts`.

What happens if a syntax error occurs in the middle of an `exp`? The error recovery rule, interpreted strictly, applies to the precise sequence of a `stmnts`, an `error` and a newline. If an error occurs in the middle of an `exp`, there will probably be some additional tokens and subexpressions on the stack after the last `stmnts`, and there will be tokens to read before the next newline. So the rule is not applicable in the ordinary way.

But Bison can force the situation to fit the rule, by discarding part of the semantic context and part of the input. First it discards states and objects from the stack until it gets back to a state in which the `error` token is acceptable. (This means that the subexpressions already parsed are discarded, back to the last complete `stmnts`.) At this point the `error` token can be shifted. Then, if the old look-ahead token is not acceptable to be shifted next, the parser reads tokens and discards them until it finds a token which is acceptable. In this example, Bison reads and discards input until the next newline so that the fourth rule can apply.

The choice of error rules in the grammar is a choice of strategies for error recovery. A simple and useful strategy is simply to skip the rest of the current input line or current statement if an error is detected:


```
stmnt: error ';'  /* on error, skip until ';' is read */
```

It is also useful to recover to the matching close-delimiter of an opening-delimiter that has already been parsed. Otherwise the close-delimiter will probably appear to be unmatched, and generate another, spurious error message:

```
primary:  '(' expr ')'
        | '(' error ')'
        ...
        ;
```

Error recovery strategies are necessarily guesses. When they guess wrong, one syntax error often leads to another. In the above example, the error recovery rule guesses that an error is due to bad input within one `stmnt`. Suppose that instead a spurious semicolon is inserted in the middle of a valid `stmnt`. After the error recovery rule recovers from the first error, another syntax error will be found straightaway, since the text following the spurious semicolon is also an invalid `stmnt`.

To prevent an outpouring of error messages, the parser will output no error message for another syntax error that happens shortly after the first; only after three consecutive input tokens have been successfully shifted will error messages resume.

Note that rules which accept the `error` token may have actions, just as any other rules can.

You can make error messages resume immediately by using the macro `yyerrok` in an action. If you do this in the error rule's action, no error messages will be suppressed. This macro requires no arguments; `yyerrok;' is a valid C statement.

The previous look-ahead token is reanalyzed immediately after an error. If this is unacceptable, then the macro `yyclearin` may be used to clear this token. Write the statement `yyclearin;' in the error rule's action.

For example, suppose that on a parse error, an error handling routine is called that advances the input stream to some point where parsing should once again commence. The next symbol returned by the lexical scanner is probably correct. The previous look-ahead token ought to be discarded with `yyclearin;'.

The macro `YYRECOVERING` stands for an expression that has the value 1 when the parser is recovering from a syntax error, and 0 the rest of the time. A value of 1 indicates that error messages are currently suppressed for new syntax errors.

Go to the previous, next section.

Go to the previous, next section.

Handling Context Dependencies

The Bison paradigm is to parse tokens first, then group them into larger syntactic units. In many languages, the meaning of a token is affected by its context. Although this violates the Bison paradigm, certain techniques (known as kludges) may enable you to write Bison parsers for such languages.

(Actually, "kludge" means any technique that gets its job done but is neither clean nor robust.)

Semantic Info in Token Types

The C language has a context dependency: the way an identifier is used depends on what its current meaning is. For example, consider this:

```
foo (x);
```

This looks like a function call statement, but if `foo` is a typedef name, then this is actually a declaration of `x`. How can a Bison parser for C decide how to parse this input?

The method used in GNU C is to have two different token types, `IDENTIFIER` and `TYPENAME`. When `yylex` finds an identifier, it looks up the current declaration of the identifier in order to decide which token type to return: `TYPENAME` if the identifier is declared as a typedef, `IDENTIFIER` otherwise.

The grammar rules can then express the context dependency by the choice of token type to recognize. `IDENTIFIER` is accepted as an expression, but `TYPENAME` is not. `TYPENAME` can start a declaration, but `IDENTIFIER` cannot. In contexts where the meaning of the identifier is *not* significant, such as in declarations that can shadow a typedef name, either `TYPENAME` or `IDENTIFIER` is accepted--there is one rule for each of the two token types.

This technique is simple to use if the decision of which kinds of identifiers to allow is made at a place close to where the identifier is parsed. But in C this is not always so: C allows a declaration to redeclare a typedef name provided an explicit type has been specified earlier:

```
typedef int foo, bar, lose;
static foo (bar);        /* redeclare bar as static variable */
static int foo (lose);   /* redeclare foo as function */
```

Unfortunately, the name being declared is separated from the declaration construct itself by a complicated syntactic structure--the "declarator".

As a result, the part of Bison parser for C needs to be duplicated, with all the nonterminal names changed: once for parsing a declaration in which a typedef name can be redefined, and once for parsing a declaration in which that can't be done. Here is a part of the duplication, with actions omitted for brevity:


```
initdcl:
          declarator maybeasm '='
          init
        | declarator maybeasm
        ;

notype_initdcl:
          notype_declarator maybeasm '='
          init
        | notype_declarator maybeasm
        ;
```

Here `initdcl` can redeclare a typedef name, but `notype_initdcl` cannot. The distinction between `declarator` and `notype_declarator` is the same sort of thing.

There is some similarity between this technique and a lexical tie-in (described next), in that information which alters the lexical analysis is changed during parsing by other parts of the program. The difference is here the information is global, and is used for other purposes in the program. A true lexical tie-in has a special-purpose flag controlled by the syntactic context.

Lexical Tie-ins

One way to handle context-dependency is the lexical tie-in: a flag which is set by Bison actions, whose purpose is to alter the way tokens are parsed.

For example, suppose we have a language vaguely like C, but with a special construct `hex (hex-expr)'. After the keyword `hex` comes an expression in parentheses in which all integers are hexadecimal. In particular, the token `a1b' must be treated as an integer rather than as an identifier if it appears in that context. Here is how you can do it:

```
%{
int hexflag;
%}
%%
...
expr:   IDENTIFIER
        | constant
        | HEX '('
                { hexflag = 1; }
          expr ')'
                { hexflag = 0;
                   $$ = $4; }
        | expr '+' expr
                { $$ = make_sum ($1, $3); }
        ...
        ;
```


```
constant:
          INTEGER
        | STRING
        ;
```

Here we assume that `yylex` looks at the value of `hexflag`; when it is nonzero, all integers are parsed in hexadecimal, and tokens starting with letters are parsed as integers if possible.

The declaration of `hexflag` shown in the C declarations section of the parser file is needed to make it accessible to the actions (see section The C Declarations Section). You must also write the code in `yylex` to obey the flag.

Lexical Tie-ins and Error Recovery

Lexical tie-ins make strict demands on any error recovery rules you have. See section Error Recovery.

The reason for this is that the purpose of an error recovery rule is to abort the parsing of one construct and resume in some larger construct. For example, in C-like languages, a typical error recovery rule is to skip tokens until the next semicolon, and then start a new statement, like this:

```
stmt:   expr ';'
        | IF '(' expr ')' stmt { ... }
        ...
        error ';'
                { hexflag = 0; }
        ;
```

If there is a syntax error in the middle of a `hex (expr)' construct, this error rule will apply, and then the action for the completed `hex (expr)' will never run. So `hexflag` would remain set for the entire rest of the input, or until the next `hex` keyword, causing identifiers to be misinterpreted as integers.

To avoid this problem the error recovery rule itself clears `hexflag`.

There may also be an error recovery rule that works within expressions. For example, there could be a rule which applies within parentheses and skips to the close-parenthesis:

```
expr:   ...
        | '(' expr ')'
                { $$ = $2; }
        | '(' error ')'
        ...
```

If this rule acts within the `hex` construct, it is not going to abort that construct (since it applies to an inner level of parentheses within the construct). Therefore, it should not clear the flag: the rest of the `hex` construct should be parsed with the flag still in effect.

What if there is an error recovery rule which might abort out of the `hex` construct or might not, depending on circumstances? There is no way you can write the action to determine whether a `hex` construct is being aborted or not. So if you are using a lexical tie-in, you had better make sure your error recovery rules are not of this kind. Each rule must be such that you can be sure that it always will, or always won't, have to clear the flag.

Go to the previous, next section.

Go to the previous, next section.

Debugging Your Parser

If a Bison grammar compiles properly but doesn't do what you want when it runs, the `yydebug` parser-trace feature can help you figure out why.

To enable compilation of trace facilities, you must define the macro `YYDEBUG` when you compile the parser. You could use `-DYYDEBUG=1' as a compiler option or you could put `#define YYDEBUG 1' in the C declarations section of the grammar file (see section The C Declarations Section). Alternatively, use the `-t' option when you run Bison (see section Invoking Bison). We always define `YYDEBUG` so that debugging is always possible.

The trace facility uses `stderr`, so you must add `#include <stdio.h>` to the C declarations section unless it is already there.

Once you have compiled the program with trace facilities, the way to request a trace is to store a nonzero value in the variable `yydebug`. You can do this by making the C code do it (in `main`, perhaps), or you can alter the value with a C debugger.

Each step taken by the parser when `yydebug` is nonzero produces a line or two of trace information, written on `stderr`. The trace messages tell you these things:

- Each time the parser calls `yylex`, what kind of token was read.
- Each time a token is shifted, the depth and complete contents of the state stack (see section Parser States).
- Each time a rule is reduced, which rule it is, and the complete contents of the state stack afterward.

To make sense of this information, it helps to refer to the listing file produced by the Bison `-v' option (see section Invoking Bison). This file shows the meaning of each state in terms of positions in various rules, and also what each state will do with each possible input token. As you read the successive trace messages, you can see that the parser is functioning according to its specification in the listing file. Eventually you will arrive at the place where something undesirable happens, and you will see which parts of the grammar are to blame.

The parser file is a C program and you can use C debuggers on it, but it's not easy to interpret what it is doing. The parser function is a finite-state machine interpreter, and aside from the actions it executes the same code over and over. Only the values of variables show where in the grammar it is working.

The debugging information normally gives the token type of each token read, but not its semantic value. You can optionally define a macro named `YYPRINT` to provide a way to print the value. If you define `YYPRINT`, it should take three arguments. The parser will pass a standard I/O stream, the numeric code for the token type, and the token value (from `yylval`).

Here is an example of `YYPRINT` suitable for the multi-function calculator (see section Declarations for `mfcalc`):


```
#define YYPRINT(file, type, value)   yyprint (file, type, value)

static void
yyprint (file, type, value)
     FILE *file;
     int type;
     YYSTYPE value;
{
  if (type == VAR)
    fprintf (file, " %s", value.tptr->name);
  else if (type == NUM)
    fprintf (file, " %d", value.val);
}
```

Go to the previous, next section.

Go to the previous, next section.

Invoking Bison

The usual way to invoke Bison is as follows:

```
bison infile
```

Here infile is the grammar file name, which usually ends in `.y'. The parser file's name is made by replacing the `.y' with `.tab.c'. Thus, the `bison foo.y' filename yields `foo.tab.c', and the `bison hack/foo.y' filename yields `hack/foo.tab.c'.

Bison Options

Bison supports both traditional single-letter options and mnemonic long option names. Long option names are indicated with `--' instead of `-'. Abbreviations for option names are allowed as long as they are unique. When a long option takes an argument, like `--file-prefix', connect the option name and the argument with `='.

Here is a list of options that can be used with Bison, alphabetized by short option. It is followed by a cross key alphabetized by long option.

`-b file-prefix'

`--file-prefix=prefix'

> Specify a prefix to use for all Bison output file names. The names are chosen as if the input file were named `prefix.c'.

`-d'

`--defines'

> Write an extra output file containing macro definitions for the token type names defined in the grammar and the semantic value type YYSTYPE, as well as a few `extern` variable declarations.
>
> If the parser output file is named `name.c' then this file is named `name.h'.
>
> This output file is essential if you wish to put the definition of `yylex` in a separate source file, because `yylex` needs to be able to refer to token type codes and the variable `yylval`. See section Semantic Values of Tokens.

`-l'

`--no-lines'

> Don't put any `#line` preprocessor commands in the parser file. Ordinarily Bison puts them in the parser file so that the C compiler and debuggers will associate errors with your source file, the grammar file. This option causes them to associate errors with the parser file, treating it as an independent source file in its own right.

`-n'

`--no-parser'

Do not include any C code in the parser file; generate tables only. The parser file contains just `#define` directives and static variable declarations.

This option also tells Bison to write the C code for the grammar actions into a file named `` `filename.act' ``, in the form of a brace-surrounded body fit for a `switch` statement.

`-o outfile'

`--output-file=outfile'

Specify the name outfile for the parser file.

The other output files' names are constructed from outfile as described under the `-v' and `-d' options.

`-p prefix'

`--name-prefix=prefix'

Rename the external symbols used in the parser so that they start with prefix instead of `yy'. The precise list of symbols renamed is `yyparse`, `yylex`, `yyerror`, `yynerrs`, `yylval`, `yychar` and `yydebug`.

For example, if you use `-p c', the names become `cparse`, `clex`, and so on.

See section Multiple Parsers in the Same Program.

`-r'

`--raw'

Pretend that `%raw` was specified. See section Bison Declaration Summary.

`-t'

`--debug'

Output a definition of the macro `YYDEBUG` into the parser file, so that the debugging facilities are compiled. See section Debugging Your Parser.

`-v'

`--verbose'

Write an extra output file containing verbose descriptions of the parser states and what is done for each type of look-ahead token in that state.

This file also describes all the conflicts, both those resolved by operator precedence and the unresolved ones.

The file's name is made by removing `.tab.c' or `.c' from the parser output file name, and adding `.output' instead.

Therefore, if the input file is `` `foo.y' ``, then the parser file is called `` `foo.tab.c' `` by default. As a consequence, the verbose output file is called `` `foo.output' ``.

`-V'

`--version'

Print the version number of Bison and exit.

`-h'

`--help'

Print a summary of the command-line options to Bison and exit.

`-y'

`--yacc'

`--fixed-output-files'

Equivalent to `-o y.tab.c'; the parser output file is called `y.tab.c', and the other outputs are called `y.output' and `y.tab.h'. The purpose of this option is to imitate Yacc's output file name conventions. Thus, the following shell script can substitute for Yacc:

```
bison -y $*
```

Option Cross Key

Here is a list of options, alphabetized by long option, to help you find the corresponding short option.

Invoking Bison under VMS

The command line syntax for Bison on VMS is a variant of the usual Bison command syntax--adapted to fit VMS conventions.

To find the VMS equivalent for any Bison option, start with the long option, and substitute a `/' for the leading `--', and substitute a `_' for each `-' in the name of the long option. For example, the following invocation under VMS:

```
bison /debug/name_prefix=bar foo.y
```

is equivalent to the following command under POSIX.

```
bison --debug --name-prefix=bar foo.y
```

The VMS file system does not permit filenames such as `foo.tab.c'. In the above example, the output file would instead be named `foo_tab.c'.

Go to the previous, next section.

Go to the previous, next section.

Bison Symbols

`error`

A token name reserved for error recovery. This token may be used in grammar rules so as to allow the Bison parser to recognize an error in the grammar without halting the process. In effect, a sentence containing an error may be recognized as valid. On a parse error, the token `error` becomes the current look-ahead token. Actions corresponding to `error` are then executed, and the look-ahead token is reset to the token that originally caused the violation. See section Error Recovery.

`YYABORT`

Macro to pretend that an unrecoverable syntax error has occurred, by making `yyparse` return 1 immediately. The error reporting function `yyerror` is not called. See section The Parser Function `yyparse`.

`YYACCEPT`

Macro to pretend that a complete utterance of the language has been read, by making `yyparse` return 0 immediately. See section The Parser Function `yyparse`.

`YYBACKUP`

Macro to discard a value from the parser stack and fake a look-ahead token. See section Special Features for Use in Actions.

`YYERROR`

Macro to pretend that a syntax error has just been detected: call `yyerror` and then perform normal error recovery if possible (see section Error Recovery), or (if recovery is impossible) make `yyparse` return 1. See section Error Recovery.

`YYERROR_VERBOSE`

Macro that you define with `#define` in the Bison declarations section to request verbose, specific error message strings when `yyerror` is called.

`YYINITDEPTH`

Macro for specifying the initial size of the parser stack. See section Stack Overflow, and How to Avoid It.

`YYLEX_PARAM`

Macro for specifying an extra argument (or list of extra arguments) for `yyparse` to pass to `yylex`. See section Calling Conventions for Pure Parsers.

`YYLTYPE`

Macro for the data type of `yylloc`; a structure with four members. See section Textual Positions of Tokens.

`yyltype`

Default value for YYLTYPE.

`YYMAXDEPTH`

Macro for specifying the maximum size of the parser stack. See section Stack Overflow, and How to Avoid It.

`YYPARSE_PARAM`

Macro for specifying the name of a parameter that `yyparse` should accept. See section Calling Conventions for Pure Parsers.

`YYRECOVERING`

Macro whose value indicates whether the parser is recovering from a syntax error. See section Special Features for Use in Actions.

`YYSTYPE`

Macro for the data type of semantic values; `int` by default. See section Data Types of Semantic Values.

`yychar`

External integer variable that contains the integer value of the current look-ahead token. (In a pure parser, it is a local variable within `yyparse`.) Error-recovery rule actions may examine this variable. See section Special Features for Use in Actions.

`yyclearin`

Macro used in error-recovery rule actions. It clears the previous look-ahead token. See section Error Recovery.

`yydebug`

External integer variable set to zero by default. If `yydebug` is given a nonzero value, the parser will output information on input symbols and parser action. See section Debugging Your Parser.

`yyerrok`

Macro to cause parser to recover immediately to its normal mode after a parse error. See section Error Recovery.

`yyerror`

User-supplied function to be called by `yyparse` on error. The function receives one argument, a pointer to a character string containing an error message. See section The Error Reporting Function `yyerror`.

`yylex`

User-supplied lexical analyzer function, called with no arguments to get the next token. See section The Lexical Analyzer Function `yylex`.

`yylval`

External variable in which `yylex` should place the semantic value associated with a token. (In a pure parser, it is a local variable within `yyparse`, and its address is passed to `yylex`.) See section Semantic Values of Tokens.

`yylloc`

External variable in which `yylex` should place the line and column numbers associated with a token. (In a pure parser, it is a local variable within `yyparse`, and its address is passed to `yylex`.) You can ignore this variable if you don't use the `@' feature in the grammar actions. See section Textual Positions of Tokens.

`yynerrs`

Global variable which Bison increments each time there is a parse error. (In a pure parser, it is a local variable within `yyparse`.) See section The Error Reporting Function `yyerror`.

`yyparse`

The parser function produced by Bison; call this function to start parsing. See section The Parser Function `yyparse`.

`%left`

Bison declaration to assign left associativity to token(s). See section Operator Precedence.

`%no_lines`

Bison declaration to avoid generating `#line` directives in the parser file. See section Bison Declaration Summary.

`%nonassoc`

Bison declaration to assign nonassociativity to token(s). See section Operator Precedence.

`%prec`

Bison declaration to assign a precedence to a specific rule. See section Context-Dependent Precedence.

`%pure_parser`

Bison declaration to request a pure (reentrant) parser. See section A Pure (Reentrant) Parser.

`%raw`

Bison declaration to use Bison internal token code numbers in token tables instead of the usual Yacc-compatible token code numbers. See section Bison Declaration Summary.

`%right`

Bison declaration to assign right associativity to token(s). See section Operator Precedence.

`%start`

Bison declaration to specify the start symbol. See section The Start-Symbol.

`%token`

Bison declaration to declare token(s) without specifying precedence. See section Token Type Names.

`%token_table`

Bison declaration to include a token name table in the parser file. See section Bison Declaration Summary.

`%type`

Bison declaration to declare nonterminals. See section Nonterminal Symbols.

`%union`

Bison declaration to specify several possible data types for semantic values. See section The Collection of Value Types.

These are the punctuation and delimiters used in Bison input:

`%%'

Delimiter used to separate the grammar rule section from the Bison declarations section or the additional C code section. See section The Overall Layout of a Bison Grammar.

`%{ %}'

All code listed between `%{' and `%}' is copied directly to the output file uninterpreted. Such code forms the "C declarations" section of the input file. See section Outline of a Bison Grammar.

`/*...*/'

Comment delimiters, as in C.

`:'

Separates a rule's result from its components. See section Syntax of Grammar Rules.

`;'

Terminates a rule. See section Syntax of Grammar Rules.

`|'

Separates alternate rules for the same result nonterminal. See section Syntax of Grammar Rules.

Go to the previous, next section.

Go to the previous, next section.

TOC

Glossary

Backus-Naur Form (BNF)

Formal method of specifying context-free grammars. BNF was first used in the ALGOL-60 report, 1963. See section Languages and Context-Free Grammars.

Context-free grammars

Grammars specified as rules that can be applied regardless of context. Thus, if there is a rule which says that an integer can be used as an expression, integers are allowed *anywhere* an expression is permitted. See section Languages and Context-Free Grammars.

Dynamic allocation

Allocation of memory that occurs during execution, rather than at compile time or on entry to a function.

Empty string

Analogous to the empty set in set theory, the empty string is a character string of length zero.

Finite-state stack machine

A "machine" that has discrete states in which it is said to exist at each instant in time. As input to the machine is processed, the machine moves from state to state as specified by the logic of the machine. In the case of the parser, the input is the language being parsed, and the states correspond to various stages in the grammar rules. See section The Bison Parser Algorithm.

Grouping

A language construct that is (in general) grammatically divisible; for example, `expression' or `declaration' in C. See section Languages and Context-Free Grammars.

Infix operator

An arithmetic operator that is placed between the operands on which it performs some operation.

Input stream

A continuous flow of data between devices or programs.

Language construct

One of the typical usage schemas of the language. For example, one of the constructs of the C language is the `if` statement. See section Languages and Context-Free Grammars.

Left associativity

Operators having left associativity are analyzed from left to right: `a+b+c' first computes `a+b' and then combines with `c'. See section Operator Precedence.

Left recursion

A rule whose result symbol is also its first component symbol; for example, `expseq1 : expseq1 ',' exp;'. See section Recursive Rules.

Left-to-right parsing

Parsing a sentence of a language by analyzing it token by token from left to right. See section Bison Parser Algorithm.

Lexical analyzer (scanner)

A function that reads an input stream and returns tokens one by one. See section The Lexical Analyzer Function `yylex`.

Lexical tie-in

A flag, set by actions in the grammar rules, which alters the way tokens are parsed. See section Lexical Tie-ins.

Literal string token

A token which constists of two or more fixed characters. See section Symbols, Terminal and Nonterminal.

Look-ahead token

A token already read but not yet shifted. See section Look-Ahead Tokens.

LALR(1)

The class of context-free grammars that Bison (like most other parser generators) can handle; a subset of LR(1). See section Mysterious Reduce/Reduce Conflicts.

LR(1)

The class of context-free grammars in which at most one token of look-ahead is needed to disambiguate the parsing of any piece of input.

Nonterminal symbol

A grammar symbol standing for a grammatical construct that can be expressed through rules in terms of smaller constructs; in other words, a construct that is not a token. See section Symbols, Terminal and Nonterminal.

Parse error

An error encountered during parsing of an input stream due to invalid syntax. See section Error Recovery.

Parser

A function that recognizes valid sentences of a language by analyzing the syntax structure of a set of tokens passed to it from a lexical analyzer.

Postfix operator

An arithmetic operator that is placed after the operands upon which it performs some operation.

Reduction

Replacing a string of nonterminals and/or terminals with a single nonterminal, according to a grammar rule. See section The Bison Parser Algorithm.

Reentrant

A reentrant subprogram is a subprogram which can be in invoked any number of times in parallel, without interference between the various invocations. See section A Pure (Reentrant) Parser.

Reverse polish notation

A language in which all operators are postfix operators.

Right recursion

A rule whose result symbol is also its last component symbol; for example, `expseq1: exp ',' expseq1;'. See section Recursive Rules.

Semantics

In computer languages, the semantics are specified by the actions taken for each instance of the language, i.e., the meaning of each statement. See section Defining Language Semantics.

Shift

A parser is said to shift when it makes the choice of analyzing further input from the stream rather than reducing immediately some already-recognized rule. See section The Bison Parser Algorithm.

Single-character literal

A single character that is recognized and interpreted as is. See section From Formal Rules to Bison Input.

Start symbol

The nonterminal symbol that stands for a complete valid utterance in the language being parsed. The start symbol is usually listed as the first nonterminal symbol in a language specification. See section The Start-Symbol.

Symbol table

A data structure where symbol names and associated data are stored during parsing to allow for recognition and use of existing information in repeated uses of a symbol. See section Multi-Function Calculator: `mfcalc`.

Token

A basic, grammatically indivisible unit of a language. The symbol that describes a token in the grammar is a terminal symbol. The input of the Bison parser is a stream of tokens which comes from the lexical analyzer. See section Symbols, Terminal and Nonterminal.

Terminal symbol

A grammar symbol that has no rules in the grammar and therefore is grammatically indivisible. The piece of text it represents is a token. See section Languages and Context-Free Grammars.

Go to the previous, next section.

Go to the previous section.

Index

$

%

@

a

b

c

d

e

f

g

i

l

m

n

o

p

r

s

t

u

v

w

y

- |

Go to the previous section.

GNU Emacs Calc 2.02 Manual

Go to the next section.

@textfont0=@tenrm @font@teni=cmmi10 scaled @magstephalf @textfont1=@teni @font@seveni=cmmi7 scaled @magstephalf @scriptfont1=@seveni @font@fivei=cmmi5 scaled @magstephalf @scriptscriptfont1=@fivei @font@tensy=cmsy10 scaled @magstephalf @textfont2=@tensy @font@sevensy=cmsy7 scaled @magstephalf @scriptfont2=@sevensy @font@fivesy=cmsy5 scaled @magstephalf @scriptscriptfont2=@fivesy @font@tenex=cmex10 scaled @magstephalf @textfont3=@tenex @scriptfont3=@tenex @scriptscriptfont3=@tenex @textfont7=@tentt @scriptfont7=@tentt @scriptscriptfont7=@tentt

@mathcode`=` @tocindent=.5pc @rightskip=0pt plus 2pt @newdimen@kyvpos @kyvpos=0pt @newdimen@kyhpos @kyhpos=0pt @newcount@calcclubpenalty @calcclubpenalty=1000 @newcount@calcpageno @newtoks@calcoldeverypar @calcoldeverypar=@everypar @everypar={@calceverypar@the@calcoldeverypar} @ifx@turnoffactive@undefinedzzz@def@turnoffactive{}@fi @ifx@ninett@undefinedzzz@font@ninett=cmtt9@fi @catcode`@\=0 \catcode`\@=11 \r@ggedbottomtrue \catcode`\@=0 @catcode`@\=@active

Calc Manual

GNU Emacs Calc Version 2.02

January 1992

Dave Gillespie daveg@synaptics.com

GNU GENERAL PUBLIC LICENSE

Version 1, February 1989

Preamble

The license agreements of most software companies try to keep users at the mercy of those companies. By contrast, our General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. The General Public License applies to the Free Software Foundation's software and to any other program whose authors commit to using it. You can use it for your programs, too.

When we speak of free software, we are referring to freedom, not price. Specifically, the General Public License is designed to make sure that you have the freedom to give away or sell copies of free software, that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of a such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

1. This License Agreement applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any work containing the Program or a portion of it, either verbatim or with modifications. Each licensee is addressed as "you".
2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this General Public License and to the absence of any warranty; and give any other recipients of the Program a copy of this General Public License along with the Program. You may charge a fee for the physical act of transferring a copy.
3. You may modify your copy or copies of the Program or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:
   - cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and
   - cause the whole of any work that you distribute or publish, that in whole or in part contains the Program or any part thereof, either with or without modifications, to be licensed at no charge to all third parties under the terms of this General Public License (except that you may choose to grant warranty protection to some or all third parties, at your option).
   - If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the simplest and most usual way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this General Public License.
   - You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

   Mere aggregation of another independent work with the Program (or its derivative) on a volume of a storage or distribution medium does not bring the other work under the scope of these terms.
4. You may copy and distribute the Program (or a portion or derivative of it, under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:
   - accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal charge for the cost of distribution) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only

if you received the program in object code or executable form alone.)

Source code for a work means the preferred form of the work for making modifications to it. For an executable file, complete source code means all the source code for all modules it contains, but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs, or for standard header files or definitions files that accompany that operating system.

5. You may not copy, modify, sublicense, distribute or transfer the Program except as expressly provided under this General Public License. Any attempt otherwise to copy, modify, sublicense, distribute or transfer the Program is void, and will automatically terminate your rights to use the Program under this License. However, parties who have received copies, or rights to use copies, from you under this General Public License will not have their licenses terminated so long as such parties remain in full compliance.
6. By copying, distributing or modifying the Program (or any work based on the Program) you indicate your acceptance of this license to do so, and all its terms and conditions.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.
8. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of the license which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the license, you may choose any version ever published by the Free Software Foundation.
9. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

10. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS

WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

11. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Go to the next section.

Go to the previous, next section.

Getting Started

This chapter provides a general overview of Calc, the GNU Emacs Calculator: What it is, how to start it and how to exit from it, and what are the various ways that it can be used.

What is Calc?

Calc is an advanced calculator and mathematical tool that runs as part of the GNU Emacs environment. Very roughly based on the HP-28/48 series of calculators, its many features include:

- Choice of algebraic or RPN (stack-based) entry of calculations.
- Arbitrary precision integers and floating-point numbers.
- Arithmetic on rational numbers, complex numbers (rectangular and polar), error forms with standard deviations, open and closed intervals, vectors and matrices, dates and times, infinities, sets, quantities with units, and algebraic formulas.
- Mathematical operations such as logarithms and trigonometric functions.
- Programmer's features (bitwise operations, non-decimal numbers).
- Financial functions such as future value and internal rate of return.
- Number theoretical features such as prime factorization and arithmetic modulo *M* for any *M*.
- Algebraic manipulation features, including symbolic calculus.
- Moving data to and from regular editing buffers.
- "Embedded mode" for manipulating Calc formulas and data directly inside any editing buffer.
- Graphics using GNUPLOT, a versatile (and free) plotting program.
- Easy programming using keyboard macros, algebraic formulas, algebraic rewrite rules, or extended Emacs Lisp.

Calc tries to include a little something for everyone; as a result it is large and might be intimidating to the first-time user. If you plan to use Calc only as a traditional desk calculator, all you really need to read is the "Getting Started" chapter of this manual and possibly the first few sections of the tutorial. As you become more comfortable with the program you can learn its additional features. In terms of efficiency, scope and depth, Calc cannot replace a powerful tool like Mathematica. But Calc has the advantages of convenience, portability, and availability of the source code. And, of course, it's free!

About This Manual

This document serves as a complete description of the GNU Emacs Calculator. It works both as an introduction for novices, and as a reference for experienced users. While it helps to have some experience with GNU Emacs in order to get the most out of Calc, this manual ought to be readable even if you don't know or use Emacs regularly.

The manual is divided into three major parts: the "Getting Started" chapter you are reading now, the Calc

tutorial (chapter 2), and the Calc reference manual (the remaining chapters and appendices).

If you are in a hurry to use Calc, there is a brief "demonstration" below which illustrates the major features of Calc in just a couple of pages. If you don't have time to go through the full tutorial, this will show you everything you need to know to begin. See section A Demonstration of Calc.

The tutorial chapter walks you through the various parts of Calc with lots of hands-on examples and explanations. If you are new to Calc and you have some time, try going through at least the beginning of the tutorial. The tutorial includes about 70 exercises with answers. These exercises give you some guided practice with Calc, as well as pointing out some interesting and unusual ways to use its features.

The reference section discusses Calc in complete depth. You can read the reference from start to finish if you want to learn every aspect of Calc. Or, you can look in the table of contents or the Concept Index to find the parts of the manual that discuss the things you need to know.

Every Calc keyboard command is listed in the Calc Summary, and also in the Key Index. Algebraic functions, M-x commands, and variables also have their own indices. @c{Each} In the printed manual, each paragraph that is referenced in the Key or Function Index is marked in the margin with its index entry.

You can access this manual on-line at any time within Calc by pressing the h i key sequence. Outside of the Calc window, you can press M-# i to read the manual on-line. Also, you can jump directly to the Tutorial by pressing h t or M-# t, or to the Summary by pressing h s or M-# s. Within Calc, you can also go to the part of the manual describing any Calc key, function, or variable using h k, h f, or h v, respectively. See section Help Commands.

Printed copies of this manual are also available from the Free Software Foundation.

Notations Used in This Manual

This section describes the various notations that are used throughout the Calc manual.

In keystroke sequences, uppercase letters mean you must hold down the shift key while typing the letter. Keys pressed with Control held down are shown as C-x. Keys pressed with Meta held down are shown as M-x. Other notations are RET for the Return key, SPC for the space bar, TAB for the Tab key, DEL for the Delete key, and LFD for the Line-Feed key.

(If you don't have the LFD or TAB keys on your keyboard, the C-j and C-i keys are equivalent to them, respectively. If you don't have a Meta key, look for Alt or Extend Char. You can also press ESC or C-[first to get the same effect, so that M-x, ESC x, and C-[x are all equivalent.)

Sometimes the RET key is not shown when it is "obvious" that you must press RET to proceed. For example, the RET is usually omitted in key sequences like M-x calc-keypad RET.

Commands are generally shown like this: p (`calc-precision`) or M-# k (`calc-keypad`). This means that the command is normally used by pressing the p key or M-# k key sequence, but it also has the full-name equivalent shown, e.g., M-x calc-precision.

Commands that correspond to functions in algebraic notation are written: C (`calc-cos`) [`cos`]. This means the C key is equivalent to M-x calc-cos, and that the corresponding function in an algebraic-style formula would be `cos(x)'.

A few commands don't have key equivalents: `calc-sincos` [`sincos`].

A Demonstration of Calc

This section will show some typical small problems being solved with Calc. The focus is more on demonstration than explanation, but everything you see here will be covered more thoroughly in the Tutorial.

To begin, start Emacs if necessary (usually the command `emacs` does this), and type M-# c (or ESC # c) to start the Calculator. (See section Starting Calc, if this doesn't work for you.)

Be sure to type all the sample input exactly, especially noting the difference between lower-case and upper-case letters. Remember, RET, TAB, DEL, and SPC are the Return, Tab, Delete, and Space keys.

RPN calculation. In RPN, you type the input number(s) first, then the command to operate on the numbers.

Type 2 RET 3 + Q to compute @c{$\sqrt{2+3} = 2.2360679775$} the square root of 2+3, which is 2.2360679775.

Type P 2 ^ to compute @c{$\pi^2 = 9.86960440109$} the value of `pi' squared, 9.86960440109.

Type TAB to exchange the order of these two results.

Type - I H S to subtract these results and compute the Inverse Hyperbolic sine of the difference, 2.72996136574.

Type DEL to erase this result.

Algebraic calculation. You can also enter calculations using conventional "algebraic" notation. To enter an algebraic formula, use the apostrophe key.

Type ' sqrt(2+3) RET to compute @c{$\sqrt{2+3}$} the square root of 2+3.

Type ' pi^2 RET to enter @c{π^2} `pi' squared. To evaluate this symbolic formula as a number, type =.

Type ' arcsinh($ - $$) RET to subtract the second-most-recent result from the most-recent and compute the Inverse Hyperbolic sine.

Keypad mode. If you are using the X window system, press M-# k to get Keypad mode. (If you don't use X, skip to the next section.)

Click on the 2, ENTER, 3, +, and SQRT "buttons" using your left mouse button.

Click on PI, 2, and `y^x`.

Click on INV, then ENTER to swap the two results.

Click on -, INV, HYP, and SIN.

Click on <- to erase the result, then click OFF to turn the Keypad Calculator off.

Grabbing data. Type M-# x if necessary to exit Calc. Now select the following numbers as an Emacs region: "Mark" the front of the list by typing control-SPC or control-@ there, then move to the other end of the list. (Either get this list from the on-line copy of this manual, accessed by M-# i, or just type these

numbers into a scratch file.) Now type M-# g to "grab" these numbers into Calc.

```
1.23  1.97
1.6   2
1.19  1.08
```

The result `[1.23, 1.97, 1.6, 2, 1.19, 1.08]' is a Calc "vector." Type V R + to compute the sum of these numbers.

Type U to Undo this command, then type V R * to compute the product of the numbers.

You can also grab data as a rectangular matrix. Place the cursor on the upper-leftmost `1' and set the mark, then move to just after the lower-right `8' and press M-# r.

Type v t to transpose this @c{3×2} 3x2 matrix into a @c{2×3} 2x3 matrix. Type v u to unpack the rows into two separate vectors. Now type V R + TAB V R + to compute the sums of the two original columns. (There is also a special grab-and-sum-columns command, M-# :.)

Units conversion. Units are entered algebraically. Type ' 43 mi/hr RET to enter the quantity 43 miles-per-hour. Type u c km/hr RET. Type u c m/s RET.

Date arithmetic. Type t N to get the current date and time. Type 90 + to find the date 90 days from now. Type ' <25 dec 87> RET to enter a date, then - 7 / to see how many weeks have passed since then.

Algebra. Algebraic entries can also include formulas or equations involving variables. Type ' [x + y = a, x y = 1] RET to enter a pair of equations involving three variables. (Note the leading apostrophe in this example; also, note that the space between `x y' is required.) Type a S x,y RET to solve these equations for the variables x and y.

Type d B to view the solutions in more readable notation. Type d C to view them in C language notation, and d T to view them in the notation for the TeX typesetting system. Type d N to return to normal notation.

Type 7.5, then s l a RET to let a = 7.5 in these formulas. (That's a letter l, not a numeral 1.)

Help functions. You can read about any command in the on-line manual. Type M-# c to return to Calc after each of these commands: h k t N to read about the t N command, h f sqrt RET to read about the `sqrt` function, and h s to read the Calc summary.

Press DEL repeatedly to remove any leftover results from the stack. To exit from Calc, press q or M-# c again.

Using Calc

Calc has several user interfaces that are specialized for different kinds of tasks. As well as Calc's standard interface, there are Quick Mode, Keypad Mode, and Embedded Mode.

Calc must be installed before it can be used. See section Installation, for instructions on setting up and installing Calc. We will assume you or someone on your system has already installed Calc as described there.

Starting Calc

On most systems, you can type M-# to start the Calculator. The notation M-# is short for Meta-#. On most keyboards this means holding down the Meta (or Alt) and Shift keys while typing 3.

Once again, if you don't have a Meta key on your keyboard you can type ESC first, then #, to accomplish the same thing. If you don't even have an ESC key, you can fake it by holding down Control or CTRL while typing a left square bracket (that's C-[in Emacs notation).

M-# is a prefix key; when you press it, Emacs waits for you to press a second key to complete the command. In this case, you will follow M-# with a letter (upper- or lower-case, it doesn't matter for M-#) that says which Calc interface you want to use.

To get Calc's standard interface, type M-# c. To get Keypad Mode, type M-# k. Type M-# ? to get a brief list of the available options, and type a second ? to get a complete list.

To ease typing, M-# M-# (or M-# # if that's easier) also works to start Calc. It starts the same interface (either M-# c or M-# k) that you last used, selecting the M-# c interface by default. (If your installation has a special function key set up to act like M-#, hitting that function key twice is just like hitting M-# M-#.)

If M-# doesn't work for you, you can always type explicit commands like M-x calc (for the standard user interface) or M-x calc-keypad (for Keypad Mode). First type M-x (that's Meta with the letter x), then, at the prompt, type the full command (like calc-keypad) and press Return.

If you type M-x calc and Emacs still doesn't recognize the command (it will say `[No match]' when you try to press RET), then Calc has not been properly installed.

The same commands (like M-# c or M-# M-#) that start the Calculator also turn it off if it is already on.

The Standard Calc Interface

Calc's standard interface acts like a traditional RPN calculator, operated by the normal Emacs keyboard. When you type M-# c to start the Calculator, the Emacs screen splits into two windows with the file you were editing on top and Calc on the bottom.

@advance@hsize20pt

```
...
--**-Emacs: myfile             (Fundamental)----All----------------------
--- Emacs Calculator Mode --                      |Emacs Calc Mode v2.00...
2:  17.3                                          |    17.3
1:  -5                                            |    3
    .                                             |    2
                                                  |    4
                                                  |  * 8
                                                  |  ->-5
                                                  |
--%%-Calc: 12 Deg       (Calculator)----All-----  --%%-Emacs: *Calc Trail*
```

In this figure, the mode-line for `myfile' has moved up and the "Calculator" window has appeared b it. As you can see, Calc actually makes two windows side-by-side. The lefthand one is called the stack window and the righthand one is called the trail window. The stack holds the numbers involved in the calculation you are currently performing. The trail holds a complete record of all calculations you have done. In a desk calculator with a printer, the trail corresponds to the paper tape that records what you do.

In this case, the trail shows that four numbers (17.3, 3, 2, and 4) were first entered into the Calculator, then the 2 and 4 were multiplied to get 8, then the 3 and 8 were subtracted to get -5. (The `>' symbol shows that this was the most recent calculation.) The net result is the two numbers 17.3 and -5 sitting on the stack.

Most Calculator commands deal explicitly with the stack only, but there is a set of commands that allow you to search back through the trail and retrieve any previous result.

Calc commands use the digits, letters, and punctuation keys. Shifted (i.e., upper-case) letters are different from lowercase letters. Some letters are prefix keys that begin two-letter commands. For example, e means "enter exponent" and shifted E means e^x. With the d ("display modes") prefix the letter "e" takes on very different meanings: d e means "engineering notation" and d E means "eqn language mode."

There is nothing stopping you from switching out of the Calc window and back into your editing window, say by using the Emacs C-x o (`other-window`) command. When the cursor is inside a regular window, Emacs acts just like normal. When the cursor is in the Calc stack or trail windows, keys are interpreted as Calc commands.

When you quit by pressing M-# c a second time, the Calculator windows go away but the actual Stack and Trail are not gone, just hidden. When you press M-# c once again you will get the same stack and trail contents you had when you last used the Calculator.

The Calculator does not remember its state between Emacs sessions. Thus if you quit Emacs and start it again, M-# c will give you a fresh stack and trail. There is a command (m m) that lets you save your favorite mode settings between sessions, though. One of the things it saves is which user interface (standard or Keypad) you last used; otherwise, a freshly started Emacs will always treat M-# M-# the same as M-# c.

The q key is another equivalent way to turn the Calculator off.

If you type M-# b first and then M-# c, you get a full-screen version of Calc (`full-calc`) in which the stack and trail windows are still side-by-side but are now as tall as the whole Emacs screen. When you press q or M-# c again to quit, the file you were editing before reappears. The M-# b key switches back and forth between "big" full-screen mode and the normal partial-screen mode.

Finally, M-# o (`calc-other-window`) is like M-# c except that the Calc window is not selected. The buffer you were editing before remains selected instead. M-# o is a handy way to switch out of Calc momentarily to edit your file; type M-# c to switch back into Calc when you are done.

Quick Mode (Overview)

Quick Mode is a quick way to use Calc when you don't need the full complexity of the stack and trail. To use it, type M-# q (`quick-calc`) in any regular editing buffer.

Quick Mode is very simple: It prompts you to type any formula in standard algebraic notation (like `4 - 2/3') and then displays the result at the bottom of the Emacs screen (*3.33333333333* in this case). You are then back in the same editing buffer you were in before, ready to continue editing or to type M-# q again to do

another quick calculation. The result of the calculation will also be in the Emacs "kill ring" so that a C-y command at this point will yank the result into your editing buffer.

Calc mode settings affect Quick Mode, too, though you will have to go into regular Calc (with M-# c) to change the mode settings.

See section "Quick Calculator" Mode, for further information.

Keypad Mode (Overview)

Keypad Mode is a mouse-based interface to the Calculator. It is designed for use with the X window system. If you don't have X, you will have to operate keypad mode with your arrow keys (which is probably more trouble than it's worth).

Type M-# k to turn Keypad Mode on or off. Once again you get two new windows, this time on the righthand side of the screen instead of at the bottom. The upper window is the familiar Calc Stack; the lower window is a picture of a typical calculator keypad.

```
|--- Emacs Calculator Mode ---
|2:  17.3
|1:  -5
|    .
|--%%-Calc: 12 Deg       (Calcul
|----+-----Calc 2.00-----+----1
|FLR |CEIL|RND |TRNC|CLN2|FLT |
|----+----+----+----+----+----|
| LN |EXP |    |ABS |IDIV|MOD |
|----+----+----+----+----+----|
|SIN |COS |TAN |SQRT|y^x |1/x |
|----+----+----+----+----+----|
|  ENTER  |+/- |EEX |UNDO| <- |
|-----+---+-+--+--+-+---++----|
| INV |  7  |  8  |  9  |  /  |
|-----+-----+-----+-----+-----|
| HYP |  4  |  5  |  6  |  *  |
|-----+-----+-----+-----+-----|
|EXEC |  1  |  2  |  3  |  -  |
|-----+-----+-----+-----+-----|
| OFF |  0  |  .  | PI  |  +  |
|-----+-----+-----+-----+-----+
```

@begingroup @ifdim@hsize=5in @advance@hsize-2.2in @else @advance@hsize-3.05in @advance@vsize.1in @fi

Keypad Mode is much easier for beginners to learn, because there is no need to memorize lots of obscure key sequences. But not all commands in regular Calc are available on the Keypad. You can always switch the cursor into the Calc stack window to use standard Calc commands if you need. Serious Calc users, though, often find they prefer the standard interface over Keypad Mode.

To operate the Calculator, just click on the "buttons" of the keypad using your left mouse button. To en the two numbers shown here you would click 1 7 . 3 ENTER 5 +/- ENTER; to add them together you w then click + (to get 12.3 on the stack).

If you click the right mouse button, the top three rows of the keypad change to show other sets of commands, such as advanced math functions, vector operations, and operations on binary numbers.

@endgroup Because Keypad Mode doesn't use the regular keyboard, Calc leaves the cursor in your original editing buffer. You can type in this buffer in the usual way while also clicking on the Calculator keypad. One advantage of Keypad Mode is that you don't need an explicit command to switch between editing and calculating.

If you press M-# b first, you get a full-screen Keypad Mode (`full-calc-keypad`) with three windows: The keypad in the lower left, the stack in the lower right, and the trail on top.

See section "Keypad" Mode, for further information.

Standalone Operation

If you are not in Emacs at the moment but you wish to use Calc, you must start Emacs first. If all you want is to run Calc, you can give the commands:

```
emacs -f full-calc
```

or

```
emacs -f full-calc-keypad
```

which run a full-screen Calculator (as if by M-# b M-# c) or a full-screen X-based Calculator (as if by M-# b M-# k). In standalone operation, quitting the Calculator (by pressing q or clicking on the keypad EXIT button) quits Emacs itself.

Embedded Mode (Overview)

Embedded Mode is a way to use Calc directly from inside an editing buffer. Suppose you have a formula written as part of a document like this:

```
The derivative of

                                    ln(ln(x))

is
```

and you wish to have Calc compute and format the derivative for you and store this derivative in the buffer automatically. To do this with Embedded Mode, first copy the formula down to where you want the result to be:

```
The derivative of
```


```
                                ln(ln(x))
```

is

```
                                ln(ln(x))
```

Now, move the cursor onto this new formula and press M-# e. Calc will read the formula (using the surrounding blank lines to tell how much text to read), then push this formula (invisibly) onto the Calc stack. The cursor will stay on the formula in the editing buffer, but the buffer's mode line will change to look like the Calc mode line (with mode indicators like `12 Deg' and so on). Even though you are still in your editing buffer, the keyboard now acts like the Calc keyboard, and any new result you get is copied from the stack back into the buffer. To take the derivative, you would type a d x RET.

```
The derivative of

                                ln(ln(x))

is

1 / ln(x) x
```

To make this look nicer, you might want to press d = to center the formula, and even d B to use "big" display mode.

```
The derivative of

                                ln(ln(x))

is
% [calc-mode: justify: center]
% [calc-mode: language: big]

                                   1
                                -------
                                ln(x) x
```

Calc has added annotations to the file to help it remember the modes that were used for this formula. They are formatted like comments in the TeX typesetting language, just in case you are using TeX. (In this example TeX is not being used, so you might want to move these comments up to the top of the file or otherwise put them out of the way.)

As an extra flourish, we can add an equation number using a righthand label: Type d } (1) RET.

```
% [calc-mode: justify: center]
% [calc-mode: language: big]
% [calc-mode: right-label: " (1)"]
```

$$\frac{1}{\ln(x)\ x} \qquad (1)$$

TOC

To leave Embedded Mode, type M-# e again. The mode line and keyboard will revert to the way they were before. (If you have actually been trying this as you read along, you'll want to press M-# 0 [with the digit zero] now to reset the modes you changed.)

The related command M-# w operates on a single word, which generally means a single number, inside text. It uses any non-numeric characters rather than blank lines to delimit the formula it reads. Here's an example of its use:

```
A slope of one-third corresponds to an angle of 1 degrees.
```

Place the cursor on the `1', then type M-# w to enable Embedded Mode on that number. Now type 3 / (to get one-third), and I T (the Inverse Tangent converts a slope into an angle), then M-# w again to exit Embedded mode.

```
A slope of one-third corresponds to an angle of 18.4349488229 degrees.
```

See section Embedded Mode, for full details.

Other M-# Commands

Two more Calc-related commands are M-# g and M-# r, which "grab" data from a selected region of a buffer into the Calculator. The region is defined in the usual Emacs way, by a "mark" placed at one end of the region, and the Emacs cursor or "point" placed at the other.

The M-# g command reads the region in the usual left-to-right, top-to-bottom order. The result is packaged into a Calc vector of numbers and placed on the stack. Calc (in its standard user interface) is then started. Type v u if you want to unpack this vector into separate numbers on the stack. Also, C-u M-# g interprets the region as a single number or formula.

The M-# r command reads a rectangle, with the point and mark defining opposite corners of the rectangle. The result is a matrix of numbers on the Calculator stack.

Complementary to these is M-# y, which "yanks" the value at the top of the Calc stack back into an editing buffer. If you type M-# y while in such a buffer, the value is yanked at the current position. If you type M-# y while in the Calc buffer, Calc makes an educated guess as to which editing buffer you want to use. The Calc window does not have to be visible in order to use this command, as long as there is something on the Calc stack.

Here, for reference, is the complete list of M-# commands. The shift, control, and meta keys are ignored for the keystroke following M-#.

Commands for turning Calc on and off:

#

Turn Calc on or off, employing the same user interface as last time.

C

Turn Calc on or off using its standard bottom-of-the-screen interface. If Calc is already turned on but the cursor is not in the Calc window, move the cursor into the window.

O

Same as C, but don't select the new Calc window. If Calc is already turned on and the cursor is in the Calc window, move it out of that window.

B

Control whether M-# c and M-# k use the full screen.

Q

Use Quick Mode for a single short calculation.

K

Turn Calc Keypad mode on or off.

E

Turn Calc Embedded mode on or off at the current formula.

J

Turn Calc Embedded mode on or off, select the interesting part.

W

Turn Calc Embedded mode on or off at the current word (number).

Z

Turn Calc on in a user-defined way, as defined by a Z I command.

X

Quit Calc; turn off standard, Keypad, or Embedded mode if on. (This is like q or OFF inside of Calc.)

Commands for moving data into and out of the Calculator:

G

Grab the region into the Calculator as a vector.

R

Grab the rectangular region into the Calculator as a matrix.

:

Grab the rectangular region and compute the sums of its columns.

_

Grab the rectangular region and compute the sums of its rows.

Y

Yank a value from the Calculator into the current editing buffer.

Commands for use with Embedded Mode:

A

"Activate" the current buffer. Locate all formulas that contain `:=' or `=>' symbols and record their locations so that they can be updated automatically as variables are changed.

D

Duplicate the current formula immediately below and select the duplicate.

F
: Insert a new formula at the current point.

N
: Move the cursor to the next active formula in the buffer.

P
: Move the cursor to the previous active formula in the buffer.

U
: Update (i.e., as if by the = key) the formula at the current point.

`
: Edit (as if by `calc-edit`) the formula at the current point.

Miscellaneous commands:

I
: Run the Emacs Info system to read the Calc manual. (This is the same as h i inside of Calc.)

T
: Run the Emacs Info system to read the Calc Tutorial.

S
: Run the Emacs Info system to read the Calc Summary.

L
: Load Calc entirely into memory. (Normally the various parts are loaded only as they are needed.)

M
: Read a region of written keystroke names (like `C-n a b c RET') and record them as the current keyboard macro.

0
: (This is the "zero" digit key.) Reset the Calculator to its default state: Empty stack, and default mode settings. With any prefix argument, reset everything but the stack.

History and Acknowledgements

Calc was originally started as a two-week project to occupy a lull in the author's schedule. Basically, a friend asked if I remembered the value of @c{2^{32}} 2^32. I didn't offhand, but I said, "that's easy, just call up an `xcalc`." `Xcalc` duly reported that the answer to our question was `4.294967e+09'---with no way to see the full ten digits even though we knew they were there in the program's memory! I was so annoyed, I vowed to write a calculator of my own, once and for all.

I chose Emacs Lisp, a) because I had always been curious about it and b) because, being only a text editor extension language after all, Emacs Lisp would surely reach its limits long before the project got too far out of hand.

To make a long story short, Emacs Lisp turned out to be a distressingly solid implementation of Lisp, and the humble task of calculating turned out to be more open-ended than one might have expected.

Emacs Lisp doesn't have built-in floating point math, so it had to be simulated in software. In fact, Emacs

integers will only comfortably fit six decimal digits or so--not enough for a decent calculator. So I had to write my own high-precision integer code as well, and once I had this I figured that arbitrary-size integers were just as easy as large integers. Arbitrary floating-point precision was the logical next step. Also, since the large integer arithmetic was there anyway it seemed only fair to give the user direct access to it, which in turn made it practical to support fractions as well as floats. All these features inspired me to look around for other data types that might be worth having.

Around this time, my friend Rick Koshi showed me his nifty new HP-28 calculator. It allowed the user to manipulate formulas as well as numerical quantities, and it could also operate on matrices. I decided that these would be good for Calc to have, too. And once things had gone this far, I figured I might as well take a look at serious algebra systems like Mathematica, Macsyma, and Maple for further ideas. Since these systems did far more than I could ever hope to implement, I decided to focus on rewrite rules and other programming features so that users could implement what they needed for themselves.

Rick complained that matrices were hard to read, so I put in code to format them in a 2D style. Once these routines were in place, Big mode was obligatory. Gee, what other language modes would be useful?

Scott Hemphill and Allen Knutson, two friends with a strong mathematical bent, contributed ideas and algorithms for a number of Calc features including modulo forms, primality testing, and float-to-fraction conversion.

Units were added at the eager insistence of Mass Sivilotti. Later, Ulrich Mueller at CERN and Przemek Klosowski at NIST provided invaluable expert assistance with the units table. As far as I can remember, the idea of using algebraic formulas and variables to represent units dates back to an ancient article in Byte magazine about muMath, an early algebra system for microcomputers.

Many people have contributed to Calc by reporting bugs and suggesting features, large and small. A few deserve special mention: Tim Peters, who helped develop the ideas that led to the selection commands, rewrite rules, and many other algebra features; @c{Fran\c cois} Francois Pinard, who contributed an early prototype of the Calc Summary appendix as well as providing valuable suggestions in many other areas of Calc; Carl Witty, whose eagle eyes discovered many typographical and factual errors in the Calc manual; Tim Kay, who drove the development of Embedded mode; Ove Ewerlid, who made many suggestions relating to the algebra commands and contributed some code for polynomial operations; Randal Schwartz, who suggested the `calc-eval` function; Robert J. Chassell, who suggested the Calc Tutorial and exercises; and Juha Sarlin, who first worked out how to split Calc into quickly-loading parts.

Among the books used in the development of Calc were Knuth's *Art of Computer Programming* (especially volume II, *Seminumerical Algorithms*); *Numerical Recipes* by Press, Flannery, Teukolsky, and Vetterling; Bevington's *Data Reduction and Error Analysis for the Physical Sciences*; *Concrete Mathematics* by Graham, Knuth, and Patashnik; Steele's *Common Lisp, the Language*; the *CRC Standard Math Tables* (William H. Beyer, ed.); and Abramowitz and Stegun's venerable *Handbook of Mathematical Functions*. I consulted the user's manuals for the HP-28 and HP-48 calculators, as well as for the programs Mathematica, SMP, Macsyma, Maple, MathCAD, Gnuplot, and others. Also, of course, Calc could not have been written without the excellent *GNU Emacs Lisp Reference Manual*, by Bil Lewis and Dan LaLiberte.

Final thanks go to Richard Stallman, without whose fine implementations of the Emacs editor, language, and environment, Calc would have been finished in two weeks.

Go to the previous, next section.

Go to the previous, next section.

Tutorial

This chapter explains how to use Calc and its many features, in a step-by-step, tutorial way. You are encouraged to run Calc and work along with the examples as you read (see section Starting Calc). If you are already familiar with advanced calculators, you may wish to skip on to the rest of this manual.

This tutorial describes the standard user interface of Calc only. The "Quick Mode" and "Keypad Mode" interfaces are fairly self-explanatory. See section Embedded Mode, for a description of the "Embedded Mode" interface.

The easiest way to read this tutorial on-line is to have two windows on your Emacs screen, one with Calc and one with the Info system. (If you have a printed copy of the manual you can use that instead.) Press M-# c to turn Calc on or to switch into the Calc window, and press M-# i to start the Info system or to switch into its window.

This tutorial is designed to be done in sequence. But the rest of this manual does not assume you have gone through the tutorial. The tutorial does not cover everything in the Calculator, but it touches on most general areas.

The Calc Summary at the end of the reference manual includes some blank space for your own use. You may wish to keep notes there as you learn Calc.

Basic Tutorial

In this section, we learn how RPN and algebraic-style calculations work, how to undo and redo an operation done by mistake, and how to control various modes of the Calculator.

RPN Calculations and the Stack

The central component of an RPN calculator is the stack. A calculator stack is like a stack of dishes. New dishes (numbers) are added at the top of the stack, and numbers are normally only removed from the top of the stack.

In an operation like 2+3, the 2 and 3 are called the operands and the + is the operator. In an RPN calculator you always enter the operands first, then the operator. Each time you type a number, Calc adds or pushes it onto the top of the Stack. When you press an operator key like +, Calc pops the appropriate number of operands from the stack and pushes back the result.

Thus we could add the numbers 2 and 3 in an RPN calculator by typing: 2 RET 3 RET +. (The RET key, Return, corresponds to the ENTER key on traditional RPN calculators.) Try this now if you wish; type M-# c to switch into the Calc window (you can type M-# c again or M-# o to switch back to the Tutorial window). The first four keystrokes "push" the numbers 2 and 3 onto the stack. The + key "pops" the top two numbers from the stack, adds them, and pushes the result (5) back onto the stack. Here's how the stack will look at various points throughout the calculation:


```
    .          1:  2          2:  2          1:  5              .
                   .          1:  3              .
                                  .

  M-# c          2 RET          3 RET          +              DEL
```

The `.' symbol is a marker that represents the top of the stack. Note that the "top" of the stack is really shown at the bottom of the Stack window. This may seem backwards, but it turns out to be less distracting in regular use.

The numbers `1:' and `2:' on the left are stack level numbers. Old RPN calculators always had four stack levels called x, y, z, and t. Calc's stack can grow as large as you like, so it uses numbers instead of letters. Some stack-manipulation commands accept a numeric argument that says which stack level to work on. Normal commands like + always work on the top few levels of the stack.

The Stack buffer is just an Emacs buffer, and you can move around in it using the regular Emacs motion commands. But no matter where the cursor is, even if you have scrolled the `.' marker out of view, most Calc commands always move the cursor back down to level 1 before doing anything. It is possible to move the `.' marker upwards through the stack, temporarily "hiding" some numbers from commands like +. This is called stack truncation and we will not cover it in this tutorial; see section Truncating the Stack, if you are interested.

You don't really need the second RET in 2 RET 3 RET +. That's because if you type any operator name or other non-numeric key when you are entering a number, the Calculator automatically enters that number and then does the requested command. Thus 2 RET 3 + will work just as well.

Examples in this tutorial will often omit RET even when the stack displays shown would only happen if you did press RET:

```
1:  2          2:  2          1:  5
    .          1:  3              .
                   .

  2 RET            3              +
```

Here, after pressing 3 the stack would really show `1: 2' with `Calc: 3' in the minibuffer. In these situations, you can press the optional RET to see the stack as the figure shows.

(*) **Exercise 1.** (This tutorial will include exercises at various points. Try them if you wish. Answers to all the exercises are located at the end of the Tutorial chapter. Each exercise will include a cross-reference to its particular answer. If you are reading with the Emacs Info system, press f and the exercise number to go to the answer, then the letter l to return to where you were.)

Here's the first exercise: What will the keystrokes 1 RET 2 RET 3 RET 4 + * - compute? (`*' is the symbol for multiplication.) Figure it out by hand, then try it with Calc to see if you're right. See section RPN Tutorial Exercise 1. (*)

(*) **Exercise 2.** Compute @c{$(2\times4) + (7\times9.4) + {5\over4}$} 2*4 + 7*9.5 + 5/4 using the stack. See section RPN Tutorial Exercise 2. (*)

The DEL key is called Backspace on some keyboards. It is whatever key you would use to correct a simple typing error when regularly using Emacs. The DEL key pops and throws away the top value on the stack.

(You can still get that value back from the Trail if you should need it later on.) There are many places in t tutorial where we assume you have used DEL to erase the results of the previous example at the beginnin new example. In the few places where it is really important to use DEL to clear away old results, the text wi remind you to do so.

(It won't hurt to let things accumulate on the stack, except that whenever you give a display-mode-changing command Calc will have to spend a long time reformatting such a large stack.)

Since the - key is also an operator (it subtracts the top two stack elements), how does one enter a negative number? Calc uses the _ (underscore) key to act like the minus sign in a number. So, typing -5 RET won't work because the - key will try to do a subtraction, but _5 RET works just fine.

You can also press n, which means "change sign." It changes the number at the top of the stack (or the number being entered) from positive to negative or vice-versa: 5 n RET.

If you press RET when you're not entering a number, the effect is to duplicate the top number on the stack. Consider this calculation:

```
1:  3          2:  3          1:  9          2:  9          1:  81
    .          1:  3              .          1:  9              .
                   .                             .

  3 RET            RET              *              RET              *
```

(Of course, an easier way to do this would be 3 RET 4 ^, to raise 3 to the fourth power.)

The space-bar key (denoted SPC here) performs the same function as RET; you could replace all three occurrences of RET in the above example with SPC and the effect would be the same.

Another stack manipulation key is TAB. This exchanges the top two stack entries. Suppose you have computed 2 RET 3 + to get 5, and then you realize what you really wanted to compute was 20 / (2+3).

```
1:  5          2:  5          2:  20         1:  4
    .          1:  20         1:  5              .
                   .                 .

 2 RET 3 +         20             TAB              /
```

Planning ahead, the calculation would have gone like this:

```
1:  20         2:  20         3:  20         2:  20         1:  4
    .          1:  2          2:  2          1:  5              .
                   .          1:  3              .
                                 .

  20 RET          2 RET            3              +              /
```

A related stack command is M-TAB (hold META and type TAB). It rotates the top three elements of the stack upward, bringing the object in level 3 to the top.


```
1:  10          2:  10          3:  10          3:  20          3:  30
    .           1:  20          2:  20          2:  30          2:  10
                    .           1:  30          1:  10          1:  20
                                    .               .               .

  10 RET          20 RET          30 RET          M-TAB           M-TAB
```

(*) **Exercise 3.** Suppose the numbers 10, 20, and 30 are on the stack. Figure out how to add one to the number in level 2 without affecting the rest of the stack. Also figure out how to add one to the number in level 3. See section RPN Tutorial Exercise 3. (*)

Operations like +, -, *, /, and ^ pop two arguments from the stack and push a result. Operations like n and Q (square root) pop a single number and push the result. You can think of them as simply operating on the top element of the stack.

```
1:  3           1:  9           2:  9           1:  25          1:  5
    .               .           1:  16              .               .
                                    .

  3 RET           RET *           4 RET RET *         +               Q
```

(Note that capital Q means to hold down the Shift key while typing q. Remember, plain unshifted q is the Quit command.)

Here we've used the Pythagorean Theorem to determine the hypotenuse of a right triangle. Calc actually has a built-in command for that called f h, but let's suppose we can't remember the necessary keystrokes. We can still enter it by its full name using M-x notation:

```
1:  3           2:  3           1:  5
    .           1:  4               .
                    .

  3 RET           4 RET       M-x calc-hypot
```

All Calculator commands begin with the word `calc-'. Since it gets tiring to type this, Calc provides an x key which is just like the regular Emacs M-x key except that it types the `calc-' prefix for you:

```
1:  3           2:  3           1:  5
    .           1:  4               .
                    .

  3 RET           4 RET          x hypot
```

What happens if you take the square root of a negative number?

```
1:  4           1:  -4          1:  (0, 2)
    .               .               .

  4 RET             n               Q
```

The notation (a, b) represents a complex number. Complex numbers are more traditionally written @c{$a i$} a + b i; Calc can display in this format, too, but for now we'll stick to the (a, b) notation.

If you don't know how complex numbers work, you can safely ignore this feature. Complex numbers only arise from operations that would be errors in a calculator that didn't have complex numbers. (For example, taking the square root or logarithm of a negative number produces a complex result.)

Complex numbers are entered in the notation shown. The (and , and) keys manipulate "incomplete complex numbers."

```
1:  ( ...      2:  ( ...      1:  (2, ...    1:  (2, ...    1:  (2, 3)
    .          1:  2              .              3              .
                   .                             .

    (              2              ,              3              )
```

You can perform calculations while entering parts of incomplete objects. However, an incomplete object cannot actually participate in a calculation:

```
1:  ( ...      2:  ( ...      3:  ( ...      1:  ( ...      1:  ( ...
    .          1:  2          2:  2              5              5
                   .          1:  3              .              .
                                  .
                                                             (error)
    (              2 RET          3              +              +
```

Adding 5 to an incomplete object makes no sense, so the last command produces an error message and leaves the stack the same.

Incomplete objects can't participate in arithmetic, but they can be moved around by the regular stack commands.

```
2:  2          3:  2          3:  3          1:  ( ...      1:  (2, 3)
1:  3          2:  3          2:  ( ...          2              .
    .          1:  ( ...      1:  2              3
                   .              .              .

2 RET 3 RET        (              M-TAB          M-TAB          )
```

Note that the , (comma) key did not have to be used here. When you press) all the stack entries between the incomplete entry and the top are collected, so there's never really a reason to use the comma. It's up to you.

(*) **Exercise 4.** To enter the complex number (2, 3), your friend Joe typed (2 , SPC 3). What happened? (Joe thought of a clever way to correct his mistake in only two keystrokes, but it didn't quite work. Try it to find out why.) See section RPN Tutorial Exercise 4. (*)

Vectors are entered the same way as complex numbers, but with square brackets in place of parentheses. We'll meet vectors again later in the tutorial.

Any Emacs command can be given a numeric prefix argument by typing a series of META-digits beforehand.

If META is awkward for you, you can instead type C-u followed by the necessary digits. Numeric prefix arguments can be negative, as in M-- M-3 M-5 or C-u - 3 5. Calc commands use numeric prefix arguments in a variety of ways. For example, a numeric prefix on the + operator adds any number of stack entries at once:

```
1:  10          2:  10          3:  10          3:  10          1:  60
    .           1:  20          2:  20          2:  20              .
                    .           1:  30          1:  30
                                    .               .

  10 RET          20 RET          30 RET          C-u 3           +
```

For stack manipulation commands like RET, a positive numeric prefix argument operates on the top n stack entries at once. A negative argument operates on the entry in level n only. An argument of zero operates on the entire stack. In this example, we copy the second-to-top element of the stack:

```
1:  10          2:  10          3:  10          3:  10          4:  10
    .           1:  20          2:  20          2:  20          3:  20
                    .           1:  30          1:  30          2:  30
                                    .               .           1:  20
                                                                    .

  10 RET          20 RET          30 RET          C-u -2          RET
```

Another common idiom is M-0 DEL, which clears the stack. (The M-0 numeric prefix tells DEL to operate on the entire stack.)

Algebraic-Style Calculations

If you are not used to RPN notation, you may prefer to operate the Calculator in "algebraic mode," which is closer to the way non-RPN calculators work. In algebraic mode, you enter formulas in traditional 2+3 notation.

You don't really need any special "mode" to enter algebraic formulas. You can enter a formula at any time by pressing the apostrophe (') key. Answer the prompt with the desired formula, then press RET. The formula is evaluated and the result is pushed onto the RPN stack. If you don't want to think in RPN at all, you can enter your whole computation as a formula, read the result from the stack, then press DEL to delete it from the stack.

Try pressing the apostrophe key, then 2+3+4, then RET. The result should be the number 9.

Algebraic formulas use the operators `+', `-', `*', `/', and `^'. You can use parentheses to make the order of evaluation clear. In the absence of parentheses, `^' is evaluated first, then `*', then `/', then finally `+' and `-'. For example, the expression

```
2 + 3*4*5 / 6*7^8 - 9
```

is equivalent to

```
2 + ((3*4*5) / (6*(7^8)) - 9
```

or, in large mathematical notation,

The result of this expression will be the number *-6.99999826533*.

Calc's order of evaluation is the same as for most computer languages, except that `*' binds more strongly than `/', as the above example shows. As in normal mathematical notation, the `*' symbol can often be omitted: `2 a' is the same as `2*a'.

Operators at the same level are evaluated from left to right, except that `^' is evaluated from right to left. Thus, `2-3-4' is equivalent to `(2-3)-4' or *-5*, whereas `2^3^4' is equivalent to `2^(3^4)' (a very large integer; try it!).

If you tire of typing the apostrophe all the time, there is an "algebraic mode" you can select in which Calc automatically senses when you are about to type an algebraic expression. To enter this mode, press the two letters m a. (An `Alg' indicator should appear in the Calc window's mode line.)

Press m a, then 2+3+4 with no apostrophe, then RET.

In algebraic mode, when you press any key that would normally begin entering a number (such as a digit, a decimal point, or the _ key), or if you press (or [, Calc automatically begins an algebraic entry.

Functions which do not have operator symbols like `+' and `*' must be entered in formulas using function-call notation. For example, the function name corresponding to the square-root key Q is `sqrt`. To compute a square root in a formula, you would use the notation `sqrt(x)'.

Press the apostrophe, then type sqrt(5*2) - 3. The result should be 0.16227766017.

Note that if the formula begins with a function name, you need to use the apostrophe even if you are in algebraic mode. If you type arcsin out of the blue, the a r will be taken as an Algebraic Rewrite command, and the csin will be taken as the name of the rewrite rule to use!

Some people prefer to enter complex numbers and vectors in algebraic form because they find RPN entry with incomplete objects to be too distracting, even though they otherwise use Calc as an RPN calculator.

Still in algebraic mode, type:

```
1:  (2, 3)      2:  (2, 3)      1:  (8, -1)     2:  (8, -1)     1:  (9, -1)
    .           1:  (1, -2)         .           1:  1               .
                    .                               .

 (2,3) RET       (1,-2) RET          *               1 RET           +
```

Algebraic mode allows us to enter complex numbers without pressing an apostrophe first, but it also means we need to press RET after every entry, even for a simple number like 1.

(You can type C-u m a to enable a special "incomplete algebraic mode" in which the (and [keys use algebraic entry even though regular numeric keys still use RPN numeric entry. There is also a "total algebraic mode," started by typing m t, in which all normal keys begin algebraic entry. You must then use the META key to type Calc commands: M-m t to get back out of total algebraic mode, M-q to quit, etc.)

If you're still in algebraic mode, press m a again to turn it off.

Actual non-RPN calculators use a mixture of algebraic and RPN styles. In general, operators of two numbers (like + and *) use algebraic form, but operators of one number (like n and Q) use RPN form. Also, a non-RPN

calculator allows you to see the intermediate results of a calculation as you go along. You can accomplish in Calc by performing your calculation as a series of algebraic entries, using the $ sign to tie them together an algebraic formula, $ represents the number on the top of the stack. Here, we perform the calculation @c{$\sqrt{2\times4+1}$} sqrt(2*4+1), which on a traditional calculator would be done by pressing 2 * 4 + 1 = and then the square-root key.

```
1:  8          1:  9          1:  3
    .              .              .

  ' 2*4 RET      $+1 RET        Q
```

Notice that we didn't need to press an apostrophe for the $+1, because the dollar sign always begins an algebraic entry.

(*) **Exercise 1.** How could you get the same effect as pressing Q but using an algebraic entry instead? How about if the Q key on your keyboard were broken? See section Algebraic Entry Tutorial Exercise 1. (*)

The notations $$, $$$, and so on stand for higher stack entries. For example, ' $$+$ RET is just like typing +.

Algebraic formulas can include variables. To store in a variable, press s s, then type the variable name, then press RET. (There are actually two flavors of store command: s s stores a number in a variable but also leaves the number on the stack, while s t removes a number from the stack and stores it in the variable.) A variable name should consist of one or more letters or digits, beginning with a letter.

```
1:  17             .          1:  a + a^2    1:  306
    .                             .              .

    17          s t a RET       ' a+a^2 RET    =
```

The = key evaluates a formula by replacing all its variables by the values that were stored in them.

For RPN calculations, you can recall a variable's value on the stack either by entering its name as a formula and pressing =, or by using the s r command.

```
1:  17         2:  17         3:  17         2:  17         1:  306
    .          1:  17         2:  17         1:  289            .
               .              1:  2              .
                                 .

  s r a RET      ' a RET =      2              ^              +
```

If you press a single digit for a variable name (as in s t 3, you get one of ten quick variables `q0` through `q9`. They are "quick" simply because you don't have to type the letter `q` or the RET after their names. In fact, you can type simply s 3 as a shorthand for s s 3, and likewise for t 3 and r 3.

Any variables in an algebraic formula for which you have not stored values are left alone, even when you evaluate the formula.

```
1:  2 a + 2 b      1:  34 + 2 b
    .                  .
```


```
' 2a+2b RET          =
```

Calls to function names which are undefined in Calc are also left alone, as are calls for which the value is undefined.

```
1:  2 + log10(0) + log10(x) + log10(5, 6) + foo(3)
    .

' log10(100) + log10(0) + log10(x) + log10(5,6) + foo(3) RET
```

In this example, the first call to `log10` works, but the other calls are not evaluated. In the second call, the logarithm is undefined for that value of the argument; in the third, the argument is symbolic, and in the fourth, there are too many arguments. In the fifth case, there is no function called `foo`. You will see a "Wrong number of arguments" message referring to `log10(5,6)'. Press the w ("why") key to see any other messages that may have arisen from the last calculation. In this case you will get "logarithm of zero," then "number expected: `x`". Calc automatically displays the first message only if the message is sufficiently important; for example, Calc considers "wrong number of arguments" and "logarithm of zero" to be important enough to report automatically, while a message like "number expected: `x`" will only show up if you explicitly press the w key.

(*) **Exercise 2.** Joe entered the formula `2 x y', stored 5 in `x`, pressed =, and got the expected result, `10 y'. He then tried the same for the formula `2 x (1+y)', expecting `10 (1+y)', but it didn't work. Why not? See section Algebraic Entry Tutorial Exercise 2. (*)

(*) **Exercise 3.** What result would you expect 1 RET 0 / to give? What if you then type 0 *? See section Algebraic Entry Tutorial Exercise 3. (*)

One interesting way to work with variables is to use the evaluates-to (`=>') operator. It works like this: Enter a formula algebraically in the usual way, but follow the formula with an `=>' symbol. (There is also an s = command which builds an `=>' formula using the stack.) On the stack, you will see two copies of the formula with an `=>' between them. The lefthand formula is exactly like you typed it; the righthand formula has been evaluated as if by typing =.

```
2:  2 + 3 => 5                      2:  2 + 3 => 5
1:  2 a + 2 b => 34 + 2 b           1:  2 a + 2 b => 20 + 2 b
    .                                   .

' 2+3 => RET  ' 2a+2b RET s =          10 s t a RET
```

Notice that the instant we stored a new value in `a`, all `=>' operators already on the stack that referred to a were updated to use the new value. With `=>', you can push a set of formulas on the stack, then change the variables experimentally to see the effects on the formulas' values.

You can also "unstore" a variable when you are through with it:

```
2:  2 + 5 => 5
1:  2 a + 2 b => 2 a + 2 b
    .
```


```
    s u a RET
```

We will encounter formulas involving variables and functions again when we discuss the algebra and calculus features of the Calculator.

Undo and Redo

If you make a mistake, you can usually correct it by pressing shift-U, the "undo" command. First, clear the stack (M-0 DEL) and exit and restart Calc (M-# M-# M-# M-#) to make sure things start off with a clean slate. Now:

```
1:  2          2:  2          1:  8          2:  2          1:  6
    .          1:  3              .          1:  3              .
                   .                             .

   2 RET           3              ^              U              *
```

You can undo any number of times. Calc keeps a complete record of all you have done since you last opened the Calc window. After the above example, you could type:

```
1:  6          2:  2          1:  2              .              .
    .          1:  3              .
                   .
                                                        (error)
                   U              U              U              U
```

You can also type D to "redo" a command that you have undone mistakenly.

```
    .          1:  2          2:  2          1:  6          1:  6
                   .          1:  3              .              .
                                  .
                                                        (error)
                   D              D              D              D
```

It was not possible to redo past the 6, since that was placed there by something other than an undo command.

You can think of undo and redo as a sort of "time machine." Press U to go backward in time, D to go forward. If you go backward and do something (like *) then, as any science fiction reader knows, you have changed your future and you cannot go forward again. Thus, the inability to redo past the 6 even though there was an earlier undo command.

You can always recall an earlier result using the Trail. We've ignored the trail so far, but it has been faithfully recording everything we did since we loaded the Calculator. If the Trail is not displayed, press t d now to turn it on.

Let's try grabbing an earlier result. The 8 we computed was undone by a U command, and was lost even to Redo when we pressed *, but it's still there in the trail. There should be a little `>' arrow (the trail pointer) resting on the last trail entry. If there isn't, press t] to reset the trail pointer. Now, press t p to move the arrow onto the line containing 8, and press t y to "yank" that number back onto the stack.

If you press t] again, you will see that even our Yank command went into the trail.

Let's go further back in time. Earlier in the tutorial we computed a huge integer using the formula `2^3^4'. We don't remember what it was, but the first digits were "241". Press t r (which stands for trail-search-reverse), then type 241. The trail cursor will jump back to the next previous occurrence of the string "241" in the trail. This is just a regular Emacs incremental search; you can now press C-s or C-r to continue the search forwards or backwards as you like.

To finish the search, press RET. This halts the incremental search and leaves the trail pointer at the thing we found. Now we can type t y to yank that number onto the stack. If we hadn't remembered the "241", we could simply have searched for 2^3^4, then pressed RET t n to halt and then move to the next item.

You may have noticed that all the trail-related commands begin with the letter t. (The store-and-recall commands, on the other hand, all began with s.) Calc has so many commands that there aren't enough keys for all of them, so various commands are grouped into two-letter sequences where the first letter is called the prefix key. If you type a prefix key by accident, you can press C-g to cancel it. (In fact, you can press C-g to cancel almost anything in Emacs.) To get help on a prefix key, press that key followed by ?. Some prefixes have several lines of help, so you need to press ? repeatedly to see them all.

Try pressing t ? now. You will see a line of the form,

```
trail/time: Display; Fwd, Back; Next, Prev, Here, [, ]; Yank:  [MORE]  t-
```

The word "trail" indicates that the t prefix key contains trail-related commands. Each entry on the line shows one command, with a single capital letter showing which letter you press to get that command. We have used t n, t p, t], and t y so far. The `[MORE]' means you can press ? again to see more t-prefix comands. Notice that the commands are roughly divided (by semicolons) into related groups.

When you are in the help display for a prefix key, the prefix is still active. If you press another key, like y for example, it will be interpreted as a t y command. If all you wanted was to look at the help messages, press C-g afterwards to cancel the prefix.

One more way to correct an error is by editing the stack entries. The actual Stack buffer is marked read-only and must not be edited directly, but you can press ` (the backquote or accent grave) to edit a stack entry.

Try entering `3.141439' now. If this is supposed to represent pi, it's got several errors. Press ` to edit this number. Now use the normal Emacs cursor motion and editing keys to change the second 4 to a 5, and to transpose the 3 and the 9. When you press RET, the number on the stack will be replaced by your new number. This works for formulas, vectors, and all other types of values you can put on the stack. The ` key also works during entry of a number or algebraic formula.

Mode-Setting Commands

Calc has many types of modes that affect the way it interprets your commands or the way it displays data. We have already seen one mode, namely algebraic mode. There are many others, too; we'll try some of the most common ones here.

Perhaps the most fundamental mode in Calc is the current precision. Notice the `12' on the Calc window's mode line:


```
--%%-Calc: 12 Deg       (Calculator)----All------
```

Most of the symbols there are Emacs things you don't need to worry about, but the `12' and the `Deg' are mode indicators. The `12' means that calculations should always be carried to 12 significant figures. That is why, when we type 1 RET 7 /, we get 0.142857142857 with exactly 12 digits, not counting leading and trailing zeros.

You can set the precision to anything you like by pressing p, then entering a suitable number. Try pressing p 30 RET, then doing 1 RET 7 / again:

```
1:  0.142857142857
2:  0.142857142857142857142857142857
    .
```

Although the precision can be set arbitrarily high, Calc always has to have *some* value for the current precision. After all, the true value 1/7 is an infinitely repeating decimal; Calc has to stop somewhere.

Of course, calculations are slower the more digits you request. Press p 12 now to set the precision back down to the default.

Calculations always use the current precision. For example, even though we have a 30-digit value for 1/7 on the stack, if we use it in a calculation in 12-digit mode it will be rounded down to 12 digits before it is used. Try it; press RET to duplicate the number, then 1 +. Notice that the RET key didn't round the number, because it doesn't do any calculation. But the instant we pressed +, the number was rounded down.

```
1:  0.142857142857
2:  0.142857142857142857142857142857
3:  1.14285714286
    .
```

In fact, since we added a digit on the left, we had to lose one digit on the right from even the 12-digit value of 1/7.

How did we get more than 12 digits when we computed `2^3^4'? The answer is that Calc makes a distinction between integers and floating-point numbers, or floats. An integer is a number that does not contain a decimal point. There is no such thing as an "infinitely repeating fraction integer," so Calc doesn't have to limit itself. If you asked for `2^10000' (don't try this!), you would have to wait a long time but you would eventually get an exact answer. If you ask for `2.^10000', you will quickly get an answer which is correct only to 12 places. The decimal point tells Calc that it should use floating-point arithmetic to get the answer, not exact integer arithmetic.

You can use the F (`calc-floor`) command to convert a floating-point value to an integer, and c f (`calc-float`) to convert an integer to floating-point form.

Let's try entering that last calculation:

```
1:  2.          2:  2.          1:  1.99506311689e3010
    .           1:  10000           .
                    .
```


```
  2.0 RET          10000 RET      ^
```

Notice the letter `e' in there. It represents "times ten to the power of," and is used by Calc automatically whenever writing the number out fully would introduce more extra zeros than you probably want to see. You can enter numbers in this notation, too.

```
1:  2.          2:  2.          1:  1.99506311678e3010
    .           1:  10000.          .
                    .

  2.0 RET          1e4 RET        ^
```

Hey, the answer is different! Look closely at the middle columns of the two examples. In the first, the stack contained the exact integer 10000, but in the second it contained a floating-point value with a decimal point. When you raise a number to an integer power, Calc uses repeated squaring and multiplication to get the answer. When you use a floating-point power, Calc uses logarithms and exponentials. As you can see, a slight error crept in during one of these methods. Which one should we trust? Let's raise the precision a bit and find out:

```
    .           1:  2.          2:  2.          1:  1.9950631168808 28e3010
                    .           1:  10000.          .
                                    .

 p 16 RET         2. RET          1e4            ^    p 12 RET
```

Presumably, it doesn't matter whether we do this higher-precision calculation using an integer or floating-point power, since we have added enough "guard digits" to trust the first 12 digits no matter what. And the verdict is... Integer powers were more accurate; in fact, the result was only off by one unit in the last place.

Calc does many of its internal calculations to a slightly higher precision, but it doesn't always bump the precision up enough. In each case, Calc added about two digits of precision during its calculation and then rounded back down to 12 digits afterward. In one case, it was enough; in the the other, it wasn't. If you really need x digits of precision, it never hurts to do the calculation with a few extra guard digits.

What if we want guard digits but don't want to look at them? We can set the float format. Calc supports four major formats for floating-point numbers, called normal, fixed-point, scientific notation, and engineering notation. You get them by pressing d n, d f, d s, and d e, respectively. In each case, you can supply a numeric prefix argument which says how many digits should be displayed. As an example, let's put a few numbers onto the stack and try some different display modes. First, use M-0 DEL to clear the stack, then enter the four numbers shown here:

```
4:  12345       4:  12345       4:  12345       4:  12345       4:  12345
3:  12345.      3:  12300.      3:  1.2345e4    3:  1.23e4      3:  12345.000
2:  123.45      2:  123.        2:  1.2345e2    2:  1.23e2      2:  123.450
1:  12.345      1:  12.3        1:  1.2345e1    1:  1.23e1      1:  12.345
    .               .               .               .               .

   d n           M-3 d n         d s           M-3 d s         M-3 d f
```

Notice that when we typed M-3 d n, the numbers were rounded down to three significant digits, but then w we typed d s all five significant figures reappeared. The float format does not affect how numbers are stor only affects how they are displayed. Only the current precision governs the actual rounding of numbers in the Calculator's memory.

Engineering notation, not shown here, is like scientific notation except the exponent (the power-of-ten part) is always adjusted to be a multiple of three (as in "kilo," "micro," etc.). As a result there will be one, two, or three digits before the decimal point.

Whenever you change a display-related mode, Calc redraws everything in the stack. This may be slow if there are many things on the stack, so Calc allows you to type shift-H before any mode command to prevent it from updating the stack. Anything Calc displays after the mode-changing command will appear in the new format.

```
4:  12345      4:  12345      4:  12345      4:  12345      4:  12345
3:  12345.000  3:  12345.000  3:  12345.000  3:  1.2345e4   3:  12345.
2:  123.450    2:  123.450    2:  1.2345e1   2:  1.2345e1   2:  123.45
1:  12.345     1:  1.2345e1   1:  1.2345e2   1:  1.2345e2   1:  12.345
    .              .              .              .              .

    H d s          DEL U          TAB            d SPC          d n
```

Here the H d s command changes to scientific notation but without updating the screen. Deleting the top stack entry and undoing it back causes it to show up in the new format; swapping the top two stack entries reformats both entries. The d SPC command refreshes the whole stack. The d n command changes back to the normal float format; since it doesn't have an H prefix, it also updates all the stack entries to be in d n format.

Notice that the integer 12345 was not affected by any of the float formats. Integers are integers, and are always displayed exactly.

Large integers have their own problems. Let's look back at the result of 2^3^4.

```
241785163922925834941235 2
```

Quick--how many digits does this have? Try typing d g:

```
2,417,851,639,229,258,349,412,352
```

Now how many digits does this have? It's much easier to tell! We can actually group digits into clumps of any size. Some people prefer M-5 d g:

```
24178,51639,22925,83494,12352
```

Let's see what happens to floating-point numbers when they are grouped. First, type p 25 RET to make sure we have enough precision to get ourselves into trouble. Now, type 1e13 /:

```
24,17851,63922.9258349412352
```

The integer part is grouped but the fractional part isn't. Now try M-- M-5 d g (that's meta-minus-sign, meta-five):


```
24,17851,63922.92583,49412,352
```

If you find it hard to tell the decimal point from the commas, try changing the grouping character to a space with d , SPC:

```
24 17851 63922.92583 49412 352
```

Type d , , to restore the normal grouping character, then d g again to turn grouping off. Also, press p 12 to restore the default precision.

Press U enough times to get the original big integer back. (Notice that U does not undo each mode-setting command; if you want to undo a mode-setting command, you have to do it yourself.) Now, type d r 16 RET:

```
16#2000000000000000000
```

The number is now displayed in hexadecimal, or "base-16" form. Suddenly it looks pretty simple; this should be no surprise, since we got this number by computing a power of two, and 16 is a power of 2. In fact, we can use d r 2 RET to see it in actual binary form:

```
2#1000000000000000000000000000000000000000000000000000000000000000 ...
```

We don't have enough space here to show all the zeros! They won't fit on a typical screen, either, so you will have to use horizontal scrolling to see them all. Press < and > to scroll the stack window left and right by half its width. Another way to view something large is to press ` (back-quote) to edit the top of stack in a separate window. (Press M-# M-# when you are done.)

You can enter non-decimal numbers using the # symbol, too. Let's see what the hexadecimal number `5FE' looks like in binary. Type 16#5FE (the letters can be typed in upper or lower case; they will always appear in upper case). It will also help to turn grouping on with d g:

```
2#101,1111,1110
```

Notice that d g groups by fours by default if the display radix is binary or hexadecimal, but by threes if it is decimal, octal, or any other radix.

Now let's see that number in decimal; type d r 10:

```
1,534
```

Numbers are not *stored* with any particular radix attached. They're just numbers; they can be entered in any radix, and are always displayed in whatever radix you've chosen with d r. The current radix applies to integers, fractions, and floats.

(*) **Exercise 1.** Your friend Joe tried to enter one-third as `3#0.1' in d r 3 mode with a precision of 12. He got `3#0.0222222...' (with 25 2's) in the display. When he multiplied that by three, he got `3#0.222222...' instead of the expected `3#1'. Next, Joe entered `3#0.2' and, to his great relief, saw `3#0.2' on the screen. But when he typed 2 /, he got `3#0.10000001' (some zeros omitted). What's going on here? See section Modes Tutorial Exercise 1. (*)

(*) **Exercise 2.** Scientific notation works in non-decimal modes in the natural way (the exponent is a power of

the radix instead of a power of ten, although the exponent itself is always written in decimal). Thus `8#1.2 = 8#1230.0'. Suppose we have the hexadecimal number `f.e8f' times 16 to the 15th power: We write `16#f.e8fe15'. What is wrong with this picture? What could we write instead that would work better? See section Modes Tutorial Exercise 2. (*)

The m prefix key has another set of modes, relating to the way Calc interprets your inputs and does computations. Whereas d-prefix modes generally affect the way things look, m-prefix modes affect the way they are actually computed.

The most popular m-prefix mode is the angular mode. Notice the `Deg' indicator in the mode line. This means that if you use a command that interprets a number as an angle, it will assume the angle is measured in degrees. For example,

```
1:  45          1:  0.707106781187   1:  0.500000000001    1:  0.5
    .               .                    .                     .

    45              S                    2 ^                   c 1
```

The shift-S command computes the sine of an angle. The sine of 45 degrees is @c{$\sqrt{2}/2$} sqrt(2)/2; squaring this yields 2/4 = 0.5. However, there has been a slight roundoff error because the representation of @c{$\sqrt{2}/2$} sqrt(2)/2 wasn't exact. The c 1 command is a handy way to clean up numbers in this case; it temporarily reduces the precision by one digit while it re-rounds the number on the top of the stack.

(*) **Exercise 3.** Your friend Joe computed the sine of 45 degrees as shown above, then, hoping to avoid an inexact result, he increased the precision to 16 digits before squaring. What happened? See section Modes Tutorial Exercise 3. (*)

To do this calculation in radians, we would type m r first. (The indicator changes to `Rad'.) 45 degrees corresponds to pi/4 radians. To get @c{π} pi, press the P key. (Once again, this is a shifted capital P. Remember, unshifted p sets the precision.)

```
1:  3.14159265359   1:  0.785398163398   1:  0.707106781187
    .                   .                   .

    P                   4 /       m r       S
```

Likewise, inverse trigonometric functions generate results in either radians or degrees, depending on the current angular mode.

```
1:  0.707106781187   1:  0.785398163398   1:  45.
    .                    .                   .

    .5 Q         m r     I S         m d     U I S
```

Here we compute the Inverse Sine of @c{$\sqrt{0.5}$} sqrt(0.5), first in radians, then in degrees.

Use c d and c r to convert a number from radians to degrees and vice-versa.

```
1:  45          1:  0.785398163397     1:  45.
```


```
    .               .                   .

    45              c r                 c d
```

Another interesting mode is fraction mode. Normally, dividing two integers produces a floating-point result if the quotient can't be expressed as an exact integer. Fraction mode causes integer division to produce a fraction, i.e., a rational number, instead.

```
2:  12          1:  1.33333333333    1:  4:3
1:  9               .                    .
    .

 12 RET 9           /           m f      U /      m f
```

In the first case, we get an approximate floating-point result. In the second case, we get an exact fractional result (four-thirds).

You can enter a fraction at any time using : notation. (Calc uses : instead of / as the fraction separator because / is already used to divide the top two stack elements.) Calculations involving fractions will always produce exact fractional results; fraction mode only says what to do when dividing two integers.

(*) **Exercise 4.** If fractional arithmetic is exact, why would you ever use floating-point numbers instead? See section Modes Tutorial Exercise 4. (*)

Typing m f doesn't change any existing values in the stack. In the above example, we had to Undo the division and do it over again when we changed to fraction mode. But if you use the evaluates-to operator you can get commands like m f to recompute for you.

```
1:  12 / 9 => 1.33333333333    1:  12 / 9 => 1.333    1:  12 / 9 => 4:3
    .                              .                      .

   ' 12/9 => RET                   p 4 RET                m f
```

In this example, the righthand side of the `=>' operator on the stack is recomputed when we change the precision, then again when we change to fraction mode. All `=>' expressions on the stack are recomputed every time you change any mode that might affect their values.

Arithmetic Tutorial

In this section, we explore the arithmetic and scientific functions available in the Calculator.

The standard arithmetic commands are +, -, *, /, and ^. Each normally takes two numbers from the top of the stack and pushes back a result. The n and & keys perform change-sign and reciprocal operations, respectively.

```
1:  5           1:  0.2         1:  5.          1:  -5.         1:  5.
    .               .               .               .               .

    5               &               &               n               n
```

You can apply a "binary operator" like + across any number of stack entries by giving it a numeric prefix. can also apply it pairwise to several stack elements along with the top one if you use a negative prefix.

```
3:  2          1:  9          3:  2          4:  2          3:  12
2:  3              .          2:  3          3:  3          2:  13
1:  4                         1:  4          2:  4          1:  14
    .                             .          1:  10             .
                                                 .

2 RET 3 RET 4     M-3 +           U              10          M-- M-3 +
```

You can apply a "unary operator" like & to the top n stack entries with a numeric prefix, too.

```
3:  2          3:  0.5                3:  0.5
2:  3          2:  0.333333333333     2:  3.
1:  4          1:  0.25               1:  4.
    .              .                      .

2 RET 3 RET 4      M-3 &                  M-2 &
```

Notice that the results here are left in floating-point form. We can convert them back to integers by pressing F, the "floor" function. This function rounds down to the next lower integer. There is also R, which rounds to the nearest integer.

```
7:  2.         7:  2          7:  2
6:  2.4        6:  2          6:  2
5:  2.5        5:  2          5:  3
4:  2.6        4:  2          4:  3
3:  -2.        3:  -2         3:  -2
2:  -2.4       2:  -3         2:  -2
1:  -2.6       1:  -3         1:  -3
    .              .              .

                  M-7 F          U M-7 R
```

Since dividing-and-flooring (i.e., "integer quotient") is such a common operation, Calc provides a special command for that purpose, the backslash \. Another common arithmetic operator is %, which computes the remainder that would arise from a \ operation, i.e., the "modulo" of two numbers. For example,

```
2:  1234       1:  12         2:  1234       1:  34
1:  100            .          1:  100            .
    .                             .

1234 RET 100       \              U              %
```

These commands actually work for any real numbers, not just integers.

```
2:  3.1415     1:  3          2:  3.1415     1:  0.1415
```


```
1:  1              .          1:  1              .
    .                             .

3.1415 RET 1       \              U              %
```

(*) **Exercise 1.** The \ command would appear to be a frill, since you could always do the same thing with / F. Think of a situation where this is not true---/ F would be inadequate. Now think of a way you could get around the problem if Calc didn't provide a \ command. See section Arithmetic Tutorial Exercise 1. (*)

We've already seen the Q (square root) and S (sine) commands. Other commands along those lines are C (cosine), T (tangent), E (e^x) and L (natural logarithm). These can be modified by the I (inverse) and H (hyperbolic) prefix keys.

Let's compute the sine and cosine of an angle, and verify the identity @c{$\sin^2x + \cos^2x = 1$} sin(x)^2 + cos(x)^2 = 1. We'll arbitrarily pick -*64* degrees as a good value for x. With the angular mode set to degrees (type m d), do:

```
2:  -64        2:  -64        2:  -0.89879   2:  -0.89879   1:  1.
1:  -64        1:  -0.89879   1:  -64        1:  0.43837        .
    .              .              .              .

 64 n RET RET      S              TAB            C              f h
```

(For brevity, we're showing only five digits of the results here. You can of course do these calculations to any precision you like.)

Remember, f h is the `calc-hypot`, or square-root of sum of squares, command.

Another identity is @c{$\displaystyle\tan x = {\sin x \over \cos x}$} tan(x) = sin(x) / cos(x).

```
2:  -0.89879   1:  -2.0503    1:  -64.
1:  0.43837        .              .
    .

    U              /              I T
```

A physical interpretation of this calculation is that if you move 0.89879 units downward and 0.43837 units to the right, your direction of motion is -*64* degrees from horizontal. Suppose we move in the opposite direction, up and to the left:

```
2:  -0.89879   2:  0.89879    1:  -2.0503    1:  -64.
1:  0.43837    1:  -0.43837       .              .
    .              .

    U U            M-2 n          /              I T
```

How can the angle be the same? The answer is that the / operation loses information about the signs of its inputs. Because the quotient is negative, we know exactly one of the inputs was negative, but we can't tell which one. There is an f T [`arctan2`] function which computes the inverse tangent of the quotient of a pair

of numbers. Since you feed it the two original numbers, it has enough information to give you a full 360-degree answer.

```
2:  0.89879    1:  116.       3:  116.       2:  116.       1:  180.
1:  -0.43837       .          2:  -0.89879   1:  -64.           .
    .                         1:  0.43837        .
                                  .

    U U            f T          M-RET M-2 n       f T            -
```

The resulting angles differ by 180 degrees; in other words, they point in opposite directions, just as we would expect.

The META-RET we used in the third step is the "last-arguments" command. It is sort of like Undo, except that it restores the arguments of the last command to the stack without removing the command's result. It is useful in situations like this one, where we need to do several operations on the same inputs. We could have accomplished the same thing by using M-2 RET to duplicate the top two stack elements right after the U U, then a pair of M-TAB commands to cycle the 116 up around the duplicates.

A similar identity is supposed to hold for hyperbolic sines and cosines, except that it is the *difference* cosh(x)^2 - sinh(x)^2 that always equals one. Let's try to verify this identity.

```
2:  -64        2:  -64        2:  -64        2:  9.7192e54  2:  9.7192e54
1:  -64        1:  -3.1175e27 1:  9.7192e54  1:  -64        1:  9.7192e54
    .              .              .              .              .

 64 n RET RET      H C            2 ^            TAB            H S 2 ^
```

Something's obviously wrong, because when we subtract these numbers the answer will clearly be zero! But if you think about it, if these numbers *did* differ by one, it would be in the 55th decimal place. The difference we seek has been lost entirely to roundoff error.

We could verify this hypothesis by doing the actual calculation with, say, 60 decimal places of precision. This will be slow, but not enormously so. Try it if you wish; sure enough, the answer is 0.99999, reasonably close to 1.

Of course, a more reasonable way to verify the identity is to use a more reasonable value for x!

Some Calculator commands use the Hyperbolic prefix for other purposes. The logarithm and exponential functions, for example, work to the base e normally but use base-10 instead if you use the Hyperbolic prefix.

```
1:  1000       1:  6.9077     1:  1000       1:  3
    .              .              .              .

    1000           L              U              H L
```

First, we mistakenly compute a natural logarithm. Then we undo and compute a common logarithm instead.

The B key computes a general base-b logarithm for any value of b.


```
2:  1000        1:  3           1:  1000.       2:  1000.       1:  6.9077
1:  10              .               .           1:  2.71828         .
    .                                               .

 1000 RET 10        B               H E             H P             B
```

Here we first use B to compute the base-10 logarithm, then use the "hyperbolic" exponential as a cheap hack to recover the number 1000, then use B again to compute the natural logarithm. Note that P with the hyperbolic prefix pushes the constant e onto the stack.

You may have noticed that both times we took the base-10 logarithm of 1000, we got an exact integer result. Calc always tries to give an exact rational result for calculations involving rational numbers where possible. But when we used H E, the result was a floating-point number for no apparent reason. In fact, if we had computed 10 RET 3 ^ we *would* have gotten an exact integer 1000. But the H E command is rigged to generate a floating-point result all of the time so that 1000 H E will not waste time computing a thousand-digit integer when all you probably wanted was `1e1000'.

(*) **Exercise 2.** Find a pair of integer inputs to the B command for which Calc could find an exact rational result but doesn't. See section Arithmetic Tutorial Exercise 2. (*)

The Calculator also has a set of functions relating to combinatorics and statistics. You may be familiar with the factorial function, which computes the product of all the integers up to a given number.

```
1:  100         1:  93326215443...      1:  100.        1:  9.3326e157
    .               .                       .               .

    100             !                       U c f           !
```

Recall, the c f command converts the integer or fraction at the top of the stack to floating-point format. If you take the factorial of a floating-point number, you get a floating-point result accurate to the current precision. But if you give ! an exact integer, you get an exact integer result (158 digits long in this case).

If you take the factorial of a non-integer, Calc uses a generalized factorial function defined in terms of Euler's Gamma function gamma(n) (which is itself available as the f g command).

```
3:  4.          3:  24.                 1:  5.5         1:  52.342777847
2:  4.5         2:  52.3427777847           .               .
1:  5.          1:  120.
    .               .

                    M-3 !                M-0 DEL 5.5        f g
```

Here we verify the identity @c{$n! = \Gamma(n+1)$} n! = gamma(n+1).

The binomial coefficient n-choose-m@c{ or $\displaystyle {n \choose m}$} is defined by n! / m! (n-m)! for all reals n and m. The intermediate results in this formula can become quite large even if the final result is small; the k c command computes a binomial coefficient in a way that avoids large intermediate values.

The k prefix key defines several common functions out of combinatorics and number theory. Here we compute the binomial coefficient 30-choose-20, then determine its prime factorization.


```
2:  30          1:  30045015    1:  [3, 3, 5, 7, 11, 13, 23, 29]
1:  20              .               .
    .

 30 RET 20          k c             k f
```

You can verify these prime factors by using v u to "unpack" this vector into 8 separate stack entries, then M-8 * to multiply them back together. The result is the original number, 30045015.

Suppose a program you are writing needs a hash table with at least 10000 entries. It's best to use a prime number as the actual size of a hash table. Calc can compute the next prime number after 10000:

```
1:  10000       1:  10007       1:  9973
    .               .               .

    10000           k n             I k n
```

Just for kicks we've also computed the next prime *less* than 10000.

See section Financial Functions, for a description of the Calculator commands that deal with business and financial calculations (functions like `pv`, `rate`, and `sln`).

See section Binary Number Functions, to read about the commands for operating on binary numbers (like `and`, `xor`, and `lsh`).

Vector/Matrix Tutorial

A vector is a list of numbers or other Calc data objects. Calc provides a large set of commands that operate on vectors. Some are familiar operations from vector analysis. Others simply treat a vector as a list of objects.

Vector Analysis

If you add two vectors, the result is a vector of the sums of the elements, taken pairwise.

```
1:  [1, 2, 3]       2:  [1, 2, 3]       1:  [8, 8, 3]
    .               1:  [7, 6, 0]           .
                        .

    [1,2,3]  s 1        [7 6 0]  s 2        +
```

Note that we can separate the vector elements with either commas or spaces. This is true whether we are using incomplete vectors or algebraic entry. The s 1 and s 2 commands save these vectors so we can easily reuse them later.

If you multiply two vectors, the result is the sum of the products of the elements taken pairwise. This is called the dot product of the vectors.

```
2:  [1, 2, 3]       1:  19
```


```
1:  [7, 6, 0]          .
    .

    r 1 r 2            *
```

The dot product of two vectors is equal to the product of their lengths times the cosine of the angle between them. (Here the vector is interpreted as a line from the origin (0,0,0) to the specified point in three-dimensional space.) The A (absolute value) command can be used to compute the length of a vector.

```
3:  19             3:  19          1:  0.550782    1:  56.579
2:  [1, 2, 3]      2:  3.741657        .               .
1:  [7, 6, 0]      1:  9.219544
    .                  .

    M-RET              M-2 A           * /             I C
```

First we recall the arguments to the dot product command, then we compute the absolute values of the top two stack entries to obtain the lengths of the vectors, then we divide the dot product by the product of the lengths to get the cosine of the angle. The inverse cosine finds that the angle between the vectors is about 56 degrees.

The cross product of two vectors is a vector whose length is the product of the lengths of the inputs times the sine of the angle between them, and whose direction is perpendicular to both input vectors. Unlike the dot product, the cross product is defined only for three-dimensional vectors. Let's double-check our computation of the angle using the cross product.

```
2:  [1, 2, 3]  3:  [-18, 21, -8]  1:  [-0.52, 0.61, -0.23]  1:  56.579
1:  [7, 6, 0]  2:  [1, 2, 3]          .                         .
    .          1:  [7, 6, 0]
               .

    r 1 r 2        V C  s 3  M-RET    M-2 A * /                 A I S
```

First we recall the original vectors and compute their cross product, which we also store for later reference. Now we divide the vector by the product of the lengths of the original vectors. The length of this vector should be the sine of the angle; sure enough, it is!

Vector-related commands generally begin with the v prefix key. Some are uppercase letters and some are lowercase. To make it easier to type these commands, the shift-V prefix key acts the same as the v key. (See section General Mode Commands, for a way to make all prefix keys have this property.)

If we take the dot product of two perpendicular vectors we expect to get zero, since the cosine of 90 degrees is zero. Let's check that the cross product is indeed perpendicular to both inputs:

```
2:  [1, 2, 3]      1:  0          2:  [7, 6, 0]      1:  0
1:  [-18, 21, -8]      .          1:  [-18, 21, -8]      .
    .                             .

    r 1 r 3            *          DEL r 2 r 3            *
```

(*) **Exercise 1.** Given a vector on the top of the stack, what keystrokes would you use to normalize the ve
i.e., to reduce its length to one without changing its direction? See section Vector Tutorial Exercise 1. (*)

(*) **Exercise 2.** Suppose a certain particle can be at any of several positions along a ruler. You have a list of those positions in the form of a vector, and another list of the probabilities for the particle to be at the corresponding positions. Find the average position of the particle. See section Vector Tutorial Exercise 2. (*)

Matrices

A matrix is just a vector of vectors, all the same length. This means you can enter a matrix using nested brackets. You can also use the semicolon character to enter a matrix. We'll show both methods here:

```
1:  [ [ 1, 2, 3 ]                1:  [ [ 1, 2, 3 ]
      [ 4, 5, 6 ] ]                     [ 4, 5, 6 ] ]
    .                                 .

  [[1 2 3] [4 5 6]]                 ' [1 2 3; 4 5 6] RET
```

We'll be using this matrix again, so type s 4 to save it now.

Note that semicolons work with incomplete vectors, but they work better in algebraic entry. That's why we use the apostrophe in the second example.

When two matrices are multiplied, the lefthand matrix must have the same number of columns as the righthand matrix has rows. Row i, column j of the result is effectively the dot product of row i of the left matrix by column j of the right matrix.

If we try to duplicate this matrix and multiply it by itself, the dimensions are wrong and the multiplication cannot take place:

```
1:  [ [ 1, 2, 3 ]   * [ [ 1, 2, 3 ]
      [ 4, 5, 6 ] ]     [ 4, 5, 6 ] ]
    .

    RET *
```

Though rather hard to read, this is a formula which shows the product of two matrices. The `*' function, having invalid arguments, has been left in symbolic form.

We can multiply the matrices if we transpose one of them first.

```
2:  [ [ 1, 2, 3 ]       1:  [ [ 14, 32 ]      1:  [ [ 17, 22, 27 ]
      [ 4, 5, 6 ] ]           [ 32, 77 ] ]          [ 22, 29, 36 ]
1:  [ [ 1, 4 ]            .                       [ 27, 36, 45 ] ]
      [ 2, 5 ]                                  .
      [ 3, 6 ] ]
    .

    U v t                 *                   U TAB *
```

Matrix multiplication is not commutative; indeed, switching the order of the operands can even change th dimensions of the result matrix, as happened here!

If you multiply a plain vector by a matrix, it is treated as a single row or column depending on which side of the matrix it is on. The result is a plain vector which should also be interpreted as a row or column as appropriate.

```
2:  [ [ 1, 2, 3 ]      1:  [14, 32]
      [ 4, 5, 6 ] ]        .
1:  [1, 2, 3]
    .

    r 4 r 1              *
```

Multiplying in the other order wouldn't work because the number of rows in the matrix is different from the number of elements in the vector.

(*) **Exercise 1.** Use `*' to sum along the rows of the above @c{$2\times3$} 2x3 matrix to get [6, 15]. Now use `*' to sum along the columns to get [5, 7, 9]. See section Matrix Tutorial Exercise 1. (*)

An identity matrix is a square matrix with ones along the diagonal and zeros elsewhere. It has the property that multiplication by an identity matrix, on the left or on the right, always produces the original matrix.

```
1:  [ [ 1, 2, 3 ]      2:  [ [ 1, 2, 3 ]      1:  [ [ 1, 2, 3 ]
      [ 4, 5, 6 ] ]          [ 4, 5, 6 ] ]          [ 4, 5, 6 ] ]
    .                  1:  [ [ 1, 0, 0 ]        .
                           [ 0, 1, 0 ]
                           [ 0, 0, 1 ] ]
                       .

    r 4                v i 3 RET              *
```

If a matrix is square, it is often possible to find its inverse, that is, a matrix which, when multiplied by the original matrix, yields an identity matrix. The & (reciprocal) key also computes the inverse of a matrix.

```
1:  [ [ 1, 2, 3 ]      1:  [ [   -2.4,     1.2,   -0.2 ]
      [ 4, 5, 6 ]            [    2.8,    -1.4,    0.4 ]
      [ 7, 6, 0 ] ]          [ -0.73333, 0.53333, -0.2 ] ]
    .                    .

    r 4 r 2 |  s 5       &
```

The vertical bar | concatenates numbers, vectors, and matrices together. Here we have used it to add a new row onto our matrix to make it square.

We can multiply these two matrices in either order to get an identity.

```
1:  [ [ 1., 0., 0. ]      1:  [ [ 1., 0., 0. ]
      [ 0., 1., 0. ]            [ 0., 1., 0. ]
```


```
      [ 0., 0., 1. ] ]          [ 0., 0., 1. ] ]
    .                         .

    M-RET  *                  U TAB *
```

Matrix inverses are related to systems of linear equations in algebra. Suppose we had the following set of equations:

This can be cast into the matrix equation,

We can solve this system of equations by multiplying both sides by the inverse of the matrix. Calc can do this all in one step:

```
2:  [6, 2, 3]          1:  [-12.6, 15.2, -3.93333]
1:  [ [ 1, 2, 3 ]          .
      [ 4, 5, 6 ]
      [ 7, 6, 0 ] ]
    .

    [6,2,3] r 5            /
```

The result is the [a, b, c] vector that solves the equations. (Dividing by a square matrix is equivalent to multiplying by its inverse.)

Let's verify this solution:

```
2:  [ [ 1, 2, 3 ]              1:  [6., 2., 3.]
      [ 4, 5, 6 ]                  .
      [ 7, 6, 0 ] ]
1:  [-12.6, 15.2, -3.93333]
    .

    r 5  TAB                       *
```

Note that we had to be careful about the order in which we multiplied the matrix and vector. If we multiplied in the other order, Calc would assume the vector was a row vector in order to make the dimensions come out right, and the answer would be incorrect. If you don't feel safe letting Calc take either interpretation of your vectors, use explicit @c{$N\times1$} Nx1 or @c{$1\times N$} 1xN matrices instead. In this case, you would enter the original column vector as `[[6], [2], [3]]' or `[6; 2; 3]'.

(*) **Exercise 2.** Algebraic entry allows you to make vectors and matrices that include variables. Solve the following system of equations to get expressions for x and y in terms of a and b.

See section Matrix Tutorial Exercise 2. (*)

(*) **Exercise 3.** A system of equations is "over-determined" if it has more equations than variables. It is often the case that there are no values for the variables that will satisfy all the equations at once, but it is still useful to find a set of values which "nearly" satisfy all the equations. In terms of matrix equations, you can't solve A X = B directly because the matrix A is not square for an over-determined system. Matrix inversion works only for square matrices. One common trick is to multiply both sides on the left by the transpose of A: Now @c{$A^T A$} trn(A)*A is a square matrix so a solution is possible. It turns out that the X vector you

compute in this way will be a "least-squares" solution, which can be regarded as the "closest" solution to t set of equations. Use Calc to solve the following over-determined system:

See section Matrix Tutorial Exercise 3. (*)

Vectors as Lists

Although Calc has a number of features for manipulating vectors and matrices as mathematical objects, you can also treat vectors as simple lists of values. For example, we saw that the k f command returns a vector which is a list of the prime factors of a number.

You can pack and unpack stack entries into vectors:

```
3:  10          1:  [10, 20, 30]      3:  10
2:  20              .                 2:  20
1:  30                                1:  30
    .                                     .

                  M-3 v p                 v u
```

You can also build vectors out of consecutive integers, or out of many copies of a given value:

```
1:  [1, 2, 3, 4]    2:  [1, 2, 3, 4]    2:  [1, 2, 3, 4]
    .               1:  17              1:  [17, 17, 17, 17]
                        .                   .

    v x 4 RET           17                  v b 4 RET
```

You can apply an operator to every element of a vector using the map command.

```
1:  [17, 34, 51, 68]   1:  [289, 1156, 2601, 4624]  1:  [17, 34, 51, 68]
    .                      .                            .

    V M *                  2 V M ^                      V M Q
```

In the first step, we multiply the vector of integers by the vector of 17's elementwise. In the second step, we raise each element to the power two. (The general rule is that both operands must be vectors of the same length, or else one must be a vector and the other a plain number.) In the final step, we take the square root of each element.

(*) **Exercise 1.** Compute a vector of powers of two from @c{2^{-4}} 2^-4 to 2^4. See section List Tutorial Exercise 1. (*)

You can also reduce a binary operator across a vector. For example, reducing `*' computes the product of all the elements in the vector:

```
1:  123123      1:  [3, 7, 11, 13, 41]      1:  123123
    .               .                           .
```


```
    123123          k f                             V R *
```

In this example, we decompose 123123 into its prime factors, then multiply those factors together again to yield the original number.

We could compute a dot product "by hand" using mapping and reduction:

```
2:  [1, 2, 3]     1:  [7, 12, 0]     1:  19
1:  [7, 6, 0]         .                  .
    .

    r 1 r 2           V M *              V R +
```

Recalling two vectors from the previous section, we compute the sum of pairwise products of the elements to get the same answer for the dot product as before.

A slight variant of vector reduction is the accumulate operation, V U. This produces a vector of the intermediate results from a corresponding reduction. Here we compute a table of factorials:

```
1:  [1, 2, 3, 4, 5, 6]    1:  [1, 2, 6, 24, 120, 720]
    .                         .

    v x 6 RET                 V U *
```

Calc allows vectors to grow as large as you like, although it gets rather slow if vectors have more than about a hundred elements. Actually, most of the time is spent formatting these large vectors for display, not calculating on them. Try the following experiment (if your computer is very fast you may need to substitute a larger vector size).

```
1:  [1, 2, 3, 4, ...      1:  [2, 3, 4, 5, ...
    .                         .

    v x 500 RET               1 V M +
```

Now press v . (the letter v, then a period) and try the experiment again. In v . mode, long vectors are displayed "abbreviated" like this:

```
1:  [1, 2, 3, ..., 500]   1:  [2, 3, 4, ..., 501]
    .                         .

    v x 500 RET               1 V M +
```

(where now the `...' is actually part of the Calc display). You will find both operations are now much faster. But notice that even in v . mode, the full vectors are still shown in the Trail. Type t . to cause the trail to abbreviate as well, and try the experiment one more time. Operations on long vectors are now quite fast! (But of course if you use t . you will lose the ability to get old vectors back using the t y command.)

An easy way to view a full vector when v . mode is active is to press ` (back-quote) to edit the vector; editing always works with the full, unabbreviated value.

As a larger example, let's try to fit a straight line to some data, using the method of least squares. (Calc has built-in command for least-squares curve fitting, but we'll do it by hand here just to practice working with vectors.) Suppose we have the following list of values in a file we have loaded into Emacs:

```
  x         y
 --       ---
 1.34    0.234
 1.41    0.298
 1.49    0.402
 1.56    0.412
 1.64    0.466
 1.73    0.473
 1.82    0.601
 1.91    0.519
 2.01    0.603
 2.11    0.637
 2.22    0.645
 2.33    0.705
 2.45    0.917
 2.58    1.009
 2.71    0.971
 2.85    1.062
 3.00    1.148
 3.15    1.157
 3.32    1.354
```

If you are reading this tutorial in printed form, you will find it easiest to press M-# i to enter the on-line Info version of the manual and find this table there. (Press g, then type List Tutorial, to jump straight to this section.)

Position the cursor at the upper-left corner of this table, just to the left of the 1.34. Press C-@ to set the mark. (On your system this may be C-2, C-SPC, or NUL.) Now position the cursor to the lower-right, just after the 1.354. You have now defined this region as an Emacs "rectangle." Still in the Info buffer, type M-# r. This command (`calc-grab-rectangle`) will pop you back into the Calculator, with the contents of the rectangle you specified in the form of a matrix.

```
1:  [ [ 1.34, 0.234 ]
      [ 1.41, 0.298 ]
      ...
```

(You may wish to use v . mode to abbreviate the display of this large matrix.)

We want to treat this as a pair of lists. The first step is to transpose this matrix into a pair of rows. Remember, a matrix is just a vector of vectors. So we can unpack the matrix into a pair of row vectors on the stack.

```
1:  [ [ 1.34,  1.41,  1.49,  ... ]     2:  [1.34, 1.41, 1.49, ... ]
      [ 0.234, 0.298, 0.402, ... ] ]   1:  [0.234, 0.298, 0.402, ... ]
    .                                      .
```


```
    v t                                         v u
```

Let's store these in quick variables 1 and 2, respectively.

```
1:  [1.34, 1.41, 1.49, ... ]          .
    .

    t 2                                 t 1
```

(Recall that t 2 is a variant of s 2 that removes the stored value from the stack.)

In a least squares fit, the slope m is given by the formula

where @c{$\sum x$} sum(x) represents the sum of all the values of x. While there is an actual `sum` function in Calc, it's easier to sum a vector using a simple reduction. First, let's compute the four different sums that this formula uses.

```
1:  41.63                   1:  98.0003
    .                           .

 r 1 V R +   t 3             r 1 2 V M ^ V R +   t 4


1:  13.613                  1:  33.36554
    .                           .

 r 2 V R +   t 5             r 1 r 2 V M * V R +   t 6
```

Finally, we also need N, the number of data points. This is just the length of either of our lists.

```
1:  19
    .

 r 1 v l   t 7
```

(That's v followed by a lower-case l.)

Now we grind through the formula:

```
1:  633.94526  2:  633.94526  1:  67.23607
    .          1:  566.70919      .
                   .

 r 7 r 6 *      r 3 r 5 *          -


2:  67.23607   3:  67.23607   2:  67.23607   1:  0.52141679
1:  1862.0057  2:  1862.0057  1:  128.9488       .
    .          1:  1733.0569      .
```


```
                    .

 r 7 r 4 *      r 3 2 ^           -              /   t 8
```

That gives us the slope m. The y-intercept b can now be found with the simple formula,

```
1:  13.613     2:  13.613     1:  -8.09358   1:  -0.425978
    .          1:  21.70658       .              .
                   .

   r 5            r 8 r 3 *       -              r 7 /   t 9
```

Let's "plot" this straight line approximation, @c{$y \approx m x + b$} m x + b, and compare it with the original data.

```
1:  [0.699, 0.735, ... ]    1:  [0.273, 0.309, ... ]
    .                           .

    r 1 r 8 *                   r 9 +    s 0
```

Notice that multiplying a vector by a constant, and adding a constant to a vector, can be done without mapping commands since these are common operations from vector algebra. As far as Calc is concerned, we've just been doing geometry in 19-dimensional space!

We can subtract this vector from our original y vector to get a feel for the error of our fit. Let's find the maximum error:

```
1:  [0.0387, 0.0112, ... ]   1:  [0.0387, 0.0112, ... ]   1:  0.0897
    .                            .                           .

    r 2 -                        V M A                       V R X
```

First we compute a vector of differences, then we take the absolute values of these differences, then we reduce the `max` function across the vector. (The `max` function is on the two-key sequence f x; because it is so common to use `max` in a vector operation, the letters X and N are also accepted for `max` and `min` in this context. In general, you answer the V M or V R prompt with the actual key sequence that invokes the function you want. You could have typed V R f x or even V R x max RET if you had preferred.)

If your system has the GNUPLOT program, you can see graphs of your data and your straight line to see how well they match. (If you have GNUPLOT 3.0, the following instructions will work regardless of the kind of display you have. Some GNUPLOT 2.0, non-X-windows systems may require additional steps to view the graphs.)

Let's start by plotting the original data. Recall the "*x*" and "*y*" vectors onto the stack and press g f. This "fast" graphing command does everything you need to do for simple, straightforward plotting of data.

```
2:  [1.34, 1.41, 1.49, ... ]
1:  [0.234, 0.298, 0.402, ... ]
    .
```


```
    r 1 r 2    g f
```

If all goes well, you will shortly get a new window containing a graph of the data. (If not, contact your GNUPLOT or Calc installer to find out what went wrong.) In the X window system, this will be a separate graphics window. For other kinds of displays, the default is to display the graph in Emacs itself using rough character graphics. Press q when you are done viewing the character graphics.

Next, let's add the line we got from our least-squares fit:

```
2:  [1.34, 1.41, 1.49, ... ]
1:  [0.273, 0.309, 0.351, ... ]
    .

    DEL r 0    g a  g p
```

It's not very useful to get symbols to mark the data points on this second curve; you can type g S g p to remove them. Type g q when you are done to remove the X graphics window and terminate GNUPLOT.

(*) **Exercise 2.** An earlier exercise showed how to do least squares fitting to a general system of equations. Our 19 data points are really 19 equations of the form y_i = m x_i + b for different pairs of (x_i,y_i). Use the matrix-transpose method to solve for m and b, duplicating the above result. See section List Tutorial Exercise 2. (*)

(*) **Exercise 3.** If the input data do not form a rectangle, you can use M-# g (`calc-grab-region`) to grab the data the way Emacs normally works with regions--it reads left-to-right, top-to-bottom, treating line breaks the same as spaces. Use this command to find the geometric mean of the following numbers. (The geometric mean is the nth root of the product of n numbers.)

```
2.3  6  22  15.1  7
  15  14  7.5
  2.5
```

The M-# g command accepts numbers separated by spaces or commas, with or without surrounding vector brackets. See section List Tutorial Exercise 3. (*)

```
1:  [1, 2, 3, 4, 5, 6, 7]      1:  [0, 1, 2, 3, 4, 5, 6]
    .                              .

    v x 7 RET                      1 -


1:  [1, -6, 15, -20, 15, -6, 1]          1:  0
    .                                        .

    V M ' (-1)^$ choose(6,$) RET             V R +
```

The V M ' command prompts you to enter any algebraic expression to define the function to map over the vector. The symbol `$' inside this expression represents the argument to the function. The Calculator applies

this formula to each element of the vector, substituting each element's value for the `$' sign(s) in turn.

To define a two-argument function, use `$$' for the first argument and `$' for the second: V M ' $$-$ RET is equivalent to V M -. This is analogous to regular algebraic entry, where `$$' would refer to the next-to-top stack entry and `$' would refer to the top stack entry, and ' $$-$ RET would act exactly like -.

Notice that the V M ' command has recorded two things in the trail: The result, as usual, and also a funny-looking thing marked `oper' that represents the operator function you typed in. The function is enclosed in `< >' brackets, and the argument is denoted by a `#' sign. If there were several arguments, they would be shown as `#1', `#2', and so on. (For example, V M ' $$-$ will put the function `<#1 - #2>' on the trail.) This object is a "nameless function"; you can use nameless `< >' notation to answer the V M ' prompt if you like. Nameless function notation has the interesting, occasionally useful property that a nameless function is not actually evaluated until it is used. For example, V M ' $+random(2.0) evaluates `random(2.0)' once and adds that random number to all elements of the vector, but V M ' <#+random(2.0)> evaluates the `random(2.0)' separately for each vector element.

Another group of operators that are often useful with V M are the relational operators: a =, for example, compares two numbers and gives the result 1 if they are equal, or 0 if not. Similarly, a < checks for one number being less than another.

Other useful vector operations include v v, to reverse a vector end-for-end; V S, to sort the elements of a vector into increasing order; and v r and v c, to extract one row or column of a matrix, or (in both cases) to extract one element of a plain vector. With a negative argument, v r and v c instead delete one row, column, or vector element.

(*) **Exercise 4.** The kth divisor function is the sum of the kth powers of all the divisors of an integer n. Figure out a method for computing the divisor function for reasonably small values of n. As a test, the 0th and 1st divisor functions of 30 are 8 and 72, respectively. See section List Tutorial Exercise 4. (*)

(*) **Exercise 5.** The k f command produces a list of prime factors for a number. Sometimes it is important to know that a number is square-free, i.e., that no prime occurs more than once in its list of prime factors. Find a sequence of keystrokes to tell if a number is square-free; your method should leave 1 on the stack if it is, or 0 if it isn't. See section List Tutorial Exercise 5. (*)

(*) **Exercise 6.** Build a list of lists that looks like the following diagram. (You may wish to use the v / command to enable multi-line display of vectors.)

```
1:  [ [1],
      [1, 2],
      [1, 2, 3],
      [1, 2, 3, 4],
      [1, 2, 3, 4, 5],
      [1, 2, 3, 4, 5, 6] ]
```

See section List Tutorial Exercise 6. (*)

(*) **Exercise 7.** Build the following list of lists.

```
1:  [ [0],
      [1, 2],
```


```
[3, 4, 5],
[6, 7, 8, 9],
[10, 11, 12, 13, 14],
[15, 16, 17, 18, 19, 20] ]
```

See section List Tutorial Exercise 7. (*)

(*) **Exercise 8.** Compute a list of values of Bessel's J1 function `besJ(1,x)' for x from 0 to 5 in steps of 0.25. Find the value of x (from among the above set of values) for which `besJ(1,x)' is a maximum. Use an "automatic" method, i.e., just reading along the list by hand to find the largest value is not allowed! (There is an a X command which does this kind of thing automatically; see section Numerical Solutions.) See section List Tutorial Exercise 8. (*)

(*) **Exercise 9.** You are given an integer in the range 0 <= N < 10^m for m=12 (i.e., an integer of less than twelve digits). Convert this integer into a vector of m digits, each in the range from 0 to 9. In vector-of-digits notation, add one to this integer to produce a vector of m+1 digits (since there could be a carry out of the most significant digit). Convert this vector back into a regular integer. A good integer to try is 25129925999. See section List Tutorial Exercise 9. (*)

(*) **Exercise 10.** Your friend Joe tried to use V R a = to test if all numbers in a list were equal. What happened? How would you do this test? See section List Tutorial Exercise 10. (*)

(*) **Exercise 11.** The area of a circle of radius one is @c{π} pi. The area of the @c{2×2} 2x2 square that encloses that circle is 4. So if we throw *N* darts at random points in the square, about @c{$\pi/4$} pi/4 of them will land inside the circle. This gives us an entertaining way to estimate the value of @c{π} pi. The k r command picks a random number between zero and the value on the stack. We could get a random floating-point number between *-1* and 1 by typing 2.0 k r 1 -. Build a vector of 100 random (x,y) points in this square, then use vector mapping and reduction to count how many points lie inside the unit circle. Hint: Use the v b command. See section List Tutorial Exercise 11. (*)

(*) **Exercise 12.** The matchstick problem provides another way to calculate @c{π} pi. Say you have an infinite field of vertical lines with a spacing of one inch. Toss a one-inch matchstick onto the field. The probability that the matchstick will land crossing a line turns out to be @c{$2/\pi$} 2/pi. Toss 100 matchsticks to estimate pi. (If you want still more fun, the probability that the GCD (k g) of two large integers is one turns out to be @c{$6/\pi^2$} 6/pi^2. That provides yet another way to estimate @c{π} pi.) See section List Tutorial Exercise 12. (*)

(*) **Exercise 13.** An algebraic entry of a string in double-quote marks, `'"hello"', creates a vector of the numerical (ASCII) codes of the characters (here, [104, 101, 108, 108, 111]). Sometimes it is convenient to compute a hash code of a string, which is just an integer that represents the value of that string. Two equal strings have the same hash code; two different strings probably have different hash codes. (For example, Calc has over 400 function names, but Emacs can quickly find the definition for any given name because it has sorted the functions into "buckets" by their hash codes. Sometimes a few names will hash into the same bucket, but it is easier to search among a few names than among all the names.) One popular hash function is computed as follows: First set h = 0. Then, for each character from the string in turn, set h = 3h + c_i where c_i is the character's ASCII code. If we have 511 buckets, we then take the hash code modulo 511 to get the bucket number. Develop a simple command or commands for converting string vectors into hash codes. The hash code for `'"Testing, 1, 2, 3"' is 1960915098, which modulo 511 is 121. See section List Tutorial Exercise 13. (*)

(*) **Exercise 14.** The H V R and H V U commands do nested function evaluations. H V U takes a starting value and a number of steps n from the stack; it then applies the function you give to the starting value 0, up to n times and returns a vector of the results. Use this command to create a "random walk" of 50 steps. Start with the two-dimensional point (0,0); then take one step a random distance between *-1* and 1 in both x and y; then take another step, and so on. Use the g f command to display this random walk. Now modify your random walk to walk a unit distance, but in a random direction, at each step. (Hint: The `sincos` function returns a vector of the cosine and sine of an angle.) See section List Tutorial Exercise 14. (*)

Types Tutorial

Calc understands a variety of data types as well as simple numbers. In this section, we'll experiment with each of these types in turn.

The numbers we've been using so far have mainly been either integers or floats. We saw that floats are usually a good approximation to the mathematical concept of real numbers, but they are only approximations and are susceptible to roundoff error. Calc also supports fractions, which can exactly represent any rational number.

```
1:  3628800     2:  3628800     1:  518400:7    1:  518414:7    1:  7:518414
    .           1:  49              .               .               .
                    .

    10 !            49 RET          :               2 +             &
```

The : command divides two integers to get a fraction; / would normally divide integers to get a floating-point result. Notice we had to type RET between the 49 and the : since the : would otherwise be interpreted as part of a fraction beginning with 49.

You can convert between floating-point and fractional format using c f and c F:

```
1:  1.35027217629e-5    1:  7:518414
    .                       .

    c f                     c F
```

The c F command replaces a floating-point number with the "simplest" fraction whose floating-point representation is the same, to within the current precision.

```
1:  3.14159265359   1:  1146408:364913   1:  3.1416   1:  355:113
    .                   .                   .           .

    P                   c F      DEL        p 5 RET P       c F
```

(*) **Exercise 1.** A calculation has produced the result 1.26508260337. You suspect it is the square root of the product of @c{π} pi and some rational number. Is it? (Be sure to allow for roundoff error!) See section Types Tutorial Exercise 1. (*)

Complex numbers can be stored in both rectangular and polar form.


```
1:  -9      1:  (0, 3)     1:  (3; 90.)    1:  (6; 90.)    1:  (2.4495; 45.
    .           .              .               .               .

    9 n         Q              c p             2 *             Q
```

The square root of *-9* is by default rendered in rectangular form (0 + 3i), but we can convert it to polar form (3 with a phase angle of 90 degrees). All the usual arithmetic and scientific operations are defined on both types of complex numbers.

Another generalized kind of number is infinity. Infinity isn't really a number, but it can sometimes be treated like one. Calc uses the symbol `inf` to represent positive infinity, i.e., a value greater than any real number. Naturally, you can also write `-inf' for minus infinity, a value less than any real number. The word `inf` can only be input using algebraic entry.

```
2:  inf         2:  -inf        2:  -inf        2:  -inf        1:  nan
1:  -17         1:  -inf        1:  -inf        1:  inf             .
    .               .               .               .

' inf RET 17 n      *  RET          72 +            A               +
```

Since infinity is infinitely large, multiplying it by any finite number (like *-17*) has no effect, except that since *-17* is negative, it changes a plus infinity to a minus infinity. ("A huge positive number, multiplied by *-17*, yields a huge negative number.") Adding any finite number to infinity also leaves it unchanged. Taking an absolute value gives us plus infinity again. Finally, we add this plus infinity to the minus infinity we had earlier. If you work it out, you might expect the answer to be *-72* for this. But the 72 has been completely lost next to the infinities; by the time we compute `inf - inf' the finite difference between them, if any, is indetectable. So we say the result is indeterminate, which Calc writes with the symbol `nan` (for Not A Number).

Dividing by zero is normally treated as an error, but you can get Calc to write an answer in terms of infinity by pressing m i to turn on "infinite mode."

```
3:  nan         2:  nan         2:  nan         2:  nan         1:  nan
2:  1           1:  1 / 0       1:  uinf        1:  uinf            .
1:  0               .               .               .
    .

  1 RET 0           /         m i     U /             17 n *          +
```

Dividing by zero normally is left unevaluated, but after m i it instead gives an infinite result. The answer is actually `uinf`, "undirected infinity." If you look at a graph of 1 / x around x = 0, you'll see that it goes toward plus infinity as you approach zero from above, but toward minus infinity as you approach from below. Since we said only 1 / 0, Calc knows that the answer is infinite but not in which direction. That's what `uinf` means. Notice that multiplying `uinf` by a negative number still leaves plain `uinf`; there's no point in saying `-uinf' because the sign of `uinf` is unknown anyway. Finally, we add `uinf` to our `nan`, yielding `nan` again. It's easy to see that, because `nan` means "totally unknown" while `uinf` means "unknown sign but known to be infinite," the more mysterious `nan` wins out when it is combined with `uinf`, or, for that matter, with anything else.

(*) **Exercise 2.** Predict what Calc will answer for each of these formulas: `inf / inf', `exp(inf)', `exp(-inf)', `sqrt(-inf)', `sqrt(uinf)', `abs(uinf)', `ln(0)'. See section Types Tutorial Exercise 2. (*)

(*) **Exercise 3.** We saw that `inf - inf = nan', which stands for an unknown value. Can `nan' stand for a complex number? Can it stand for infinity? See section Types Tutorial Exercise 3. (*)

HMS forms represent a value in terms of hours, minutes, and seconds.

```
1:  2@ 30' 0"      1:  3@ 30' 0"      2:  3@ 30' 0"      1:  2.
    .                  .              1:  1@ 45' 0."         .
                                          .

  2@ 30' RET           1 +                RET 2 /            /
```

HMS forms can also be used to hold angles in degrees, minutes, and seconds.

```
1:  0.5         1:  26.56505    1:  26@ 33' 54.18"    1:  0.44721
    .               .               .                     .

    0.5             I T             c h                   S
```

First we convert the inverse tangent of 0.5 to degrees-minutes-seconds form, then we take the sine of that angle. Note that the trigonometric functions will accept HMS forms directly as input.

(*) **Exercise 4.** The Beatles' *Abbey Road* is 47 minutes and 26 seconds long, and contains 17 songs. What is the average length of a song on *Abbey Road*? If the Extended Disco Version of *Abbey Road* added 20 seconds to the length of each song, how long would the album be? See section Types Tutorial Exercise 4. (*)

A date form represents a date, or a date and time. Dates must be entered using algebraic entry. Date forms are surrounded by `< >' symbols; most standard formats for dates are recognized.

```
2:  <Sun Jan 13, 1991>                    1:  2.25
1:  <6:00pm Thu Jan 10, 1991>                 .
    .

' <13 Jan 1991>, <1/10/91, 6pm> RET           -
```

In this example, we enter two dates, then subtract to find the number of days between them. It is also possible to add an HMS form or a number (of days) to a date form to get another date form.

```
1:  <4:45:59pm Mon Jan 14, 1991>      1:  <2:50:59am Thu Jan 17, 1991>
    .                                     .

    t N                                   2 + 10@ 5' +
```

The t N ("now") command pushes the current date and time on the stack; then we add two days, ten hours and five minutes to the date and time. Other date-and-time related commands include t J, which does Julian day conversions, t W, which finds the beginning of the week in which a date form lies, and t I, which increments a date by one or several months. See section Date Arithmetic, for more.

(*) **Exercise 5.** How many days until the next Friday the 13th? See section Types Tutorial Exercise 5. (*)

(*) **Exercise 6.** How many leap years will there be between now and the year 10001 A.D.? See section Types Tutorial Exercise 6. (*)

An error form represents a mean value with an attached standard deviation, or error estimate. Suppose our measurements indicate that a certain telephone pole is about 30 meters away, with an estimated error of 1 meter, and 8 meters tall, with an estimated error of 0.2 meters. What is the slope of a line from here to the top of the pole, and what is the equivalent angle in degrees?

```
1:  8 +/- 0.2    2:  8 +/- 0.2   1:  0.266 +/- 0.011   1:  14.93 +/- 0.594
    .            1:  30 +/- 1        .                      .
                     .

    8 p .2 RET       30 p 1          /                      I T
```

This means that the angle is about 15 degrees, and, assuming our original error estimates were valid standard deviations, there is about a 60% chance that the result is correct within 0.59 degrees.

(*) **Exercise 7.** The volume of a torus (a donut shape) is 2 pi^2 R r^2 where R is the radius of the circle that defines the center of the tube and r is the radius of the tube itself. Suppose R is 20 cm and r is 4 cm, each known to within 5 percent. What is the volume and the relative uncertainty of the volume? See section Types Tutorial Exercise 7. (*)

An interval form represents a range of values. While an error form is best for making statistical estimates, intervals give you exact bounds on an answer. Suppose we additionally know that our telephone pole is definitely between 28 and 31 meters away, and that it is between 7.7 and 8.1 meters tall.

```
1:  [7.7 .. 8.1]  2:  [7.7 .. 8.1]  1:  [0.24 .. 0.28]  1:  [13.9 .. 16.1]
    .             1:  [28 .. 31]        .                     .
                      .

  [ 7.7 .. 8.1 ]    [ 28 .. 31 ]        /                     I T
```

If our bounds were correct, then the angle to the top of the pole is sure to lie in the range shown.

The square brackets around these intervals indicate that the endpoints themselves are allowable values. In other words, the distance to the telephone pole is between 28 and 31, *inclusive*. You can also make an interval that is exclusive of its endpoints by writing parentheses instead of square brackets. You can even make an interval which is inclusive ("closed") on one end and exclusive ("open") on the other.

```
1:  [1 .. 10)    1:  (0.1 .. 1]   2:  (0.1 .. 1]   1:  (0.2 .. 3)
    .                .            1:  [2 .. 3)          .
                                      .

  [ 1 .. 10 )        &              [ 2 .. 3 )          *
```

The Calculator automatically keeps track of which end values should be open and which should be closed. You can also make infinite or semi-infinite intervals by using `-inf' or `inf' for one or both endpoints.

(*) **Exercise 8.** What answer would you expect from `1 / (0 .. 10)'? What about `1 / (-10 .. 0)'? What about `1 / [0 .. 10]' (where the interval actually includes zero)? What about `1 / (-10 .. 10)'? See section Types Tutorial Exercise 8. (*)

(*) **Exercise 9.** Two easy ways of squaring a number are RET * and 2 ^. Normally these produce the same answer. Would you expect this still to hold true for interval forms? If not, which of these will result in a larger interval? See section Types Tutorial Exercise 9. (*)

A modulo form is used for performing arithmetic modulo *M*. For example, arithmetic involving time is generally done modulo 12 or 24 hours.

```
1:  17 mod 24     1:  3 mod 24      1:  21 mod 24     1:  9 mod 24
    .                 .                 .                 .

    17 M 24 RET       10 +              n                 5 /
```

In this last step, Calc has found a new number which, when multiplied by 5 modulo 24, produces the original number, 21. If *M* is prime it is always possible to find such a number. For non-prime *M* like 24, it is only sometimes possible.

```
1:  10 mod 24     1:  16 mod 24     1:  1000000...    1:  16
    .                 .                 .                 .

    10 M 24 RET       100 ^             10 RET 100 ^      24 %
```

These two calculations get the same answer, but the first one is much more efficient because it avoids the huge intermediate value that arises in the second one.

(*) **Exercise 10.** A theorem of Pierre de Fermat says that @c{\w{$x^{n-1} \bmod n = 1$}} x^(n-1) mod n = 1 if n is a prime number and x is an integer less than n. If n is *not* a prime number, this will *not* be true for most values of x. Thus we can test informally if a number is prime by trying this formula for several values of x. Use this test to tell whether the following numbers are prime: 811749613, 15485863. See section Types Tutorial Exercise 10. (*)

It is possible to use HMS forms as parts of error forms, intervals, modulo forms, or as the phase part of a polar complex number. For example, the `calc-time` command pushes the current time of day on the stack as an HMS/modulo form.

```
1:  17@ 34' 45" mod 24@ 0' 0"     1:  6@ 22' 15" mod 24@ 0' 0"
    .                                 .

    x time RET                        n
```

This calculation tells me it is six hours and 22 minutes until midnight.

(*) **Exercise 11.** A rule of thumb is that one year is about @c{$\pi \times 10^7$} pi * 10^7 seconds. What time will it be that many seconds from right now? See section Types Tutorial Exercise 11. (*)

(*) **Exercise 12.** You are preparing to order packaging for the CD release of the Extended Disco Version of *Abbey Road*. You are told that the songs will actually be anywhere from 20 to 60 seconds longer than the

originals. One CD can hold about 75 minutes of music. Should you order single or double packages? See section Types Tutorial Exercise 12. (*)

Another kind of data the Calculator can manipulate is numbers with units. This isn't strictly a new data type; it's simply an application of algebraic expressions, where we use variables with suggestive names like `cm' and `in' to represent units like centimeters and inches.

```
1:  2 in        1:  5.08 cm      1:  0.027778 fath   1:  0.0508 m
    .               .                .                   .

    ' 2in RET       u c cm RET       u c fath RET        u b
```

We enter the quantity "2 inches" (actually an algebraic expression which means two times the variable `in'), then we convert it first to centimeters, then to fathoms, then finally to "base" units, which in this case means meters.

```
1:  9 acre     1:  3 sqrt(acre)   1:  190.84 m   1:  190.84 m + 30 cm
    .              .                  .              .

 ' 9 acre RET      Q                  u s            ' $+30 cm RET


1:  191.14 m     1:  36536.3046 m^2    1:  365363046 cm^2
    .                .                     .

    u s              2 ^                   u c cgs
```

Since units expressions are really just formulas, taking the square root of `acre' is undefined. After all, `acre' might be an algebraic variable that you will someday assign a value. We use the "units-simplify" command to simplify the expression with variables being interpreted as unit names.

In the final step, we have converted not to a particular unit, but to a units system. The "cgs" system uses centimeters instead of meters as its standard unit of length.

There is a wide variety of units defined in the Calculator.

```
1:  55 mph     1:  88.5139 kph   1:  88.5139 km / hr   1:  8.201407e-8 c
    .              .                 .                    .

 ' 55 mph RET      u c kph RET       u c km/hr RET        u c c RET
```

We express a speed first in miles per hour, then in kilometers per hour, then again using a slightly more explicit notation, then finally in terms of fractions of the speed of light.

Temperature conversions are a bit more tricky. There are two ways to interpret "20 degrees Fahrenheit"---it could mean an actual temperature, or it could mean a change in temperature. For normal units there is no difference, but temperature units have an offset as well as a scale factor and so there must be two explicit commands for them.


```
1:  20 degF       1:  11.1111 degC     1:  -20:3 degC    1:  -6.666 deg
    .                 .                    .                 .

  ' 20 degF RET       u c degC RET         U u t degC RET    c f
```

First we convert a change of 20 degrees Fahrenheit into an equivalent change in degrees Celsius (or Centigrade). Then, we convert the absolute temperature 20 degrees Fahrenheit into Celsius. Since this comes out as an exact fraction, we then convert to floating-point for easier comparison with the other result.

For simple unit conversions, you can put a plain number on the stack. Then u c and u t will prompt for both old and new units. When you use this method, you're responsible for remembering which numbers are in which units:

```
1:  55         1:  88.5139              1:  8.201407e-8
    .              .                        .

    55             u c mph RET kph RET      u c km/hr RET c RET
```

To see a complete list of built-in units, type u v. Press M-# c again to re-enter the Calculator when you're done looking at the units table.

(*) **Exercise 13.** How many seconds are there really in a year? See section Types Tutorial Exercise 13. (*)

(*) **Exercise 14.** Supercomputer designs are limited by the speed of light (and of electricity, which is nearly as fast). Suppose a computer has a 4.1 ns (nanosecond) clock cycle, and its cabinet is one meter across. Is speed of light going to be a significant factor in its design? See section Types Tutorial Exercise 14. (*)

(*) **Exercise 15.** Sam the Slug normally travels about five yards in an hour. He has obtained a supply of Power Pills; each Power Pill he eats doubles his speed. How many Power Pills can he swallow and still travel legally on most US highways? See section Types Tutorial Exercise 15. (*)

Algebra and Calculus Tutorial

This section shows how to use Calc's algebra facilities to solve equations, do simple calculus problems, and manipulate algebraic formulas.

Basic Algebra

If you enter a formula in algebraic mode that refers to variables, the formula itself is pushed onto the stack. You can manipulate formulas as regular data objects.

```
1:  2 x^2 - 6       1:  6 - 2 x^2       1:  (6 - 2 x^2) (3 x^2 + y)
    .                   .                   .

    ' 2x^2-6 RET        n                   ' 3x^2+y RET *
```

(*) **Exercise 1.** Do ' x RET Q 2 ^ and ' x RET 2 ^ Q both wind up with the same result (`x')? Why or why not? See section Algebra Tutorial Exercise 1. (*)

There are also commands for doing common algebraic operations on formulas. Continuing with the formu from the last example,

```
1:  18 x^2 + 6 y - 6 x^4 - 2 x^2 y    1:  (18 - 2 y) x^2 - 6 x^4 + 6 y
    .                                     .

    a x                                   a c x RET
```

First we "expand" using the distributive law, then we "collect" terms involving like powers of x.

Let's find the value of this expression when x is 2 and y is one-half.

```
1:  17 x^2 - 6 x^4 + 3      1:  -25
    .                           .

    1:2 s l y RET               2 s l x RET
```

The s l command means "let"; it takes a number from the top of the stack and temporarily assigns it as the value of the variable you specify. It then evaluates (as if by the = key) the next expression on the stack. After this command, the variable goes back to its original value, if any.

(An earlier exercise in this tutorial involved storing a value in the variable x; if this value is still there, you will have to unstore it with s u x RET before the above example will work properly.)

Let's find the maximum value of our original expression when y is one-half and x ranges over all possible values. We can do this by taking the derivative with respect to x and examining values of x for which the derivative is zero. If the second derivative of the function at that value of x is negative, the function has a local maximum there.

```
1:  17 x^2 - 6 x^4 + 3      1:  34 x - 24 x^3
    .                           .

    U DEL  s 1                  a d x RET   s 2
```

Well, the derivative is clearly zero when x is zero. To find the other root(s), let's divide through by x and then solve:

```
1:  (34 x - 24 x^3) / x    1:  34 x / x - 24 x^3 / x    1:  34 - 24 x^2
    .                          .                            .

    ' x RET /                  a x                          a s


1:  34 - 24 x^2 = 0        1:  x = 1.19023
    .                          .

    0 a =  s 3                 a S x RET
```

Notice the use of a s to "simplify" the formula. When the default algebraic simplifications don't do enough,

you can use a s to tell Calc to spend more time on the job.

Now we compute the second derivative and plug in our values of x:

```
1:  1.19023          2:  1.19023          2:  1.19023
    .                1:  34 x - 24 x^3    1:  34 - 72 x^2
                         .                    .

    a .                  r 2                  a d x RET s 4
```

(The a . command extracts just the righthand side of an equation. Another method would have been to use v u to unpack the equation `x = 1.19' to `x' and `1.19', then use M-- M-2 DEL to delete the `x'.)

```
2:  34 - 72 x^2   1:  -68.          2:  34 - 72 x^2     1:  34
1:  1.19023           .             1:  0                   .
    .                                   .

    TAB                 s l x RET       U DEL 0             s l x RET
```

The first of these second derivatives is negative, so we know the function has a maximum value at x = 1.19023. (The function also has a local *minimum* at x = 0.)

When we solved for x, we got only one value even though 34 - 24 x^2 = 0 is a quadratic equation that ought to have two solutions. The reason is that a S normally returns a single "principal" solution. If it needs to come up with an arbitrary sign (as occurs in the quadratic formula) it picks +. If it needs an arbitrary integer, it picks zero. We can get a full solution by pressing H (the Hyperbolic flag) before a S.

```
1:  34 - 24 x^2 = 0    1:  x = 1.19023 s1      1:  x = -1.19023
    .                      .                       .

    r 3                    H a S x RET  s 5         1 n  s l s1 RET
```

Calc has invented the variable `s1' to represent an unknown sign; it is supposed to be either *+1* or *-1*. Here we have used the "let" command to evaluate the expression when the sign is negative. If we plugged this into our second derivative we would get the same, negative, answer, so x = -1.19023 is also a maximum.

To find the actual maximum value, we must plug our two values of x into the original formula.

```
2:  17 x^2 - 6 x^4 + 3    1:  24.08333 s1^2 - 12.04166 s1^4 + 3
1:  x = 1.19023 s1            .
    .

    r 1 r 5                   s l RET
```

(Here we see another way to use s l; if its input is an equation with a variable on the lefthand side, then s l treats the equation like an assignment to that variable if you don't give a variable name.)

It's clear that this will have the same value for either sign of `s1`, but let's work it out anyway, just for the exercise:


```
2:  [-1, 1]              1:  [15.04166, 15.04166]
1:  24.08333 s1^2 ...        .
    .

  [ 1 n , 1 ] TAB            V M $ RET
```

Here we have used a vector mapping operation to evaluate the function at several values of `s1' at once. V M $ is like V M ' except that it takes the formula from the top of the stack. The formula is interpreted as a function to apply across the vector at the next-to-top stack level. Since a formula on the stack can't contain `$' signs, Calc assumes the variables in the formula stand for different arguments. It prompts you for an argument list, giving the list of all variables in the formula in alphabetical order as the default list. In this case the default is `(s1)', which is just what we want so we simply press RET at the prompt.

If there had been several different values, we could have used V R X to find the global maximum.

Calc has a built-in a P command that solves an equation using H a S and returns a vector of all the solutions. It simply automates the job we just did by hand. Applied to our original cubic polynomial, it would produce the vector of solutions [1.19023, -1.19023, 0]. (There is also an a X command which finds a local maximum of a function. It uses a numerical search method rather than examining the derivatives, and thus requires you to provide some kind of initial guess to show it where to look.)

(*) **Exercise 2.** Given a vector of the roots of a polynomial (such as the output of an a P command), what sequence of commands would you use to reconstruct the original polynomial? (The answer will be unique to within a constant multiple; choose the solution where the leading coefficient is one.) See section Algebra Tutorial Exercise 2. (*)

The m s command enables "symbolic mode," in which formulas like `sqrt(5)' that can't be evaluated exactly are left in symbolic form rather than giving a floating-point approximate answer. Fraction mode (m f) is also useful when doing algebra.

```
2:  34 x - 24 x^3        2:  34 x - 24 x^3
1:  34 x - 24 x^3        1:  [sqrt(51) / 6, sqrt(51) / -6, 0]
    .                        .

    r 2  RET     m s  m f    a P x RET
```

One more mode that makes reading formulas easier is "Big mode."

```
                3
2:  34 x - 24 x


     ____   ____
    V 51   V 51
1:  [-----, -----, 0]
       6      -6

    .

    d B
```

Here things like powers, square roots, and quotients and fractions are displayed in a two-dimensional pict form. Calc has other language modes as well, such as C mode, FORTRAN mode, and TeX mode.

```
2:  34*x - 24*pow(x, 3)                2:  34*x - 24*x**3
1:  {sqrt(51) / 6, sqrt(51) / -6, 0}   1:  /sqrt(51) / 6, sqrt(51) / -6, 0/
    .                                      .

    d C                                    d F


3:  34 x - 24 x^3
2:  [{\sqrt{51} \over 6}, {\sqrt{51} \over -6}, 0]
1:  {2 \over 3} \sqrt{5}
    .

    d T   ' 2 \sqrt{5} \over 3 RET
```

As you can see, language modes affect both entry and display of formulas. They affect such things as the names used for built-in functions, the set of arithmetic operators and their precedences, and notations for vectors and matrices.

Notice that `sqrt(51)' may cause problems with older implementations of C and FORTRAN, which would require something more like `sqrt(51.0)'. It is always wise to check over the formulas produced by the various language modes to make sure they are fully correct.

Type m s, m f, and d N to reset these modes. (You may prefer to remain in Big mode, but all the examples in the tutorial are shown in normal mode.)

What is the area under the portion of this curve from x = 1 to 2? This is simply the integral of the function:

```
1:  17 x^2 - 6 x^4 + 3      1:  5.6666 x^3 - 1.2 x^5 + 3 x
    .                           .

    r 1                         a i x
```

We want to evaluate this at our two values for x and subtract. One way to do it is again with vector mapping and reduction:

```
2:  [2, 1]             1:  [12.93333, 7.46666]    1:  5.46666
1:  5.6666 x^3 ...         .                          .

   [ 2 , 1 ] TAB           V M $ RET                  V R -
```

(*) **Exercise 3.** Find the integral from 1 to y of @c{$x \sin \pi x$} x sin(pi x) (where the sine is calculated in radians). Find the values of the integral for integers y from 1 to 5. See section Algebra Tutorial Exercise 3. (*)

Calc's integrator can do many simple integrals symbolically, but many others are beyond its capabilities. Suppose we wish to find the area under the curve @c{$\sin x \ln x$} sin(x) ln(x) over the same range of x. If you entered this formula and typed a i x RET (don't bother to try this), Calc would work for a long time but

would be unable to find a solution. In fact, there is no closed-form solution to this integral. Now what do you do?

One approach would be to do the integral numerically. It is not hard to do this by hand using vector mapping and reduction. It is rather slow, though, since the sine and logarithm functions take a long time. We can save some time by reducing the working precision.

```
3:  10                  1:  [1, 1.1, 1.2,  ...  , 1.8, 1.9]
2:  1                       .
1:  0.1
    .

 10 RET 1 RET .1 RET        C-u v x
```

(Note that we have used the extended version of v x; we could also have used plain v x as follows: v x 10 RET 9 + .1 *.)

```
2:  [1, 1.1, ... ]              1:  [0., 0.084941, 0.16993, ... ]
1:  sin(x) ln(x)                    .
    .

    ' sin(x) ln(x) RET  s 1    m r  p 5 RET   V M $ RET


1:  3.4195     0.34195
    .          .

    V R +      0.1 *
```

(If you got wildly different results, did you remember to switch to radians mode?)

Here we have divided the curve into ten segments of equal width; approximating these segments as rectangular boxes (i.e., assuming the curve is nearly flat at that resolution), we compute the areas of the boxes (height times width), then sum the areas. (It is faster to sum first, then multiply by the width, since the width is the same for every box.)

The true value of this integral turns out to be about 0.374, so we're not doing too well. Let's try another approach.

```
1:  sin(x) ln(x)    1:  0.84147 x - 0.84147 + 0.11957 (x - 1)^2 - ...
    .                   .

    r 1                 a t x=1 RET 4 RET
```

Here we have computed the Taylor series expansion of the function about the point x=1. We can now integrate this polynomial approximation, since polynomials are easy to integrate.

```
1:  0.42074 x^2 + ...    1:  [-0.0446, -0.42073]      1:  0.3761
    .                        .                            .
```


```
    a i x RET           [ 2 , 1 ] TAB  V M $ RET        V R -
```

Better! By increasing the precision and/or asking for more terms in the Taylor series, we can get a result as accurate as we like. (Taylor series converge better away from singularities in the function such as the one at `ln(0)`, so it would also help to expand the series about the points x=2 or x=1.5 instead of x=1.)

(*) **Exercise 4.** Our first method approximated the curve by stairsteps of width 0.1; the total area was then the sum of the areas of the rectangles under these stairsteps. Our second method approximated the function by a polynomial, which turned out to be a better approximation than stairsteps. A third method is Simpson's rule, which is like the stairstep method except that the steps are not required to be flat. Simpson's rule boils down to the formula,

where n (which must be even) is the number of slices and h is the width of each slice. These are 10 and 0.1 in our example. For reference, here is the corresponding formula for the stairstep method:

Compute the integral from 1 to 2 of @c{$\sin x \ln x$} sin(x) ln(x) using Simpson's rule with 10 slices. See section Algebra Tutorial Exercise 4. (*)

Calc has a built-in a I command for doing numerical integration. It uses Romberg's method, which is a more sophisticated cousin of Simpson's rule. In particular, it knows how to keep refining the result until the current precision is satisfied.

Aside from the commands we've seen so far, Calc also provides a large set of commands for operating on parts of formulas. You indicate the desired sub-formula by placing the cursor on any part of the formula before giving a selection command. Selections won't be covered in the tutorial; see section Selecting Sub-Formulas, for details and examples.

Rewrite Rules

No matter how many built-in commands Calc provided for doing algebra, there would always be something you wanted to do that Calc didn't have in its repertoire. So Calc also provides a rewrite rule system that you can use to define your own algebraic manipulations.

Suppose we want to simplify this trigonometric formula:

```
1:  1 / cos(x) - sin(x) tan(x)
    .

    ' 1/cos(x) - sin(x) tan(x) RET   s 1
```

If we were simplifying this by hand, we'd probably replace the `tan' with a `sin/cos' first, then combine over a common denominator. There is no Calc command to do the former; the a n algebra command will do the latter but we'll do both with rewrite rules just for practice.

Rewrite rules are written with the `:=' symbol.

```
1:  1 / cos(x) - sin(x)^2 / cos(x)
    .
```


```
    a r tan(a) := sin(a)/cos(a) RET
```

(The "assignment operator" `:=' has several uses in Calc. All by itself the formula `tan(a) := sin(a)/cos(a)' doesn't do anything, but when it is given to the a r command, that command interprets it as a rewrite rule.)

The lefthand side, `tan(a)', is called the pattern of the rewrite rule. Calc searches the formula on the stack for parts that match the pattern. Variables in a rewrite pattern are called meta-variables, and when matching the pattern each meta-variable can match any sub-formula. Here, the meta-variable `a' matched the actual variable `x'.

When the pattern part of a rewrite rule matches a part of the formula, that part is replaced by the righthand side with all the meta-variables substituted with the things they matched. So the result is `sin(x) / cos(x)'. Calc's normal algebraic simplifications then mix this in with the rest of the original formula.

To merge over a common denominator, we can use another simple rule:

```
1:  (1 - sin(x)^2) / cos(x)
    .

    a r a/x + b/x := (a+b)/x RET
```

This rule points out several interesting features of rewrite patterns. First, if a meta-variable appears several times in a pattern, it must match the same thing everywhere. This rule detects common denominators because the same meta-variable `x' is used in both of the denominators.

Second, meta-variable names are independent from variables in the target formula. Notice that the meta-variable `x' here matches the subformula `cos(x)'; Calc never confuses the two meanings of `x'.

And third, rewrite patterns know a little bit about the algebraic properties of formulas. The pattern called for a sum of two quotients; Calc was able to match a difference of two quotients by matching `a = 1', `b = -sin(x)^2', and `x = cos(x)'.

We could just as easily have written `a/x - b/x := (a-b)/x' for the rule. It would have worked just the same in all cases. (If we really wanted the rule to apply only to `+' or only to `-', we could have used the `plain` symbol. See section Algebraic Properties of Rewrite Rules, for some examples of this.)

One more rewrite will complete the job. We want to use the identity `sin(x)^2 + cos(x)^2 = 1', but of course we must first rearrange the identity in a way that matches our formula. The obvious rule would be `1 - sin(x)^2 := cos(x)^2', but a little thought shows that the rule `sin(x)^2 := 1 - cos(x)^2' will also work. The latter rule has a more general pattern so it will work in many other situations, too.

```
1:  (1 + cos(x)^2 - 1) / cos(x)          1:  cos(x)
    .                                        .

    a r sin(x)^2 := 1 - cos(x)^2 RET         a s
```

You may ask, what's the point of using the most general rule if you have to type it in every time anyway? The answer is that Calc allows you to store a rewrite rule in a variable, then give the variable name in the a r command. In fact, this is the preferred way to use rewrites. For one, if you need a rule once you'll most likely need it again later. Also, if the rule doesn't work quite right you can simply Undo, edit the variable, and run the rule again without having to retype it.


```
' tan(x) := sin(x)/cos(x) RET      s t tsc RET
' a/x + b/x := (a+b)/x RET         s t merge RET
' sin(x)^2 := 1 - cos(x)^2 RET     s t sinsqr RET

1:  1 / cos(x) - sin(x) tan(x)     1:  cos(x)
    .                                  .

    r 1                a r tsc RET  a r merge RET  a r sinsqr RET  a s
```

To edit a variable, type s e and the variable name, use regular Emacs editing commands as necessary, then type M-# M-# or C-c C-c to store the edited value back into the variable. You can also use s e to create a new variable if you wish.

Notice that the first time you use each rule, Calc puts up a "compiling" message briefly. The pattern matcher converts rules into a special optimized pattern-matching language rather than using them directly. This allows a r to apply even rather complicated rules very efficiently. If the rule is stored in a variable, Calc compiles it only once and stores the compiled form along with the variable. That's another good reason to store your rules in variables rather than entering them on the fly.

(*) **Exercise 1.** Type m s to get symbolic mode, then enter the formula `(2 + sqrt(2)) / (1 + sqrt(2))'. Using a rewrite rule, simplify this formula by multiplying both sides by the conjugate `1 - sqrt(2)'. The result will have to be expanded by the distributive law; do this with another rewrite. See section Rewrites Tutorial Exercise 1. (*)

The a r command can also accept a vector of rewrite rules, or a variable containing a vector of rules.

```
1:  [tsc, merge, sinsqr]           1:  [tan(x) := sin(x) / cos(x), ... ]
    .                                  .

    ' [tsc,merge,sinsqr] RET           =


1:  1 / cos(x) - sin(x) tan(x)     1:  cos(x)
    .                                  .

    s t trig RET  r 1                  a r trig RET  a s
```

Calc tries all the rules you give against all parts of the formula, repeating until no further change is possible. (The exact order in which things are tried is rather complex, but for simple rules like the ones we've used here the order doesn't really matter. See section Nested Formulas with Rewrite Rules.)

Calc actually repeats only up to 100 times, just in case your rule set has gotten into an infinite loop. You can give a numeric prefix argument to a r to specify any limit. In particular, M-1 a r does only one rewrite at a time.

```
1:  1 / cos(x) - sin(x)^2 / cos(x)     1:  (1 - sin(x)^2) / cos(x)
    .                                      .
```


```
    r 1  M-1 a r trig RET                    M-1 a r trig RET
```

You can type M-0 a r if you want no limit at all on the number of rewrites that occur.

Rewrite rules can also be conditional. Simply follow the rule with a `::' symbol and the desired condition. For example,

```
1:  exp(2 pi i) + exp(3 pi i) + exp(4 pi i)
    .

    ' exp(2 pi i) + exp(3 pi i) + exp(4 pi i) RET


1:  1 + exp(3 pi i) + 1
    .

    a r exp(k pi i) := 1 :: k % 2 = 0 RET
```

(Recall, `k % 2' is the remainder from dividing `k' by 2, which will be zero only when `k' is an even integer.)

An interesting point is that the variables `pi' and `i' were matched literally rather than acting as meta-variables. This is because they are special-constant variables. The special constants `e', `phi', and so on also match literally. A common error with rewrite rules is to write, say, `f(a,b,c,d,e) := g(a+b+c+d+e)', expecting to match any `f' with five arguments but in fact matching only when the fifth argument is literally `e'!

Rewrite rules provide an interesting way to define your own functions. Suppose we want to define `fib(n)' to produce the nth Fibonacci number. The first two Fibonacci numbers are each 1; later numbers are formed by summing the two preceding numbers in the sequence. This is easy to express in a set of three rules:

```
' [fib(1) := 1, fib(2) := 1, fib(n) := fib(n-1) + fib(n-2)] RET  s t fib

1:  fib(7)                1:  13
    .                         .

    ' fib(7) RET              a r fib RET
```

One thing that is guaranteed about the order that rewrites are tried is that, for any given subformula, earlier rules in the rule set will be tried for that subformula before later ones. So even though the first and third rules both match `fib(1)', we know the first will be used preferentially.

This rule set has one dangerous bug: Suppose we apply it to the formula `fib(x)'? (Don't actually try this.) The third rule will match `fib(x)' and replace it with `fib(x-1) + fib(x-2)'. Each of these will then be replaced to get `fib(x-2) + 2 fib(x-3) + fib(x-4)', and so on, expanding forever. What we really want is to apply the third rule only when `n' is an integer greater than two. Type s e fib RET, then edit the third rule to:

```
fib(n) := fib(n-1) + fib(n-2) :: integer(n) :: n > 2
```

Now:

```
1:  fib(6) + fib(x) + fib(0)      1:  8 + fib(x) + fib(0)
```


```
    .                                   .

    ' fib(6)+fib(x)+fib(0) RET          a r fib RET
```

We've created a new function, `fib`, and a new command, a r fib RET, which means "evaluate all `fib` calls in this formula." To make things easier still, we can tell Calc to apply these rules automatically by storing them in the special variable `EvalRules`.

```
1:  [fib(1) := ...]     .                   1:  [8, 13]
    .                                           .

    s r fib RET         s t EvalRules RET    ' [fib(6), fib(7)] RET
```

It turns out that this rule set has the problem that it does far more work than it needs to when `n' is large. Consider the first few steps of the computation of `fib(6)':

```
fib(6) =
fib(5)                +                fib(4) =
fib(4)     +      fib(3)     +      fib(3)     +      fib(2) =
fib(3) + fib(2) + fib(2) + fib(1) + fib(2) + fib(1) + 1 = ...
```

Note that `fib(3)' appears three times here. Unless Calc's algebraic simplifier notices the multiple `fib(3)'s and combines them (and, as it happens, it doesn't), this rule set does lots of needless recomputation. To cure the problem, type `s e EvalRules` to edit the rules (or just s E, a shorthand command for editing `EvalRules`) and add another condition:

```
fib(n) := fib(n-1) + fib(n-2) :: integer(n) :: n > 2 :: remember
```

If a `:: remember' condition appears anywhere in a rule, then if that rule succeeds Calc will add another rule that describes that match to the front of the rule set. (Remembering works in any rule set, but for technical reasons it is most effective in `EvalRules`.) For example, if the rule rewrites `fib(7)' to something that evaluates to 13, then the rule `fib(7) := 13' will be added to the rule set.

Type ' fib(8) RET to compute the eighth Fibonacci number, then type s E again to see what has happened to the rule set.

With the `remember` feature, our rule set can now compute `fib(n)' in just n steps. In the process it builds up a table of all Fibonacci numbers up to n. After we have computed the result for a particular n, we can get it back (and the results for all smaller n) later in just one step.

All Calc operations will run somewhat slower whenever `EvalRules` contains any rules. You should type s u EvalRules RET now to un-store the variable.

(*) **Exercise 2.** Sometimes it is possible to reformulate a problem to reduce the amount of recursion necessary to solve it. Create a rule that, in about n simple steps and without recourse to the `remember` option, replaces `fib(n, 1, 1)' with `fib(1, x, y)' where x and y are the nth and n+1st Fibonacci numbers, respectively. This rule is rather clunky to use, so add a couple more rules to make the "user interface" the same as for our first version: enter `fib(n)', get back a plain number. See section Rewrites Tutorial Exercise 2. (*)

There are many more things that rewrites can do. For example, there are `&&&' and `|||' pattern operators that

create "and" and "or" combinations of rules. As one really simple example, we could combine our first tw Fibonacci rules thusly:

```
[fib(1 ||| 2) := 1, fib(n) := ... ]
```

That means "`fib` of something matching either 1 or 2 rewrites to 1."

You can also make meta-variables optional by enclosing them in `opt`. For example, the pattern `a + b x' matches `2 + 3 x' but not `2 + x' or `3 x' or `x'. The pattern `opt(a) + opt(b) x' matches all of these forms, filling in a default of zero for `a' and one for `b'.

(*) **Exercise 3.** Your friend Joe had `2 + 3 x' on the stack and tried to use the rule `opt(a) + opt(b) x := f(a, b, x)'. What happened? See section Rewrites Tutorial Exercise 3. (*)

(*) **Exercise 4.** Starting with a positive integer a, divide a by two if it is even, otherwise compute 3 a + 1. Now repeat this step over and over. A famous unproved conjecture is that for any starting a, the sequence always eventually reaches 1. Given the formula `seq(a, 0)', write a set of rules that convert this into `seq(1, n)' where n is the number of steps it took the sequence to reach the value 1. Now enhance the rules to accept `seq(a)' as a starting configuration, and to stop with just the number n by itself. Now make the result be a vector of values in the sequence, from a to 1. (The formula `x|y' appends the vectors x and y.) For example, rewriting `seq(6)' should yield the vector [6, 3, 10, 5, 16, 8, 4, 2, 1]. See section Rewrites Tutorial Exercise 4. (*)

(*) **Exercise 5.** Define, using rewrite rules, a function `nterms(x)' that returns the number of terms in the sum x, or 1 if x is not a sum. (A sum for our purposes is one or more non-sum terms separated by `+' or `-' signs, so that 2 - 3 (x + y) + x y is a sum of three terms.) See section Rewrites Tutorial Exercise 5. (*)

(*) **Exercise 6.** Calc considers the form 0^0 to be "indeterminate," and leaves it unevaluated (assuming infinite mode is not enabled). Some people prefer to define 0^0 = 1, so that the identity x^0 = 1 can safely be used for all x. Find a way to make Calc follow this convention. What happens if you now type m i to turn on infinite mode? See section Rewrites Tutorial Exercise 6. (*)

(*) **Exercise 7.** A Taylor series for a function is an infinite series that exactly equals the value of that function at values of x near zero.

The a t command produces a truncated Taylor series which is obtained by dropping all the terms higher than, say, x^2. Calc represents the truncated Taylor series as a polynomial in x. Mathematicians often write a truncated series using a "big-O" notation that records what was the lowest term that was truncated.

The meaning of O(x^3) is "a quantity which is negligibly small if x^3 is considered negligibly small as x goes to zero."

The exercise is to create rewrite rules that simplify sums and products of power series represented as `polynomial + O(var^n)'. For example, given `1 - x^2 / 2 + O(x^3)' and `x - x^3 / 6 + O(x^4)' on the stack, we want to be able to type * and get the result `x - 2:3 x^3 + O(x^4)'. Don't worry if the terms of the sum are rearranged or if a s needs to be typed after rewriting. (This one is rather tricky; the solution at the end of this chapter uses 6 rewrite rules. Hint: The `constant(x)' condition tests whether `x' is a number.) See section Rewrites Tutorial Exercise 7. (*)

See section Rewrite Rules, for the whole story on rewrite rules.

Programming Tutorial

The Calculator is written entirely in Emacs Lisp, a highly extensible language. If you know Lisp, you can program the Calculator to do anything you like. Rewrite rules also work as a powerful programming system. But Lisp and rewrite rules take a while to master, and often all you want to do is define a new function or repeat a command a few times. Calc has features that allow you to do these things easily.

One very limited form of programming is defining your own functions. Calc's Z F command allows you to define a function name and key sequence to correspond to any formula. Programming commands use the shift-Z prefix; the user commands they create use the lower case z prefix.

```
1:  1 + x + x^2 / 2 + x^3 / 6          1:  1 + x + x^2 / 2 + x^3 / 6
    .                                      .

    ' 1 + x + x^2/2! + x^3/3! RET          Z F e myexp RET RET RET y
```

This polynomial is a Taylor series approximation to `exp(x)'. The Z F command asks a number of questions. The above answers say that the key sequence for our function should be z e; the M-x equivalent should be `calc-myexp`; the name of the function in algebraic formulas should also be `myexp`; the default argument list `(x)' is acceptable; and finally y answers the question "leave it in symbolic form for non-constant arguments?"

```
1:  1.3495     2:  1.3495     3:  1.3495
    .          1:  1.34986    2:  1.34986
                   .          1:  myexp(a + 1)
                                  .

    .3 z e         .3 E           ' a+1 RET z e
```

First we call our new `exp` approximation with 0.3 as an argument, and compare it with the true `exp` function. Then we note that, as requested, if we try to give z e an argument that isn't a plain number, it leaves the `myexp` function call in symbolic form. If we had answered n to the final question, `myexp(a + 1)' would have evaluated by plugging in `a + 1' for `x' in the defining formula.

(*) **Exercise 1.** The "sine integral" function Si(x) is defined as the integral of `sin(t)/t' for t = 0 to x in radians. (It was invented because this integral has no solution in terms of basic functions; if you give it to Calc's a i command, it will ponder it for a long time and then give up.) We can use the numerical integration command, however, which in algebraic notation is written like `ninteg(f(t), t, 0, x)' with any integrand `f(t)'. Define a z s command and `Si` function that implement this. You will need to edit the default argument list a bit. As a test, `Si(1)' should return 0.946083. (Hint: `ninteg` will run a lot faster if you reduce the precision to, say, six digits beforehand.) See section Programming Tutorial Exercise 1. (*)

The simplest way to do real "programming" of Emacs is to define a keyboard macro. A keyboard macro is simply a sequence of keystrokes which Emacs has stored away and can play back on demand. For example, if you find yourself typing H a S x RET often, you may wish to program a keyboard macro to type this for you.

```
1:  y = sqrt(x)          1:  x = y^2
    .                        .
```


```
    ' y=sqrt(x) RET       C-x ( H a S x RET C-x )

1:  y = cos(x)          1:  x = s1 arccos(y) + 2 pi n1
    .                       .

    ' y=cos(x) RET          X
```

When you type C-x (, Emacs begins recording. But it is also still ready to execute your keystrokes, so you're really "training" Emacs by walking it through the procedure once. When you type C-x), the macro is recorded. You can now type X to re-execute the same keystrokes.

You can give a name to your macro by typing Z K.

```
1:  .             1:  y = x^4         1:  x = s2 sqrt(s1 sqrt(y))
                      .                   .

  Z K x RET           ' y=x^4 RET         z x
```

Notice that we use shift-Z to define the command, and lower-case z to call it up.

Keyboard macros can call other macros.

```
1:  abs(x)        1:  x = s1 y              1:  2 / x    1:  x = 2 / y
    .                 .                         .            .

 ' abs(x) RET   C-x ( ' y RET a = z x C-x )    ' 2/x RET       X
```

(*) **Exercise 2.** Define a keyboard macro to negate the item in level 3 of the stack, without disturbing the rest of the stack. See section Programming Tutorial Exercise 2. (*)

(*) **Exercise 3.** Define keyboard macros to compute the following functions:

1. Compute @c{$\displaystyle{\sin x \over x}$} sin(x) / x, where x is the number on the top of the stack.
2. Compute the base-b logarithm, just like the B key except the arguments are taken in the opposite order.
3. Produce a vector of integers from 1 to the integer on the top of the stack.

See section Programming Tutorial Exercise 3. (*)

(*) **Exercise 4.** Define a keyboard macro to compute the average (mean) value of a list of numbers. See section Programming Tutorial Exercise 4. (*)

In many programs, some of the steps must execute several times. Calc has looping commands that allow this. Loops are useful inside keyboard macros, but actually work at any time.

```
1:  x^6          2:  x^6        1: 360 x^2
    .            1:  4             .
                     .

  ' x^6 RET          4         Z < a d x RET Z >
```

Here we have computed the fourth derivative of x^6 by enclosing a derivative command in a "repeat loop" structure. This structure pops a repeat count from the stack, then executes the body of the loop that many times.

If you make a mistake while entering the body of the loop, type Z C-g to cancel the loop command.

Here's another example:

```
3:  1               2:  10946
2:  1               1:  17711
1:  20                  .
    .

1 RET RET 20        Z < TAB C-j + Z >
```

The numbers in levels 2 and 1 should be the 21st and 22nd Fibonacci numbers, respectively. (To see what's going on, try a few repetitions of the loop body by hand; C-j, also on the Line-Feed or LFD key if you have one, makes a copy of the number in level 2.)

A fascinating property of the Fibonacci numbers is that the nth Fibonacci number can be found directly by computing @c{$\phi^n / \sqrt{5}$} phi^n / sqrt(5) and then rounding to the nearest integer, where @c{ϕ} ("phi")} phi, the "golden ratio," is @c{$(1 + \sqrt{5}) / 2$} (1 + sqrt(5)) / 2. (For convenience, this constant is available from the `phi` variable, or the I H P command.)

```
1:  1.61803         1:  24476.0000409    1:  10945.9999817    1:  10946
    .                   .                    .                    .

    I H P               21 ^                 5 Q /                R
```

(*) **Exercise 5.** The continued fraction representation of @c{ϕ} phi is @c{$1 + 1/(1 + 1/(1 + 1/(\ldots)))$} 1 + 1/(1 + 1/(1 + 1/(...))). We can compute an approximate value by carrying this however far and then replacing the innermost @c{$1/(\ldots)$} 1/(...) by 1. Approximate phi using a twenty-term continued fraction. See section Programming Tutorial Exercise 5. (*)

(*) **Exercise 6.** Linear recurrences like the one for Fibonacci numbers can be expressed in terms of matrices. Given a vector [a, b] determine a matrix which, when multiplied by this vector, produces the vector [b, c], where a, b and c are three successive Fibonacci numbers. Now write a program that, given an integer n, computes the nth Fibonacci number using matrix arithmetic. See section Programming Tutorial Exercise 6. (*)

A more sophisticated kind of loop is the for loop. Suppose we wish to compute the 20th "harmonic" number, which is equal to the sum of the reciprocals of the integers from 1 to 20.

```
3:  0               1:  3.597739
2:  1                   .
1:  20
    .

0 RET 1 RET 20          Z ( & + 1 Z )
```

The "for" loop pops two numbers, the lower and upper limits, then repeats the body of the loop as an internal

counter increases from the lower limit to the upper one. Just before executing the loop body, it pushes the current loop counter. When the loop body finishes, it pops the "step," i.e., the amount by which to increment the loop counter. As you can see, our loop always uses a step of one.

This harmonic number function uses the stack to hold the running total as well as for the various loop housekeeping functions. If you find this disorienting, you can sum in a variable instead:

```
1:  0          2:  1                    .             1:  3.597739
    .          1:  20                                     .
                   .

    0 t 7        1 RET 20       Z ( & s + 7 1 Z )        r 7
```

The s + command adds the top-of-stack into the value in a variable (and removes that value from the stack).

It's worth noting that many jobs that call for a "for" loop can also be done more easily by Calc's high-level operations. Two other ways to compute harmonic numbers are to use vector mapping and reduction (v x 20, then V M &, then V R +), or to use the summation command a +. Both of these are probably easier than using loops. However, there are some situations where loops really are the way to go:

(*) **Exercise 7.** Use a "for" loop to find the first harmonic number which is greater than 4.0. See section Programming Tutorial Exercise 7. (*)

Of course, if we're going to be using variables in our programs, we have to worry about the programs clobbering values that the caller was keeping in those same variables. This is easy to fix, though:

```
    .          1:  0.6667        1:  0.6667      3:  0.6667
               .                   .               2:  3.597739
                                                   1:  0.6667
                                                       .

   Z `     p 4 RET 2 RET 3 /    s 7 s s a RET     Z '   r 7 s r a RET
```

When we type Z ` (that's a back-quote character), Calc saves its mode settings and the contents of the ten "quick variables" for later reference. When we type Z ' (that's an apostrophe now), Calc restores those saved values. Thus the p 4 and s 7 commands have no effect outside this sequence. Wrapping this around the body of a keyboard macro ensures that it doesn't interfere with what the user of the macro was doing. Notice that the contents of the stack, and the values of named variables, survive past the Z ' command.

The Bernoulli numbers are a sequence with the interesting property that all of the odd Bernoulli numbers are zero, and the even ones, while difficult to compute, can be roughly approximated by the formula @c{$\displaystyle{2 n! \over (2 \pi)^n}$} 2 n! / (2 pi)^n. Let's write a keyboard macro to compute (approximate) Bernoulli numbers. (Calc has a command, k b, to compute exact Bernoulli numbers, but this command is very slow for large n since the higher Bernoulli numbers are very large fractions.)

```
1:  10                1:  0.0756823
    .                     .

    10      C-x ( RET 2 % Z [ DEL 0 Z : ' 2 $! / (2 pi)^$ RET = Z ] C-x )
```

You can read Z [as "then," Z : as "else," and Z] as "end-if." There is no need for an explicit "if" comman
For the purposes of Z [, the condition is "true" if the value it pops from the stack is a nonzero number, or
"false" if it pops zero or something that is not a number (like a formula). Here we take our integer argument
modulo 2; this will be nonzero if we're asking for an odd Bernoulli number.

The actual tenth Bernoulli number is 5/66.

```
3:  0.0756823    1:  0          1:  0.25305    1:  0          1:  1.16659
2:  5:66             .              .              .              .
1:  0.0757575
    .

10 k b RET c f   M-0 DEL 11 X   DEL 12 X       DEL 13 X       DEL 14 X
```

Just to exercise loops a bit more, let's compute a table of even Bernoulli numbers.

```
3:  []             1:  [0.10132, 0.03079, 0.02340, 0.033197, ...]
2:  2                  .
1:  30
    .

 [ ] 2 RET 30          Z ( X | 2 Z )
```

The vertical-bar | is the vector-concatenation command. When we execute it, the list we are building will be in stack level 2 (initially this is an empty list), and the next Bernoulli number will be in level 1. The effect is to append the Bernoulli number onto the end of the list. (To create a table of exact fractional Bernoulli numbers, just replace X with k b in the above sequence of keystrokes.)

With loops and conditionals, you can program essentially anything in Calc. One other command that makes looping easier is Z /, which takes a condition from the stack and breaks out of the enclosing loop if the condition is true (non-zero). You can use this to make "while" and "until" style loops.

If you make a mistake when entering a keyboard macro, you can edit it using Z E. First, you must attach it to a key with Z K. One technique is to enter a throwaway dummy definition for the macro, then enter the real one in the edit command.

```
1:  3                  1:  3          Keyboard Macro Editor.
    .                      .          Original keys: 1 RET 2 +

                                      type "1\r"
                                      type "2"
                                      calc-plus

C-x ( 1 RET 2 + C-x )    Z K h RET    Z E h
```

This shows the screen display assuming you have the `macedit' keyboard macro editing package installed, which is usually the case since a copy of `macedit' comes bundled with Calc.

A keyboard macro is stored as a pure keystroke sequence. The `macedit' package (invoked by Z E) scans along the macro and tries to decode it back into human-readable steps. If a key or keys are simply shorthand

for some command with a M-x name, that name is shown. Anything that doesn't correspond to a M-x command is written as a `type' command.

Let's edit in a new definition, for computing harmonic numbers. First, erase the three lines of the old definition. Then, type in the new definition (or use Emacs M-w and C-y commands to copy it from this page of the Info file; you can skip typing the comments that begin with `#').

```
calc-kbd-push          # Save local values (Z `)
type "0"               # Push a zero
calc-store-into        # Store it in variable 1
type "1"
type "1"               # Initial value for loop
calc-roll-down         # This is the TAB key; swap initial & final
calc-kbd-for           # Begin "for" loop...
calc-inv               #   Take reciprocal
calc-store-plus        #   Add to accumulator
type "1"
type "1"               #   Loop step is 1
calc-kbd-end-for       # End "for" loop
calc-recall            # Now recall final accumulated value
type "1"
calc-kbd-pop           # Restore values (Z ')
```

Press M-# M-# to finish editing and return to the Calculator.

```
1:  20          1:  3.597739
    .               .

    20              z h
```

If you don't know how to write a particular command in `macedit' format, you can always write it as keystrokes in a `type' command. There is also a `keys' command which interprets the rest of the line as standard Emacs keystroke names. In fact, `macedit' defines a handy `read-kbd-macro' command which reads the current region of the current buffer as a sequence of keystroke names, and defines that sequence on the X (and C-x e) key. Because this is so useful, Calc puts this command on the M-# m key. Try reading in this macro in the following form: Press C-@ (or C-SPC) at one end of the text below, then type M-# m at the other.

```
Z ` 0 t 1
    1 TAB
    Z (  & s + 1  1 Z )
    r 1
Z '
```

(*) **Exercise 8.** A general algorithm for solving equations numerically is Newton's Method. Given the equation f(x) = 0 for any function f, and an initial guess x_0 which is reasonably close to the desired solution, apply this formula over and over:

where f'(x) is the derivative of f. The x values will quickly converge to a solution, i.e., eventually new_x and x

will be equal to within the limits of the current precision. Write a program which takes a formula involvin variable x, and an initial guess x_0, on the stack, and produces a value of x for which the formula is zero it to find a solution of @c{$\sin(\cos x) = 0.5$} sin(cos(x)) = 0.5 near x = 4.5. (Use angles measured in radians.) Note that the built-in a R (`calc-find-root`) command uses Newton's method when it is able. See section Programming Tutorial Exercise 8. (*)

(*) **Exercise 9.** The digamma function @c{$\psi(z)$ ("psi")} psi(z) is defined as the derivative of @c{$\ln \Gamma(z)$} ln(gamma(z)). For large values of z, it can be approximated by the infinite sum

where @c{$\sum$} sum represents the sum over n from 1 to infinity (or to some limit high enough to give the desired accuracy), and the `bern` function produces (exact) Bernoulli numbers. While this sum is not guaranteed to converge, in practice it is safe. An interesting mathematical constant is Euler's gamma, which is equal to about 0.5772. One way to compute it is by the formula, gamma = -psi(1). Unfortunately, 1 isn't a large enough argument for the above formula to work (5 is a much safer value for z). Fortunately, we can compute @c{$\psi(1)$} psi(1) from @c{$\psi(5)$} psi(5) using the recurrence @c{$\psi(z+1) = \psi(z) + {1 \over z}$} psi(z+1) = psi(z) + 1/z. Your task: Develop a program to compute @c{$\psi(z)$} psi(z); it should "pump up" z if necessary to be greater than 5, then use the above summation formula. Use looping commands to compute the sum. Use your function to compute @c{γ} gamma to twelve decimal places. (Calc has a built-in command for Euler's constant, I P, which you can use to check your answer.) See section Programming Tutorial Exercise 9. (*)

(*) **Exercise 10.** Given a polynomial in x and a number m on the stack, where the polynomial is of degree m or less (i.e., does not have any terms higher than x^m), write a program to convert the polynomial into a list-of-coefficients notation. For example, 5 x^4 + (x + 1)^2 with m = 6 should produce the list [1, 2, 1, 0, 5, 0, 0]. Also develop a way to convert from this form back to the standard algebraic form. See section Programming Tutorial Exercise 10. (*)

(*) **Exercise 11.** The Stirling numbers of the first kind are defined by the recurrences,

This can be implemented using a recursive program in Calc; the program must invoke itself in order to calculate the two righthand terms in the general formula. Since it always invokes itself with "simpler" arguments, it's easy to see that it must eventually finish the computation. Recursion is a little difficult with Emacs keyboard macros since the macro is executed before its definition is complete. So here's the recommended strategy: Create a "dummy macro" and assign it to a key with, e.g., Z K s. Now enter the true definition, using the z s command to call itself recursively, then assign it to the same key with Z K s. Now the z s command will run the complete recursive program. (Another way is to use Z E or M-# m (`read-kbd-macro`) to read the whole macro at once, thus avoiding the "training" phase.) The task: Write a program that computes Stirling numbers of the first kind, given n and m on the stack. Test it with *small* inputs like s(4,2). (There is a built-in command for Stirling numbers, k s, which you can use to check your answers.) See section Programming Tutorial Exercise 11. (*)

The programming commands we've seen in this part of the tutorial are low-level, general-purpose operations. Often you will find that a higher-level function, such as vector mapping or rewrite rules, will do the job much more easily than a detailed, step-by-step program can:

(*) **Exercise 12.** Write another program for computing Stirling numbers of the first kind, this time using rewrite rules. Once again, n and m should be taken from the stack. See section Programming Tutorial Exercise 12. (*)

This ends the tutorial section of the Calc manual. Now you know enough about Calc to use it effectively for many kinds of calculations. But Calc has many features that were not even touched upon in this tutorial. The rest of this manual tells the whole story.

Answers to Exercises

This section includes answers to all the exercises in the Calc tutorial.

RPN Tutorial Exercise 1

1 RET 2 RET 3 RET 4 + * -

The result is @c{$1 - (2 \times (3 + 4)) = -13$} 1 - (2 * (3 + 4)) = -13.

RPN Tutorial Exercise 2

2*4 + 7*9.5 + 5/4 = 75.75

After computing the intermediate term @c{$2\times4 = 8$} 2*4 = 8, you can leave that result on the stack while you compute the second term. With both of these results waiting on the stack you can then compute the final term, then press + + to add everything up.

```
2:  2          1:  8          3:  8          2:  8
1:  4              .          2:  7          1:  66.5
    .                         1:  9.5            .
                                  .

  2 RET 4          *          7 RET 9.5          *


4:  8          3:  8          2:  8          1:  75.75
3:  66.5       2:  66.5       1:  67.75          .
2:  5          1:  1.25           .
1:  4              .
    .

  5 RET 4          /              +              +
```

Alternatively, you could add the first two terms before going on with the third term.

```
2:  8          1:  74.5       3:  74.5       2:  74.5       1:  75.75
1:  66.5           .          2:  5          1:  1.25           .
    .                         1:  4              .
                                  .

   ...             +            5 RET 4          /              +
```

On an old-style RPN calculator this second method would have the advantage of using only three stack le
But since Calc's stack can grow arbitrarily large this isn't really an issue. Which method you choose is pu
matter of taste.

RPN Tutorial Exercise 3

The TAB key provides a way to operate on the number in level 2.

```
3:  10          3:  10          4:  10          3:  10          3:  10
2:  20          2:  30          3:  30          2:  30          2:  21
1:  30          1:  20          2:  20          1:  21          1:  30
    .               .           1:  1               .               .
                                    .

                   TAB              1              +               TAB
```

Similarly, M-TAB gives you access to the number in level 3.

```
3:  10          3:  21          3:  21          3:  30          3:  11
2:  21          2:  30          2:  30          2:  11          2:  21
1:  30          1:  10          1:  11          1:  21          1:  30
    .               .               .               .               .

                   M-TAB            1 +             M-TAB           M-TAB
```

RPN Tutorial Exercise 4

Either (2 , 3) or (2 SPC 3) would have worked, but using both the comma and the space at once yields:

```
1:  ( ...       2:  ( ...       1:  (2, ...     2:  (2, ...     2:  (2, ...
    .           1:  2               .           1:  (2, ...     1:  (2, 3)
                    .                               .               .

    (               2               ,               SPC             3 )
```

Joe probably tried to type TAB DEL to swap the extra incomplete object to the top of the stack and delete it. But a feature of Calc is that DEL on an incomplete object deletes just one component out of that object, so he had to press DEL twice to finish the job.

```
2:  (2, ...     2:  (2, 3)      2:  (2, 3)      1:  (2, 3)
1:  (2, 3)      1:  (2, ...     1:  ( ...           .
    .               .               .

                   TAB             DEL             DEL
```

(As it turns out, deleting the second-to-top stack entry happens often enough that Calc provides a special key, M-DEL, to do just that. M-DEL is just like TAB DEL, except that it doesn't exhibit the "feature" that tripped

poor Joe.)

Algebraic Entry Tutorial Exercise 1

Type ' sqrt($) RET.

If the Q key is broken, you could use ' $^0.5 RET. Or, RPN style, 0.5 ^.

(Actually, `$^1:2', using the fraction one-half as the power, is a closer equivalent, since `9^0.5' yields 3.0 whereas `sqrt(9)' and `9^1:2' yield the exact integer 3.)

Algebraic Entry Tutorial Exercise 2

In the formula `2 x (1+y)', `x' was interpreted as a function name with `1+y' as its argument. Assigning a value to a variable has no relation to a function by the same name. Joe needed to use an explicit `*' symbol here: `2 x*(1+y)'.

Algebraic Entry Tutorial Exercise 3

The result from 1 RET 0 / will be the formula 1 / 0. The "function" `/' cannot be evaluated when its second argument is zero, so it is left in symbolic form. When you now type 0 *, the result will be zero because Calc uses the general rule that "zero times anything is zero."

The m i command enables an infinite mode in which 1 / 0 results in a special symbol that represents "infinity." If you multiply infinity by zero, Calc uses another special new symbol to show that the answer is "indeterminate." See section Infinities, for further discussion of infinite and indeterminate values.

Modes Tutorial Exercise 1

Calc always stores its numbers in decimal, so even though one-third has an exact base-3 representation (`3#0.1'), it is still stored as 0.3333333 (chopped off after 12 or however many decimal digits) inside the calculator's memory. When this inexact number is converted back to base 3 for display, it may still be slightly inexact. When we multiply this number by 3, we get 0.999999, also an inexact value.

When Calc displays a number in base 3, it has to decide how many digits to show. If the current precision is 12 (decimal) digits, that corresponds to `12 / log10(3) = 25.15' base-3 digits. Because 25.15 is not an exact integer, Calc shows only 25 digits, with the result that stored numbers carry a little bit of extra information that may not show up on the screen. When Joe entered `3#0.2', the stored number 0.666666 happened to round to a pleasing value when it lost that last 0.15 of a digit, but it was still inexact in Calc's memory. When he divided by 2, he still got the dreaded inexact value 0.333333. (Actually, he divided 0.666667 by 2 to get 0.333334, which is why he got something a little higher than `3#0.1` instead of a little lower.)

If Joe didn't want to be bothered with all this, he could have typed M-24 d n to display with one less digit than the default. (If you give d n a negative argument, it uses default-minus-that, so M-- d n would be an easier way to get the same effect.) Those inexact results would still be lurking there, but they would now be rounded to nice, natural-looking values for display purposes. (Remember, `0.022222' in base 3 is like `0.099999' in base 10; rounding off one digit will round the number up to `0.1'.) Depending on the nature of your work, this hiding of the inexactness may be a benefit or a danger. With the d n command, Calc gives you the choice.

Incidentally, another consequence of all this is that if you type M-30 d n to display more digits than are "r there," you'll see garbage digits at the end of the number. (In decimal display mode, with decimally-stored numbers, these garbage digits are always zero so they vanish and you don't notice them.) Because Calc rounds off that 0.15 digit, there is the danger that two numbers could be slightly different internally but still look the same. If you feel uneasy about this, set the d n precision to be a little higher than normal; you'll get ugly garbage digits, but you'll always be able to tell two distinct numbers apart.

An interesting side note is that most computers store their floating-point numbers in binary, and convert to decimal for display. Thus everyday programs have the same problem: Decimal 0.1 cannot be represented exactly in binary (try it: 0.1 d 2), so `0.1 * 10' comes out as an inexact approximation to 1 on some machines (though they generally arrange to hide it from you by rounding off one digit as we did above). Because Calc works in decimal instead of binary, you can be sure that numbers that look exact *are* exact as long as you stay in decimal display mode.

It's not hard to show that any number that can be represented exactly in binary, octal, or hexadecimal is also exact in decimal, so the kinds of problems we saw in this exercise are likely to be severe only when you use a relatively unusual radix like 3.

Modes Tutorial Exercise 2

If the radix is 15 or higher, we can't use the letter `e' to mark the exponent because `e' is interpreted as a digit. When Calc needs to display scientific notation in a high radix, it writes `16#F.E8F*16.^15'. You can enter a number like this as an algebraic entry. Also, pressing e without any digits before it normally types 1e, but in a high radix it types 16.^ and puts you in algebraic entry: 16#f.e8f RET e 15 RET * is another way to enter this number.

The reason Calc puts a decimal point in the `16.^' is to prevent huge integers from being generated if the exponent is large (consider `16#1.23*16^1000', where we compute `16^1000' as a giant exact integer and then throw away most of the digits when we multiply it by the floating-point `16#1.23'). While this wouldn't normally matter for display purposes, it could give you a nasty surprise if you copied that number into a file and later moved it back into Calc.

Modes Tutorial Exercise 3

The answer he got was 0.5000000000006399.

The problem is not that the square operation is inexact, but that the sine of 45 that was already on the stack was accurate to only 12 places. Arbitrary-precision calculations still only give answers as good as their inputs.

The real problem is that there is no 12-digit number which, when squared, comes out to 0.5 exactly. The f [and f] commands decrease or increase a number by one unit in the last place (according to the current precision). They are useful for determining facts like this.

```
1:  0.707106781187      1:  0.500000000001
    .                       .

    45 S                    2 ^
```


```
1:  0.707106781187      1:  0.707106781186      1:  0.499999999999
    .                       .                       .

    U  DEL                  f [                     2 ^
```

A high-precision calculation must be carried out in high precision all the way. The only number in the original problem which was known exactly was the quantity 45 degrees, so the precision must be raised before anything is done after the number 45 has been entered in order for the higher precision to be meaningful.

Modes Tutorial Exercise 4

Many calculations involve real-world quantities, like the width and height of a piece of wood or the volume of a jar. Such quantities can't be measured exactly anyway, and if the data that is input to a calculation is inexact, doing exact arithmetic on it is a waste of time.

Fractions become unwieldy after too many calculations have been done with them. For example, the sum of the reciprocals of the integers from 1 to 10 is 7381:2520. The sum from 1 to 30 is 9304682830147:2329089562800. After a point it will take a long time to add even one more term to this sum, but a floating-point calculation of the sum will not have this problem.

Also, rational numbers cannot express the results of all calculations. There is no fractional form for the square root of two, so if you type 2 Q, Calc has no choice but to give you a floating-point answer.

Arithmetic Tutorial Exercise 1

Dividing two integers that are larger than the current precision may give a floating-point result that is inaccurate even when rounded down to an integer. Consider 123456789 / 2 when the current precision is 6 digits. The true answer is 61728394.5, but with a precision of 6 this will be rounded to @c{$12345700.0/2.0 = 61728500.0$} 12345700. / 2. = 61728500.. The result, when converted to an integer, will be off by 106.

Here are two solutions: Raise the precision enough that the floating-point round-off error is strictly to the right of the decimal point. Or, convert to fraction mode so that 123456789 / 2 produces the exact fraction 123456789:2, which can be rounded down by the F command without ever switching to floating-point format.

Arithmetic Tutorial Exercise 2

27 RET 9 B could give the exact result 3:2, but it does a floating-point calculation instead and produces 1.5.

Calc will find an exact result for a logarithm if the result is an integer or the reciprocal of an integer. But there is no efficient way to search the space of all possible rational numbers for an exact answer, so Calc doesn't try.

Vector Tutorial Exercise 1

Duplicate the vector, compute its length, then divide the vector by its length: RET A /.

```
1:  [1, 2, 3]  2:  [1, 2, 3]      1:  [0.27, 0.53, 0.80]  1:  1.
    .          1:  3.74165738677      .                       .
                   .
```


```
    r 1             RET A                 /                          A
```

The final A command shows that the normalized vector does indeed have unit length.

Vector Tutorial Exercise 2

The average position is equal to the sum of the products of the positions times their corresponding probabilities. This is the definition of the dot product operation. So all you need to do is to put the two vectors on the stack and press *.

Matrix Tutorial Exercise 1

The trick is to multiply by a vector of ones. Use r 4 [1 1 1] * to get the row sum. Similarly, use [1 1] r 4 * to get the column sum.

Matrix Tutorial Exercise 2

Just enter the righthand side vector, then divide by the lefthand side matrix as usual.

```
1:  [6, 10]    2:  [6, 10]          1:  [6 - 4 a / (b - a), 4 / (b - a) ]
    .          1:  [ [ 1, a ]           .
                     [ 1, b ] ]
                   .

' [6 10] RET     ' [1 a; 1 b] RET      /
```

This can be made more readable using d B to enable "big" display mode:

```
           4 a     4
1:  [6 - -----, -----]
         b - a   b - a
```

Type d N to return to "normal" display mode afterwards.

Matrix Tutorial Exercise 3

To solve @c{$A^T A \, X = A^T B$} trn(A) * A * X = trn(A) * B, first we compute A2 = trn(A) * A and @c{$B' = A^T B$} B2 = trn(A) * B; now, we have a system @c{$A' X = B'$} A2 * X = B2 which we can solve using Calc's `/' command.

The first step is to enter the coefficient matrix. We'll store it in quick variable number 7 for later reference. Next, we compute the B2 vector.

```
1:  [ [ 1, 2, 3 ]              2:  [ [ 1, 4, 7, 2 ]      1:  [57, 84, 96]
      [ 4, 5, 6 ]                    [ 2, 5, 6, 4 ]          .
      [ 7, 6, 0 ]                    [ 3, 6, 0, 6 ] ]
      [ 2, 4, 6 ] ]            1:  [6, 2, 3, 11]
    .                              .
```


```
' [1 2 3; 4 5 6; 7 6 0; 2 4 6] RET  s 7  v t  [6 2 3 11]   *
```

Now we compute the matrix @c{A'} A2 and divide.

```
2:  [57, 84, 96]          1:  [-11.64, 14.08, -3.64]
1:  [ [ 70, 72, 39 ]          .
      [ 72, 81, 60 ]
      [ 39, 60, 81 ] ]
    .

    r 7 v t r 7 *             /
```

(The actual computed answer will be slightly inexact due to round-off error.)

Notice that the answers are similar to those for the @c{3×3} 3x3 system solved in the text. That's because the fourth equation that was added to the system is almost identical to the first one multiplied by two. (If it were identical, we would have gotten the exact same answer since the @c{4×3} 4x3 system would be equivalent to the original @c{3×3} 3x3 system.)

Since the first and fourth equations aren't quite equivalent, they can't both be satisfied at once. Let's plug our answers back into the original system of equations to see how well they match.

```
2:  [-11.64, 14.08, -3.64]     1:  [5.6, 2., 3., 11.2]
1:  [ [ 1, 2, 3 ]                  .
      [ 4, 5, 6 ]
      [ 7, 6, 0 ]
      [ 2, 4, 6 ] ]
    .

    r 7                            TAB *
```

This is reasonably close to our original B vector, [6, 2, 3, 11].

List Tutorial Exercise 1

We can use v x to build a vector of integers. This needs to be adjusted to get the range of integers we desire. Mapping `-' across the vector will accomplish this, although it turns out the plain `-' key will work just as well.

```
2:  2                          2:  2
1:  [1, 2, 3, 4, 5, 6, 7, 8, 9]    1:  [-4, -3, -2, -1, 0, 1, 2, 3, 4]
    .                              .

    2  v x 9 RET                   5 V M -   or   5 -
```

Now we use V M ^ to map the exponentiation operator across the vector.

```
1:  [0.0625, 0.125, 0.25, 0.5, 1, 2, 4, 8, 16]
    .
```


```
    V M ^
```

List Tutorial Exercise 2

Given x and y vectors in quick variables 1 and 2 as before, the first job is to form the matrix that describes the problem.

Thus we want a @c{19×2} 19x2 matrix with our x vector as one column and ones as the other column. So, first we build the column of ones, then we combine the two columns to form our A matrix.

```
2:  [1.34, 1.41, 1.49, ... ]    1:  [ [ 1.34, 1 ]
1:  [1, 1, 1, ...]                  [ 1.41, 1 ]
    .                               [ 1.49, 1 ]
                                    ...

    r 1 1 v b 19 RET                M-2 v p v t   s 3
```

Now we compute @c{$A^T y$} trn(A) * y and @c{$A^T A$} trn(A) * A and divide.

```
1:  [33.36554, 13.613]    2:  [33.36554, 13.613]
    .                     1:  [ [ 98.0003, 41.63 ]
                                [  41.63,   19   ] ]
                              .

 v t r 2 *                    r 3 v t r 3 *
```

(Hey, those numbers look familiar!)

```
1:  [0.52141679, -0.425978]
    .

    /
```

Since we were solving equations of the form @c{$m \times x + b \times 1 = y$} m*x + b*1 = y, these numbers should be m and b, respectively. Sure enough, they agree exactly with the result computed using V M and V R!

The moral of this story: V M and V R will probably solve your problem, but there is often an easier way using the higher-level arithmetic functions!

In fact, there is a built-in a F command that does least-squares fits. See section Curve Fitting.

List Tutorial Exercise 3

Move to one end of the list and press C-@ (or C-SPC or whatever) to set the mark, then move to the other end of the list and type M-# g.

```
1:  [2.3, 6, 22, 15.1, 7, 15, 14, 7.5, 2.5]
```


```
    .
```

To make things interesting, let's assume we don't know at a glance how many numbers are in this list. Then we could type:

```
2:  [2.3, 6, 22, ... ]     2:  [2.3, 6, 22, ... ]
1:  [2.3, 6, 22, ... ]     1:  126356422.5
    .                          .

    RET                        V R *


2:  126356422.5            2:  126356422.5     1:  7.94652913734
1:  [2.3, 6, 22, ... ]     1:  9                   .
    .                          .

    TAB                        v l                 I ^
```

(The I ^ command computes the nth root of a number. You could also type & ^ to take the reciprocal of 9 and then raise the number to that power.)

List Tutorial Exercise 4

A number j is a divisor of n if @c{$n \mathbin{\hbox{\code{\%}}} j = 0$} `n % j = 0'. The first step is to get a vector that identifies the divisors.

```
2:  30                 2:  [0, 0, 0, 2, ...]    1:  [1, 1, 1, 0, ...]
1:  [1, 2, 3, 4, ...]  1:  0                        .
    .                      .

 30 RET v x 30 RET   s 1    V M %  0                V M a =  s 2
```

This vector has 1's marking divisors of 30 and 0's marking non-divisors.

The zeroth divisor function is just the total number of divisors. The first divisor function is the sum of the divisors.

```
1:  8      3:  8                  2:  8                    2:  8
           2:  [1, 2, 3, 4, ...]  1:  [1, 2, 3, 0, ...]   1:  72
           1:  [1, 1, 1, 0, ...]      .                        .
               .

   V R +       r 1 r 2                V M *                  V R +
```

Once again, the last two steps just compute a dot product for which a simple * would have worked equally well.

List Tutorial Exercise 5

The obvious first step is to obtain the list of factors with k f. This list will always be in sorted order, so if there are duplicates they will be right next to each other. A suitable method is to compare the list with a copy of itself shifted over by one.

```
1:  [3, 7, 7, 7, 19]   2:  [3, 7, 7, 7, 19]     2:  [3, 7, 7, 7, 19, 0]
    .                  1:  [3, 7, 7, 7, 19, 0]  1:  [0, 3, 7, 7, 7, 19]
                           .                        .

    19551 k f              RET 0 |                  TAB 0 TAB |


1:  [0, 0, 1, 1, 0, 0]   1:  2          1:  0
    .                        .              .

    V M a =                  V R +          0 a =
```

Note that we have to arrange for both vectors to have the same length so that the mapping operation works; no prime factor will ever be zero, so adding zeros on the left and right is safe. From then on the job is pretty straightforward.

Incidentally, Calc provides the @c{\dfn{M\"obius} μ} Moebius mu function which is zero if and only if its argument is square-free. It would be a much more convenient way to do the above test in practice.

List Tutorial Exercise 6

First use v x 6 RET to get a list of integers, then V M v x to get a list of lists of integers!

List Tutorial Exercise 7

Here's one solution. First, compute the triangular list from the previous exercise and type 1 - to subtract one from all the elements.

```
1:  [ [0],
      [0, 1],
      [0, 1, 2],
      ...

    1 -
```

The numbers down the lefthand edge of the list we desire are called the "triangular numbers" (now you know why!). The nth triangular number is the sum of the integers from 1 to n, and can be computed directly by the formula @c{$n (n+1) \over 2$} n * (n+1) / 2.

```
2:  [ [0], [0, 1], ... ]    2:  [ [0], [0, 1], ... ]
1:  [0, 1, 2, 3, 4, 5]      1:  [0, 1, 3, 6, 10, 15]
```


```
    .                               .

    v x 6 RET 1 -                   V M ' $ ($+1)/2 RET
```

Adding this list to the above list of lists produces the desired result:

```
1:  [ [0],
      [1, 2],
      [3, 4, 5],
      [6, 7, 8, 9],
      [10, 11, 12, 13, 14],
      [15, 16, 17, 18, 19, 20] ]
      .

      V M +
```

If we did not know the formula for triangular numbers, we could have computed them using a V U + command. We could also have gotten them the hard way by mapping a reduction across the original triangular list.

```
2:  [ [0], [0, 1], ... ]    2:  [ [0], [0, 1], ... ]
1:  [ [0], [0, 1], ... ]    1:  [0, 1, 3, 6, 10, 15]
    .                           .

    RET                         V M V R +
```

(This means "map a V R + command across the vector," and since each element of the main vector is itself a small vector, V R + computes the sum of its elements.)

List Tutorial Exercise 8

The first step is to build a list of values of x.

```
1:  [1, 2, 3, ..., 21]  1:  [0, 1, 2, ..., 20]  1:  [0, 0.25, 0.5, ..., 5]
    .                       .                       .

    v x 21 RET              1 -                     4 /  s 1
```

Next, we compute the Bessel function values.

```
1:  [0., 0.124, 0.242, ..., -0.328]
    .

    V M ' besJ(1,$) RET
```

(Another way to do this would be 1 TAB V M f j.)

A way to isolate the maximum value is to compute the maximum using V R X, then compare all the Bessel values with that maximum.


```
2:  [0., 0.124, 0.242, ... ]   1:  [0, 0, 0, ... ]    2:  [0, 0, 0, ...
1:  0.5801562                      .                   1:  1
    .                                                      .

    RET V R X                      V M a =                 RET V R +    DEL
```

It's a good idea to verify, as in the last step above, that only one value is equal to the maximum. (After all, a plot of @c{$\sin x$} sin(x) might have many points all equal to the maximum value, 1.)

The vector we have now has a single 1 in the position that indicates the maximum value of x. Now it is a simple matter to convert this back into the corresponding value itself.

```
2:  [0, 0, 0, ... ]          1:  [0, 0., 0., ... ]    1:  1.75
1:  [0, 0.25, 0.5, ... ]         .                        .
    .

    r 1                          V M *                    V R +
```

If a = had produced more than one 1 value, this method would have given the sum of all maximum x values; not very useful! In this case we could have used v m (`calc-mask-vector`) instead. This command deletes all elements of a "data" vector that correspond to zeros in a "mask" vector, leaving us with, in this example, a vector of maximum x values.

The built-in a X command maximizes a function using more efficient methods. Just for illustration, let's use a X to maximize `besJ(1,x)' over this same interval.

```
2:  besJ(1, x)                 1:  [1.84115, 0.581865]
1:  [0 .. 5]                       .
    .

' besJ(1,x), [0..5] RET            a X x RET
```

The output from a X is a vector containing the value of x that maximizes the function, and the function's value at that maximum. As you can see, our simple search got quite close to the right answer.

List Tutorial Exercise 9

Step one is to convert our integer into vector notation.

```
1:  25129925999            3:  25129925999
    .                      2:  10
                           1:  [11, 10, 9, ..., 1, 0]
                               .

    25129925999 RET            10 RET 12 RET v x 12 RET -


1:  25129925999              1:  [0, 2, 25, 251, 2512, ... ]
```


```
2:  [100000000000, ... ]          .
    .

    V M ^   s 1                   V M \
```

(Recall, the \ command computes an integer quotient.)

```
1:  [0, 2, 5, 1, 2, 9, 9, 2, 5, 9, 9, 9]
    .

    10 V M %   s 2
```

Next we must increment this number. This involves adding one to the last digit, plus handling carries. There is a carry to the left out of a digit if that digit is a nine and all the digits to the right of it are nines.

```
1:  [0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1]   1:  [1, 1, 1, 0, 0, 1, ... ]
    .                                          .

    9 V M a =                                  v v


1:  [1, 1, 1, 0, 0, 0, ... ]   1:  [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
    .                              .

    V U *                          v v 1 |
```

Accumulating * across a vector of ones and zeros will preserve only the initial run of ones. These are the carries into all digits except the rightmost digit. Concatenating a one on the right takes care of aligning the carries properly, and also adding one to the rightmost digit.

```
2:  [0, 0, 0, 0, ... ]     1:  [0, 0, 2, 5, 1, 2, 9, 9, 2, 6, 0, 0, 0]
1:  [0, 0, 2, 5, ... ]         .
    .

    0 r 2 |                    V M +  10 V M %
```

Here we have concatenated 0 to the *left* of the original number; this takes care of shifting the carries by one with respect to the digits that generated them.

Finally, we must convert this list back into an integer.

```
3:  [0, 0, 2, 5, ... ]         2:  [0, 0, 2, 5, ... ]
2:  1000000000000              1:  [1000000000000, 100000000000, ... ]
1:  [100000000000, ... ]           .
    .

    10 RET 12 ^  r 1               |
```


```
1:  [0, 0, 20000000000, 5000000000, ... ]    1:  25129926000
    .                                            .

    V M *                                        V R +
```

Another way to do this final step would be to reduce the formula `10 $$ + $' across the vector of digits.

```
1:  [0, 0, 2, 5, ... ]          1:  25129926000
    .                               .

                                    V R ' 10 $$ + $ RET
```

List Tutorial Exercise 10

For the list [a, b, c, d], the result is ((a = b) = c) = d, which will compare a and b to produce a 1 or 0, which is then compared with c to produce another 1 or 0, which is then compared with d. This is not at all what Joe wanted.

Here's a more correct method:

```
1:  [7, 7, 7, 8, 7]       2:  [7, 7, 7, 8, 7]
    .                         1:  7
                                  .

  ' [7,7,7,8,7] RET           RET v r 1 RET


1:  [1, 1, 1, 0, 1]       1:  0
    .                         .

    V M a =                   V R *
```

List Tutorial Exercise 11

The circle of unit radius consists of those points (x,y) for which x^2 + y^2 < 1. We start by generating a vector of x^2 and a vector of y^2.

We can make this go a bit faster by using the v . and t . commands.

```
2:  [2., 2., ..., 2.]          2:  [2., 2., ..., 2.]
1:  [2., 2., ..., 2.]          1:  [1.16, 1.98, ..., 0.81]
    .                              .

 v . t .  2. v b 100 RET RET       V M k r
```


```
2:  [2., 2., ..., 2.]          1:  [0.026, 0.96, ..., 0.036]
1:  [0.026, 0.96, ..., 0.036]  2:  [0.53, 0.81, ..., 0.094]
    .                              .

    1 -  2 V M ^                   TAB  V M k r  1 -  2 V M ^
```

Now we sum the x^2 and y^2 values, compare with 1 to get a vector of 1/0 truth values, then sum the truth values.

```
1:  [0.56, 1.78, ..., 0.13]    1:  [1, 0, ..., 1]    1:  84
    .                              .                     .

    +                              1 V M a <             V R +
```

The ratio 84/100 should approximate the ratio @c{$\pi/4$} pi/4.

```
1:  0.84       1:  3.36       2:  3.36       1:  1.0695
    .              .          1:  3.14159        .

    100 /          4 *            P              /
```

Our estimate, 3.36, is off by about 7%. We could get a better estimate by taking more points (say, 1000), but it's clear that this method is not very efficient!

(Naturally, since this example uses random numbers your own answer will be slightly different from the one shown here!)

If you typed v . and t . before, type them again to return to full-sized display of vectors.

List Tutorial Exercise 12

This problem can be made a lot easier by taking advantage of some symmetries. First of all, after some thought it's clear that the y axis can be ignored altogether. Just pick a random x component for one end of the match, pick a random direction @c{θ} theta, and see if x and @c{$x + \cos \theta$} x + cos(theta) (which is the x coordinate of the other endpoint) cross a line. The lines are at integer coordinates, so this happens when the two numbers surround an integer.

Since the two endpoints are equivalent, we may as well choose the leftmost of the two endpoints as x. Then theta is an angle pointing to the right, in the range -90 to 90 degrees. (We could use radians, but it would feel like cheating to refer to @c{$\pi/2$} pi/2 radians while trying to estimate @c{π} pi!)

In fact, since the field of lines is infinite we can choose the coordinates 0 and 1 for the lines on either side of the leftmost endpoint. The rightmost endpoint will be between 0 and 1 if the match does not cross a line, or between 1 and 2 if it does. So: Pick random x and @c{θ} theta, compute @c{$x + \cos \theta$} x + cos(theta), and count how many of the results are greater than one. Simple!

We can make this go a bit faster by using the v . and t . commands.

```
1:  [0.52, 0.71, ..., 0.72]    2:  [0.52, 0.71, ..., 0.72]
    .                          1:  [78.4, 64.5, ..., -42.9]
```


```
                                    .

v . t . 1. v b 100 RET  V M k r    180. v b 100 RET  V M k r  90 -
```

(The next step may be slow, depending on the speed of your computer.)

```
2:  [0.52, 0.71, ..., 0.72]    1:  [0.72, 1.14, ..., 1.45]
1:  [0.20, 0.43, ..., 0.73]        .
    .

    m d  V M C                     +


1:  [0, 1, ..., 1]       1:  0.64            1:  3.125
    .                        .                   .

    1 V M a >                V R + 100 /         2 TAB /
```

Let's try the third method, too. We'll use random integers up to one million. The k r command with an integer argument picks a random integer.

```
2:  [1000000, 1000000, ..., 1000000]   2:  [78489, 527587, ..., 814975]
1:  [1000000, 1000000, ..., 1000000]   1:  [324014, 358783, ..., 955450]
    .                                      .

    1000000 v b 100 RET RET                V M k r  TAB  V M k r


1:  [1, 1, ..., 25]      1:  [1, 1, ..., 0]     1:  0.56
    .                        .                       .

    V M k g                  1 V M a =               V R + 100 /


1:  10.714        1:  3.273
    .                 .

    6 TAB /           Q
```

For a proof of this property of the GCD function, see section 4.5.2, exercise 10, of Knuth's *Art of Computer Programming*, volume II.

If you typed v . and t . before, type them again to return to full-sized display of vectors.

List Tutorial Exercise 13

First, we put the string on the stack as a vector of ASCII codes.

```
1:  [84, 101, 115, ..., 51]
    .

    "Testing, 1, 2, 3 RET
```

Note that the " key, like $, initiates algebraic entry so there was no need to type an apostrophe. Also, Calc didn't mind that we omitted the closing ". (The same goes for all closing delimiters like) and] at the end of a formula.

We'll show two different approaches here. In the first, we note that if the input vector is [a, b, c, d], then the hash code is 3 (3 (3a + b) + c) + d = 27a + 9b + 3c + d. In other words, it's a sum of descending powers of three times the ASCII codes.

```
2:  [84, 101, 115, ..., 51]    2:  [84, 101, 115, ..., 51]
1:  16                         1:  [15, 14, 13, ..., 0]
    .                              .

    RET v l                        v x 16 RET -


2:  [84, 101, 115, ..., 51]    1:  1960915098    1:  121
1:  [14348907, ..., 1]             .                 .
    .

    3 TAB V M ^                    *                 511 %
```

Once again, * elegantly summarizes most of the computation. But there's an even more elegant approach: Reduce the formula 3 $$ + $ across the vector. Recall that this represents a function of two arguments that computes its first argument times three plus its second argument.

```
1:  [84, 101, 115, ..., 51]    1:  1960915098
    .                              .

    "Testing, 1, 2, 3 RET          V R ' 3$$+$ RET
```

If you did the decimal arithmetic exercise, this will be familiar. Basically, we're turning a base-3 vector of digits into an integer, except that our "digits" are much larger than real digits.

Instead of typing 511 % again to reduce the result, we can be cleverer still and notice that rather than computing a huge integer and taking the modulo at the end, we can take the modulo at each step without affecting the result. While this means there are more arithmetic operations, the numbers we operate on remain small so the operations are faster.

```
1:  [84, 101, 115, ..., 51]    1:  121
```


```
    .                                .

    "Testing, 1, 2, 3 RET            V R ' (3$$+$)%511 RET
```

Why does this work? Think about a two-step computation: 3 (3a + b) + c. Taking a result modulo 511 basically means subtracting off enough 511's to put the result in the desired range. So the result when we take the modulo after every step is,

for some suitable integers m and n. Expanding out by the distributive law yields

The m term in the latter formula is redundant because any contribution it makes could just as easily be made by the n term. So we can take it out to get an equivalent formula with n' = 3m + n,

which is just the formula for taking the modulo only at the end of the calculation. Therefore the two methods are essentially the same.

Later in the tutorial we will encounter modulo forms, which basically automate the idea of reducing every intermediate result modulo some value *M*.

List Tutorial Exercise 14

We want to use H V U to nest a function which adds a random step to an (x,y) coordinate. The function is a bit long, but otherwise the problem is quite straightforward.

```
2:  [0, 0]     1:  [ [    0,       0    ]
1:  50               [  0.4288, -0.1695 ]
    .                [ -0.4787, -0.9027 ]
                     ...

    [0,0] 50       H V U ' <# + [random(2.0)-1, random(2.0)-1]> RET
```

Just as the text recommended, we used `< >' nameless function notation to keep the two `random` calls from being evaluated before nesting even begins.

We now have a vector of [x, y] sub-vectors, which by Calc's rules acts like a matrix. We can transpose this matrix and unpack to get a pair of vectors, x and y, suitable for graphing.

```
2:  [ 0, 0.4288, -0.4787, ... ]
1:  [ 0, -0.1696, -0.9027, ... ]
    .

    v t  v u  g f
```

Incidentally, because the x and y are completely independent in this case, we could have done two separate commands to create our x and y vectors of numbers directly.

To make a random walk of unit steps, we note that `sincos` of a random direction exactly gives us an [x, y] step of unit length; in fact, the new nesting function is even briefer, though we might want to lower the precision a bit for it.


```
2:  [0, 0]      1:  [ [    0,      0    ]
1:  50                 [  0.1318, 0.9912 ]
    .                  [ -0.5965, 0.3061 ]
                       ...

    [0,0] 50   m d  p 6 RET   H V U ' <# + sincos(random(360.0))> RET
```

Another v t v u g f sequence will graph this new random walk.

An interesting twist on these random walk functions would be to use complex numbers instead of 2-vectors to represent points on the plane. In the first example, we'd use something like `random + random*(0,1)', and in the second we could use polar complex numbers with random phase angles. (This exercise was first suggested in this form by Randal Schwartz.)

Types Tutorial Exercise 1

If the number is the square root of @c{π} pi times a rational number, then its square, divided by @c{π} pi, should be a rational number.

```
1:  1.26508260337    1:  0.509433962268   1:  2486645810:4881193627
    .                    .                    .

                         2 ^ P /              c F
```

Technically speaking this is a rational number, but not one that is likely to have arisen in the original problem. More likely, it just happens to be the fraction which most closely represents some irrational number to within 12 digits.

But perhaps our result was not quite exact. Let's reduce the precision slightly and try again:

```
1:  0.509433962268     1:  27:53
    .                      .

    U p 10 RET             c F
```

Aha! It's unlikely that an irrational number would equal a fraction this simple to within ten digits, so our original number was probably sqrt(27 pi / 53).

Notice that we didn't need to re-round the number when we reduced the precision. Remember, arithmetic operations always round their inputs to the current precision before they begin.

Types Tutorial Exercise 2

`inf / inf = nan'. Perhaps `1' is the "obvious" answer. But if `17 inf = inf', then `17 inf / inf = inf / inf = 17', too.

`exp(inf) = inf'. It's tempting to say that the exponential of infinity must be "bigger" than "regular" infinity, but as far as Calc is concerned all infinities are as just as big. In other words, as x goes to infinity, e^x also goes to infinity, but the fact the e^x grows much faster than x is not relevant here.

`exp(-inf) = 0'. Here we have a finite answer even though the input is infinite.

`sqrt(-inf) = (0, 1) inf'. Remember that (0, 1) represents the imaginary number i. Here's a derivation: `sqrt(-inf) = sqrt((-1) * inf) = sqrt(-1) * sqrt(inf)'. The first part is, by definition, i; the second is `inf` because, once again, all infinities are the same size.

`sqrt(uinf) = uinf'. In fact, we do know something about the direction because `sqrt` is defined to return a value in the right half of the complex plane. But Calc has no notation for this, so it settles for the conservative answer `uinf`.

`abs(uinf) = inf'. No matter which direction x points, `abs(x)' always points along the positive real axis.

`ln(0) = -inf'. Here we have an infinite answer to a finite input. As in the 1 / 0 case, Calc will only use infinities here if you have turned on "infinite" mode. Otherwise, it will treat `ln(0)' as an error.

Types Tutorial Exercise 3

We can make `inf - inf' be any real number we like, say, a, just by claiming that we added a to the first infinity but not to the second. This is just as true for complex values of a, so `nan` can stand for a complex number. (And, similarly, `uinf` can stand for an infinity that points in any direction in the complex plane, such as `(0, 1) inf').

In fact, we can multiply the first `inf` by two. Surely `2 inf - inf = inf', but also `2 inf - inf = inf - inf = nan'. So `nan` can even stand for infinity. Obviously it's just as easy to make it stand for minus infinity as for plus infinity.

The moral of this story is that "infinity" is a slippery fish indeed, and Calc tries to handle it by having a very simple model for infinities (only the direction counts, not the "size"); but Calc is careful to write `nan` any time this simple model is unable to tell what the true answer is.

Types Tutorial Exercise 4

```
2:  0@ 47' 26"              1:  0@ 2' 47.411765"
1:  17                          .
    .

    0@ 47' 26" RET 17           /
```

The average song length is two minutes and 47.4 seconds.

```
2:  0@ 2' 47.411765"     1:  0@ 3' 7.411765"    1:  0@ 53' 6.000005"
1:  0@ 0' 20"                .                      .
    .

    20"                      +                      17 *
```

The album would be 53 minutes and 6 seconds long.

Types Tutorial Exercise 5

Let's suppose it's January 14, 1991. The easiest thing to do is to keep trying 13ths of months until Calc reports a Friday. We can do this by manually entering dates, or by using t I:

```
1:  <Wed Feb 13, 1991>    1:  <Wed Mar 13, 1991>   1:  <Sat Apr 13, 1991>
    .                         .                        .

    ' <2/13> RET       DEL    ' <3/13> RET             t I
```

(Calc assumes the current year if you don't say otherwise.)

This is getting tedious--we can keep advancing the date by typing t I over and over again, but let's automate the job by using vector mapping. The t I command actually takes a second "how-many-months" argument, which defaults to one. This argument is exactly what we want to map over:

```
2:  <Sat Apr 13, 1991>     1:  [<Mon May 13, 1991>, <Thu Jun 13, 1991>,
1:  [1, 2, 3, 4, 5, 6]          <Sat Jul 13, 1991>, <Tue Aug 13, 1991>,
    .                           <Fri Sep 13, 1991>, <Sun Oct 13, 1991>]
                                .

    v x 6 RET                   V M t I

1:  242
    .

' <sep 13> - <jan 14> RET
```

And the answer to our original question: 242 days to go.

Types Tutorial Exercise 6

The full rule for leap years is that they occur in every year divisible by four, except that they don't occur in years divisible by 100, except that they *do* in years divisible by 400. We could work out the answer by carefully counting the years divisible by four and the exceptions, but there is a much simpler way that works even if we don't know the leap year rule.

Let's assume the present year is 1991. Years have 365 days, except that leap years (whenever they occur) have 366 days. So let's count the number of days between now and then, and compare that to the number of years times 365. The number of extra days we find must be equal to the number of leap years there were.

```
1:  <Mon Jan 1, 10001>     2:  <Mon Jan 1, 10001>     1:  2925593
    .                      1:  <Tue Jan 1, 1991>          .
                               .

  ' <jan 1 10001> RET          ' <jan 1 1991> RET          -
```


```
3:  2925593        2:  2925593      2:  2925593      1:  1943
2:  10001          1:  8010         1:  2923650          .
1:  1991               .                .
    .

  10001 RET 1991       -                365 *            -
```

There will be 1943 leap years before the year 10001. (Assuming, of course, that the algorithm for computing leap years remains unchanged for that long. See section Date Forms, for some interesting background information in that regard.)

Types Tutorial Exercise 7

The relative errors must be converted to absolute errors so that `+/-' notation may be used.

```
1:  1.               2:  1.
    .                1:  0.2
                         .

    20 RET .05 *         4 RET .05 *
```

Now we simply chug through the formula.

```
1:  19.7392088022    1:  394.78 +/- 19.739    1:  6316.5 +/- 706.21
    .                    .                        .

    2 P 2 ^ *            20 p 1 *                 4 p .2 RET 2 ^ *
```

It turns out the v u command will unpack an error form as well as a vector. This saves us some retyping of numbers.

```
3:  6316.5 +/- 706.21     2:  6316.5 +/- 706.21
2:  6316.5                1:  0.1118
1:  706.21                    .
    .

    RET v u               TAB /
```

Thus the volume is 6316 cubic centimeters, within about 11 percent.

Types Tutorial Exercise 8

The first answer is pretty simple: `1 / (0 .. 10) = (0.1 .. inf)'. Since a number in the interval `(0 .. 10)' can get arbitrarily close to zero, its reciprocal can get arbitrarily large, so the answer is an interval that effectively means, "any number greater than 0.1" but with no upper bound.

The second answer, similarly, is `1 / (-10 .. 0) = (-inf .. -0.1)'.

Calc normally treats division by zero as an error, so that the formula `1 / 0' is left unsimplified. Our third

problem, `1 / [0 .. 10]', also (potentially) divides by zero because zero is now a member of the interval. So Calc leaves this one unevaluated, too.

If you turn on "infinite" mode by pressing m i, you will instead get the answer `[0.1 .. inf]', which includes infinity as a possible value.

The fourth calculation, `1 / (-10 .. 10)', has the same problem. Zero is buried inside the interval, but it's still a possible value. It's not hard to see that the actual result of `1 / (-10 .. 10)' will be either greater than *0.1*, or less than *-0.1*. Thus the interval goes from minus infinity to plus infinity, with a "hole" in it from *-0.1* to *0.1*. Calc doesn't have any way to represent this, so it just reports `[-inf .. inf]' as the answer. It may be disappointing to hear "the answer lies somewhere between minus infinity and plus infinity, inclusive," but that's the best that interval arithmetic can do in this case.

Types Tutorial Exercise 9

```
1:  [-3 .. 3]       2:  [-3 .. 3]     2:  [0 .. 9]
    .               1:  [0 .. 9]      1:  [-9 .. 9]
                        .                 .

    [ 3 n .. 3 ]        RET 2 ^           TAB RET *
```

In the first case the result says, "if a number is between *-3* and 3, its square is between 0 and 9." The second case says, "the product of two numbers each between *-3* and 3 is between *-9* and 9."

An interval form is not a number; it is a symbol that can stand for many different numbers. Two identical-looking interval forms can stand for different numbers.

The same issue arises when you try to square an error form.

Types Tutorial Exercise 10

Testing the first number, we might arbitrarily choose 17 for x.

```
1:  17 mod 811749613   2:  17 mod 811749613   1:  533694123 mod 811749613
    .                      811749612              .
                           .

    17 M 811749613 RET     811749612              ^
```

Since 533694123 is (considerably) different from 1, the number 811749613 must not be prime.

It's awkward to type the number in twice as we did above. There are various ways to avoid this, and algebraic entry is one. In fact, using a vector mapping operation we can perform several tests at once. Let's use this method to test the second number.

```
2:  [17, 42, 100000]              1:  [1 mod 15485863, 1 mod ... ]
1:  15485863                          .
    .
```


```
[17 42 100000] 15485863 RET              V M ' ($$ mod $)^($-1) RET
```

The result is three ones (modulo n), so it's very probable that 15485863 is prime. (In fact, this number is the millionth prime.)

Note that the functions `($$^($-1)) mod $' or `$$^($-1) % $' would have been hopelessly inefficient, since they would have calculated the power using full integer arithmetic.

Calc has a k p command that does primality testing. For small numbers it does an exact test; for large numbers it uses a variant of the Fermat test we used here. You can use k p repeatedly to prove that a large integer is prime with any desired probability.

Types Tutorial Exercise 11

There are several ways to insert a calculated number into an HMS form. One way to convert a number of seconds to an HMS form is simply to multiply the number by an HMS form representing one second:

```
1:  31415926.5359      2:  31415926.5359      1:  8726@ 38' 46.5359"
    .                  1:  0@ 0' 1"               .
                           .

    P 1e7 *                0@ 0' 1"               *


2:  8726@ 38' 46.5359"               1:  6@ 6' 2.5359" mod 24@ 0' 0"
1:  15@ 27' 16" mod 24@ 0' 0"            .
    .

    x time RET                           +
```

It will be just after six in the morning.

The algebraic `hms` function can also be used to build an HMS form:

```
1:  hms(0, 0, 10000000. pi)          1:  8726@ 38' 46.5359"
    .                                    .

  ' hms(0, 0, 1e7 pi) RET                =
```

The = key is necessary to evaluate the symbol `pi' to the actual number 3.14159...

Types Tutorial Exercise 12

As we recall, there are 17 songs of about 2 minutes and 47 seconds each.

```
2:  0@ 2' 47"                        1:  [0@ 3' 7" .. 0@ 3' 47"]
1:  [0@ 0' 20" .. 0@ 1' 0"]              .
    .
```


```
    [ 0@ 20" .. 0@ 1' ]                +


1:  [0@ 52' 59." .. 1@ 4' 19."]
    .

    17 *
```

No matter how long it is, the album will fit nicely on one CD.

Types Tutorial Exercise 13

Type ' 1 yr RET u c s RET. The answer is 31557600 seconds.

Types Tutorial Exercise 14

How long will it take for a signal to get from one end of the computer to the other?

```
1:  m / c          1:  3.3356 ns
    .                  .

 ' 1 m / c RET         u c ns RET
```

(Recall, `c' is a "unit" corresponding to the speed of light.)

```
1:  3.3356 ns      1:  0.81356 ns / ns      1:  0.81356
2:  4.1 ns             .                        .
    .

  ' 4.1 ns RET         /                        u s
```

Thus a signal could take up to 81 percent of a clock cycle just to go from one place to another inside the computer, assuming the signal could actually attain the full speed of light. Pretty tight!

Types Tutorial Exercise 15

The speed limit is 55 miles per hour on most highways. We want to find the ratio of Sam's speed to the US speed limit.

```
1:  55 mph         2:  55 mph           3:  11 hr mph / yd
    .              1:  5 yd / hr            .
                       .

  ' 55 mph RET       ' 5 yd/hr RET          /
```

The u s command cancels out these units to get a plain number. Now we take the logarithm base two to find the final answer, assuming that each successive pill doubles his speed.


```
1:  19360.        2:  19360.        1:  14.24
    .             1:  2                 .
                      .

    u s               2                 B
```

Thus Sam can take up to 14 pills without a worry.

Algebra Tutorial Exercise 1

The result `sqrt(x)^2' is simplified back to x by the Calculator, but `sqrt(x^2)' is not. (Consider what happens if x = -4.) If x is real, this formula could be simplified to `abs(x)', but for general complex arguments even that is not safe. (See section Declarations, for a way to tell Calc that x is known to be real.)

Algebra Tutorial Exercise 2

Suppose our roots are [a, b, c]. We want a polynomial which is zero when x is any of these values. The trivial polynomial x-a is zero when x=a, so the product (x-a)(x-b)(x-c) will do the job. We can use a c x to write this in a more familiar form.

```
1:  34 x - 24 x^3          1:  [1.19023, -1.19023, 0]
    .                          .

    r 2                        a P x RET


1:  [x - 1.19023, x + 1.19023, x]     1:  (x - 1.19023) (x + 1.19023) x
    .                                     .

    V M ' x-$ RET                         V R *


1:  x^3 - 1.41666 x        1:  34 x - 24 x^3
    .                          .

    a c x RET                  24 n *  a x
```

Sure enough, our answer (multiplied by a suitable constant) is the same as the original polynomial.

Algebra Tutorial Exercise 3

```
1:  x sin(pi x)          1:  (sin(pi x) - pi x cos(pi x)) / pi^2
    .                        .

  ' x sin(pi x) RET   m r   a i x RET
```


```
1:  [y, 1]
2:  (sin(pi x) - pi x cos(pi x)) / pi^2
    .

  ' [y,1] RET TAB


1:  [(sin(pi y) - pi y cos(pi y)) / pi^2, (sin(pi) - pi cos(pi)) / pi^2]
    .

    V M $ RET


1:  (sin(pi y) - pi y cos(pi y)) / pi^2 + (pi cos(pi) - sin(pi)) / pi^2
    .

    V R -


1:  (sin(3.14159 y) - 3.14159 y cos(3.14159 y)) / 9.8696 - 0.3183
    .

    =


1:  [0., -0.95493, 0.63662, -1.5915, 1.2732]
    .

    v x 5 RET  TAB  V M $ RET
```

Algebra Tutorial Exercise 4

The hard part is that V R + is no longer sufficient to add up all the contributions from the slices, since the slices have varying coefficients. So first we must come up with a vector of these coefficients. Here's one way:

```
2:  -1                2:  3                    1:  [4, 2, ..., 4]
1:  [1, 2, ..., 9]    1:  [-1, 1, ..., -1]         .
    .                     .

    1 n v x 9 RET         V M ^  3 TAB             -


1:  [4, 2, ..., 4, 1]     1:  [1, 4, 2, ..., 4, 1]
    .                         .
```


```
    1 |                              1 TAB |
```

Now we compute the function values. Note that for this method we need eleven values, including both endpoints of the desired interval.

```
2:  [1, 4, 2, ..., 4, 1]
1:  [1, 1.1, 1.2,  ...  , 1.8, 1.9, 2.]
    .

 11 RET 1 RET .1 RET  C-u v x


2:  [1, 4, 2, ..., 4, 1]
1:  [0., 0.084941, 0.16993, ... ]
    .

    ' sin(x) ln(x) RET   m r  p 5 RET   V M $ RET
```

Once again this calls for V M * V R +; a simple * does the same thing.

```
1:  11.22      1:  1.122      1:  0.374
    .              .              .

    *              .1 *           3 /
```

Wow! That's even better than the result from the Taylor series method.

Rewrites Tutorial Exercise 1

We'll use Big mode to make the formulas more readable.

```
                                                  ____
                                          2 + V 2
1:  (2 + sqrt(2)) / (1 + sqrt(2))     1:  --------
    .                                         ____
                                          1 + V 2

                                          .

  ' (2+sqrt(2)) / (1+sqrt(2)) RET         d B
```

Multiplying by the conjugate helps because (a+b) (a-b) = a^2 - b^2.

```
                ____   ____
1:  (2 + V 2 ) (V 2  - 1)
    .

  a r a/(b+c) := a*(b-c) / (b^2-c^2) RET
```


```
1:  2 + V 2  - 2                1:  V 2
    .                               .

  a r a*(b+c) := a*b + a*c          a s
```

(We could have used a x instead of a rewrite rule for the second step.)

The multiply-by-conjugate rule turns out to be useful in many different circumstances, such as when the denominator involves sines and cosines or the imaginary constant `i`.

Rewrites Tutorial Exercise 2

Here is the rule set:

```
[ fib(n) := fib(n, 1, 1) :: integer(n) :: n >= 1,
  fib(1, x, y) := x,
  fib(n, x, y) := fib(n-1, y, x+y) ]
```

The first rule turns a one-argument `fib` that people like to write into a three-argument `fib` that makes computation easier. The second rule converts back from three-argument form once the computation is done. The third rule does the computation itself. It basically says that if x and y are two consecutive Fibonacci numbers, then y and x+y are the next (overlapping) pair of Fibonacci numbers.

Notice that because the number n was "validated" by the conditions on the first rule, there is no need to put conditions on the other rules because the rule set would never get that far unless the input were valid. That further speeds computation, since no extra conditions need to be checked at every step.

Actually, a user with a nasty sense of humor could enter a bad three-argument `fib` call directly, say, `fib(0, 1, 1)', which would get the rules into an infinite loop. One thing that would help keep this from happening by accident would be to use something like `ZzFib' instead of `fib` as the name of the three-argument function.

Rewrites Tutorial Exercise 3

He got an infinite loop. First, Calc did as expected and rewrote `2 + 3 x' to `f(2, 3, x)'. Then it looked for ways to apply the rule again, and found that `f(2, 3, x)' looks like `a + b x' with `a = 0' and `b = 1', so it rewrote to `f(0, 1, f(2, 3, x))'. It then wrapped another `f(0, 1, ...)' around that, and so on, ad infinitum. Joe should have used M-1 a r to make sure the rule applied only once.

(Actually, even the first step didn't work as he expected. What Calc really gives for M-1 a r in this situation is `f(3 x, 1, 2)', treating 2 as the "variable," and `3 x' as a constant being added to it. While this may seem odd, it's just as valid a solution as the "obvious" one. One way to fix this would be to add the condition `:: variable(x)' to the rule, to make sure the thing that matches `x' is indeed a variable, or to change `x' to `quote(x)' on the lefthand side, so that the rule matches the actual variable `x' rather than letting `x' stand for something else.)

Rewrites Tutorial Exercise 4

Here is a suitable set of rules to solve the first part of the problem:

```
[ seq(n, c) := seq(n/2,  c+1) :: n%2 = 0,
  seq(n, c) := seq(3n+1, c+1) :: n%2 = 1 :: n > 1 ]
```

Given the initial formula `seq(6, 0)', application of these rules produces the following sequence of formulas:

```
seq( 3, 1)
seq(10, 2)
seq( 5, 3)
seq(16, 4)
seq( 8, 5)
seq( 4, 6)
seq( 2, 7)
seq( 1, 8)
```

whereupon neither of the rules match, and rewriting stops.

We can pretty this up a bit with a couple more rules:

```
[ seq(n) := seq(n, 0),
  seq(1, c) := c,
  ... ]
```

Now, given `seq(6)' as the starting configuration, we get 8 as the result.

The change to return a vector is quite simple:

```
[ seq(n) := seq(n, []) :: integer(n) :: n > 0,
  seq(1, v) := v | 1,
  seq(n, v) := seq(n/2,  v | n) :: n%2 = 0,
  seq(n, v) := seq(3n+1, v | n) :: n%2 = 1 ]
```

Given `seq(6)', the result is `[6, 3, 10, 5, 16, 8, 4, 2, 1]'.

Notice that the n > 1 guard is no longer necessary on the last rule since the n = 1 case is now detected by another rule. But a guard has been added to the initial rule to make sure the initial value is suitable before the computation begins.

While still a good idea, this guard is not as vitally important as it was for the `fib` function, since calling, say, `seq(x, [])' will not get into an infinite loop. Calc will not be able to prove the symbol `x' is either even or odd, so none of the rules will apply and the rewrites will stop right away.

Rewrites Tutorial Exercise 5

If x is the sum a + b, then `nterms(*x*)' must be `nterms(*a*)' plus `nterms(*b*)'. If x is not a sum, then `nterms(*x*)' = 1.


```
[ nterms(a + b) := nterms(a) + nterms(b),
  nterms(x)     := 1 ]
```

Here we have taken advantage of the fact that earlier rules always match before later rules; `nterms(x)' will only be tried if we already know that `x' is not a sum.

Rewrites Tutorial Exercise 6

Just put the rule `0^0 := 1' into `EvalRules'. For example, before making this definition we have:

```
2:  [-2, -1, 0, 1, 2]                 1:  [1, 1, 0^0, 1, 1]
1:  0                                     .
    .

    v x 5 RET  3 -  0                     V M ^
```

But then:

```
2:  [-2, -1, 0, 1, 2]                 1:  [1, 1, 1, 1, 1]
1:  0                                     .
    .

    U  ' 0^0:=1 RET s t EvalRules RET     V M ^
```

Perhaps more surprisingly, this rule still works with infinite mode turned on. Calc tries `EvalRules' before any built-in rules for a function. This allows you to override the default behavior of any Calc feature: Even though Calc now wants to evaluate 0^0 to `nan', your rule gets there first and evaluates it to 1 instead.

Just for kicks, try adding the rule `2+3 := 6' to `EvalRules'. What happens? (Be sure to remove this rule afterward, or you might get a nasty surprise when you use Calc to balance your checkbook!)

Rewrites Tutorial Exercise 7

Here is a rule set that will do the job:

```
[ a*(b + c) := a*b + a*c,
  opt(a) O(x^n) + opt(b) O(x^m) := O(x^n) :: n <= m
     :: constant(a) :: constant(b),
  opt(a) O(x^n) + opt(b) x^m := O(x^n) :: n <= m
     :: constant(a) :: constant(b),
  a O(x^n) := O(x^n) :: constant(a),
  x^opt(m) O(x^n) := O(x^(n+m)),
  O(x^n) O(x^m) := O(x^(n+m)) ]
```

If we really want the + and * keys to operate naturally on power series, we should put these rules in `EvalRules'. For testing purposes, it is better to put them in a different variable, say, `O', first.

The first rule just expands products of sums so that the rest of the rules can assume they have an expanded-out

polynomial to work with. Note that this rule does not mention `O' at all, so it will apply to any product-of- it encounters--this rule may surprise you if you put it into `EvalRules'!

In the second rule, the sum of two O's is changed to the smaller O. The optional constant coefficients are there mostly so that `O(x^2) - O(x^3)' and `O(x^3) - O(x^2)' are handled as well as `O(x^2) + O(x^3)'.

The third rule absorbs higher powers of `x' into O's.

The fourth rule says that a constant times a negligible quantity is still negligible. (This rule will also match `O(x^3) / 4', with `a = 1/4'.)

The fifth rule rewrites, for example, `x^2 O(x^3)' to `O(x^5)'. (It is easy to see that if one of these forms is negligible, the other is, too.) Notice the `x^opt(m)' to pick up terms like `x O(x^3)'. Optional powers will match `x' as `x^1' but not 1 as `x^0'. This turns out to be exactly what we want here.

The sixth rule is the corresponding rule for products of two O's.

Another way to solve this problem would be to create a new "data type" that represents truncated power series. We might represent these as function calls `series(coefs, x)' where coefs is a vector of coefficients for x^0, x^1, x^2, and so on. Rules would exist for sums and products of such `series' objects, and as an optional convenience could also know how to combine a `series' object with a normal polynomial. (With this, and with a rule that rewrites `O(x^n)' to the equivalent `series' form, you could still enter power series in exactly the same notation as before.) Operations on such objects would probably be more efficient, although the objects would be a bit harder to read.

Some other symbolic math programs provide a power series data type similar to this. Mathematica, for example, has an object that looks like `PowerSeries[x, x0, coefs, nmin, nmax, den]', where x0 is the point about which the power series is taken (we've been assuming this was always zero), and nmin, nmax, and den allow pseudo-power-series with fractional or negative powers. Also, the `PowerSeries' objects have a special display format that makes them look like `2 x^2 + O(x^4)' when they are printed out. (See section Compositions, for a way to do this in Calc, although for something as involved as this it would probably be better to write the formatting routine in Lisp.)

Programming Tutorial Exercise 1

Just enter the formula `ninteg(sin(t)/t, t, 0, x)', type Z F, and answer the questions. Since this formula contains two variables, the default argument list will be `(t x)'. We want to change this to `(x)' since t is really a dummy variable to be used within `ninteg'.

The exact keystrokes are Z F s Si RET RET C-b C-b DEL DEL RET y. (The C-b C-b DEL DEL are what fix the argument list.)

Programming Tutorial Exercise 2

One way is to move the number to the top of the stack, operate on it, then move it back: C-x (M-TAB n M-TAB M-TAB C-x).

Another way is to negate the top three stack entries, then negate again the top two stack entries: C-x (M-3 n M-2 n C-x).

Finally, it turns out that a negative prefix argument causes a command like n to operate on the specified stack

entry only, which is just what we want: C-x (M-- 3 n C-x).

Just for kicks, let's also do it algebraically: C-x (' -$$$, $$, $ RET C-x).

Programming Tutorial Exercise 3

Each of these functions can be computed using the stack, or using algebraic entry, whichever way you prefer:

Computing @c{$\displaystyle{\sin x \over x}$} sin(x) / x:

Using the stack: C-x (RET S TAB / C-x).

Using algebraic entry: C-x (' sin($)/$ RET C-x).

Computing the logarithm:

Using the stack: C-x (TAB B C-x)

Using algebraic entry: C-x (' log($,$$) RET C-x).

Computing the vector of integers:

Using the stack: C-x (1 RET 1 C-u v x C-x). (Recall that C-u v x takes the vector size, starting value, and increment from the stack.)

Alternatively: C-x (~ v x C-x). (The ~ key pops a number from the stack and uses it as the prefix argument for the next command.)

Using algebraic entry: C-x (' index($) RET C-x).

Programming Tutorial Exercise 4

Here's one way: C-x (RET V R + TAB v l / C-x).

Programming Tutorial Exercise 5

```
2:  1              1:  1.61803398502          2:  1.61803398502
1:  20                 .                      1:  1.61803398875
    .                                             .

   1 RET 20          Z < & 1 + Z >                 I H P
```

This answer is quite accurate.

Programming Tutorial Exercise 6

Here is the matrix:

```
[ [ 0, 1 ]   * [a, b] = [b, a + b]
  [ 1, 1 ] ]
```

Thus `[0, 1; 1, 1]^n * [1, 1]' computes Fibonacci numbers n+1 and n+2. Here's one program that does the

```
C-x ( ' [0, 1; 1, 1] ^ ($-1) * [1, 1] RET v u DEL C-x )
```

This program is quite efficient because Calc knows how to raise a matrix (or other value) to the power n in only @c{$\log_2 n$} log(n,2) steps. For example, this program can compute the 1000th Fibonacci number (a 209-digit integer!) in about 10 steps; even though the Z < ... Z > solution had much simpler steps, it would have required so many steps that it would not have been practical.

Programming Tutorial Exercise 7

The trick here is to compute the harmonic numbers differently, so that the loop counter itself accumulates the sum of reciprocals. We use a separate variable to hold the integer counter.

```
1:  1          2:  1       1:  .
    .          1:  4
                   .

    1 t 1       1 RET 4      Z ( t 2 r 1 1 + s 1 & Z )
```

The body of the loop goes as follows: First save the harmonic sum so far in variable 2. Then delete it from the stack; the for loop itself will take care of remembering it for us. Next, recall the count from variable 1, add one to it, and feed its reciprocal to the for loop to use as the step value. The for loop will increase the "loop counter" by that amount and keep going until the loop counter exceeds 4.

```
2:  31                  3:  31
1:  3.99498713092       2:  3.99498713092
    .                   1:  4.02724519544
                            .

    r 1 r 2                 RET 31 & +
```

Thus we find that the 30th harmonic number is 3.99, and the 31st harmonic number is 4.02.

Programming Tutorial Exercise 8

The first step is to compute the derivative f'(x) and thus the formula @c{$\displaystyle{x - {f(x) \over f'(x)}}$} x - f(x)/f'(x).

(Because this definition is long, it will be repeated in concise form below. You can use M-# m to load it from there. While you are entering a Z ` Z ' body in a macro, Calc simply collects keystrokes without executing them. In the following diagrams we'll pretend Calc actually executed the keystrokes as you typed them, just for purposes of illustration.)

```
2:  sin(cos(x)) - 0.5           3:  4.5
1:  4.5                         2:  sin(cos(x)) - 0.5
    .                           1:  -(sin(x) cos(cos(x)))
                                    .
```


```
' sin(cos(x))-0.5 RET 4.5  m r  C-x ( Z `  TAB RET a d x RET


2:  4.5
1:  x + (sin(cos(x)) - 0.5) / sin(x) cos(cos(x))
    .

    /  ' x RET TAB -   t 1
```

Now, we enter the loop. We'll use a repeat loop with a 20-repetition limit just in case the method fails to converge for some reason. (Normally, the Z / command will stop the loop before all 20 repetitions are done.)

```
1:  4.5         3:  4.5                     2:  4.5
    .           2:  x + (sin(cos(x)) ...    1:  5.24196456928
                1:  4.5                         .
                    .

  20 Z <          RET r 1 TAB                   s l x RET
```

This is the new guess for x. Now we compare it with the old one to see if we've converged.

```
3:  5.24196     2:  5.24196     1:  5.24196     1:  5.26345856348
2:  5.24196     1:  0               .               .
1:  4.5             .
    .

  RET M-TAB         a =             Z /             Z > Z ' C-x )
```

The loop converges in just a few steps to this value. To check the result, we can simply substitute it back into the equation.

```
2:  5.26345856348
1:  0.499999999997
    .

 RET ' sin(cos($)) RET
```

Let's test the new definition again:

```
2:  x^2 - 9           1:  3.
1:  1                     .
    .

  ' x^2-9 RET 1           X
```

Once again, here's the full Newton's Method definition:


```
C-x ( Z `  TAB RET a d x RET  /  ' x RET TAB -  t 1
           20 Z <  RET r 1 TAB  s l x RET
                   RET M-TAB  a =  Z /
              Z >
      Z '
C-x )
```

It turns out that Calc has a built-in command for applying a formula repeatedly until it converges to a number. See section Nesting and Fixed Points, to see how to use it.

Also, of course, a R is a built-in command that uses Newton's method (among others) to look for numerical solutions to any equation. See section Root Finding.

Programming Tutorial Exercise 9

The first step is to adjust z to be greater than 5. A simple "for" loop will do the job here. If z is less than 5, we reduce the problem using @c{$\psi(z) = \psi(z+1) - 1/z$} psi(z) = psi(z+1) - 1/z. We go on to compute @c{$\psi(z+1)$} psi(z+1), and remember to add back a factor of -1/z when we're done. This step is repeated until z > 5.

(Because this definition is long, it will be repeated in concise form below. You can use M-# m to load it from there. While you are entering a Z ` Z ' body in a macro, Calc simply collects keystrokes without executing them. In the following diagrams we'll pretend Calc actually executed the keystrokes as you typed them, just for purposes of illustration.)

```
1:  1.              1:  1.
    .                   .

 1.0 RET       C-x ( Z `  s 1  0 t 2
```

Here, variable 1 holds z and variable 2 holds the adjustment factor. If z < 5, we use a loop to increase it.

(By the way, we started with `1.0' instead of the integer 1 because otherwise the calculation below will try to do exact fractional arithmetic, and will never converge because fractions compare equal only if they are exactly equal, not just equal to within the current precision.)

```
3:  1.      2:  1.       1:  6.
2:  1.      1:  1            .
1:  5           .
    .

  RET 5       a <    Z [  5 Z (  & s + 2  1 s + 1  1 Z ) r 1  Z ]
```

Now we compute the initial part of the sum: @c{$\ln z - {1 \over 2z}$} ln(z) - 1/2z minus the adjustment factor.

```
2:  1.79175946923      2:  1.7084261359      1:  -0.57490719743
1:  0.0833333333333    1:  2.28333333333         .
    .                      .
```


```
    L  r 1 2 * &          -  r 2                -
```

Now we evaluate the series. We'll use another "for" loop counting up the value of 2 n. (Calc does have a summation command, a +, but we'll use loops just to get more practice with them.)

```
3:  -0.5749       3:  -0.5749       4:  -0.5749      2:  -0.5749
2:  2             2:  1:6           3:  1:6          1:  2.3148e-3
1:  40            1:  2             2:  2                .
    .                 .             1:  36.
                                        .

   2 RET 40        Z ( RET k b TAB     RET r 1 TAB ^      * /


3:  -0.5749       3:  -0.5772      2:  -0.5772     1:  -0.577215664892
2:  -0.5749       2:  -0.5772      1:  0               .
1:  2.3148e-3     1:  -0.5749          .
    .                 .

  TAB RET M-TAB       - RET M-TAB      a =     Z /    2  Z )  Z ' C-x )
```

This is the value of @c{$-\gamma$} - gamma, with a slight bit of roundoff error. To get a full 12 digits, let's use a higher precision:

```
2:  -0.577215664892      2:  -0.577215664892
1:  1.                   1:  -0.577215664901532

    1. RET                   p 16 RET X
```

Here's the complete sequence of keystrokes:

```
C-x ( Z `  s 1  0 t 2
           RET 5 a <  Z [  5 Z (  & s + 2  1 s + 1  1 Z ) r 1  Z ]
           L r 1 2 * & - r 2 -
           2 RET 40  Z (  RET k b TAB RET r 1 TAB ^ * /
                          TAB RET M-TAB - RET M-TAB a = Z /
                  2  Z )
      Z '
C-x )
```

Programming Tutorial Exercise 10

Taking the derivative of a term of the form x^n will produce a term like @c{$n x^{n-1}$} n x^(n-1). Taking the derivative of a constant produces zero. From this it is easy to see that the nth derivative of a polynomial, evaluated at x = 0, will equal the coefficient on the x^n term times n!.

(Because this definition is long, it will be repeated in concise form below. You can use M-# m to load it from

there. While you are entering a Z ` Z ' body in a macro, Calc simply collects keystrokes without executing them. In the following diagrams we'll pretend Calc actually executed the keystrokes as you typed them, ju for purposes of illustration.)

```
2:  5 x^4 + (x + 1)^2          3:  5 x^4 + (x + 1)^2
1:  6                          2:  0
    .                          1:  6
                                   .

  ' 5 x^4 + (x+1)^2 RET 6        C-x ( Z `  [ ] t 1  0 TAB
```

Variable 1 will accumulate the vector of coefficients.

```
2:  0              3:  0                  2:  5 x^4 + ...
1:  5 x^4 + ...    2:  5 x^4 + ...        1:  1
    .              1:  1                      .
                       .

   Z ( TAB         RET 0 s l x RET            M-TAB ! /  s | 1
```

Note that s | 1 appends the top-of-stack value to the vector in a variable; it is completely analogous to s + 1. We could have written instead, r 1 TAB | t 1.

```
1:  20 x^3 + 2 x + 2      1:  0         1:  [1, 2, 1, 0, 5, 0, 0]
    .                         .             .

    a d x RET                 1 Z )         DEL r 1  Z ' C-x )
```

To convert back, a simple method is just to map the coefficients against a table of powers of x.

```
2:  [1, 2, 1, 0, 5, 0, 0]    2:  [1, 2, 1, 0, 5, 0, 0]
1:  6                        1:  [0, 1, 2, 3, 4, 5, 6]
    .                            .

    6 RET                        1 + 0 RET 1 C-u v x


2:  [1, 2, 1, 0, 5, 0, 0]    2:  1 + 2 x + x^2 + 5 x^4
1:  [1, x, x^2, x^3, ... ]       .
    .

    ' x RET TAB V M ^            *
```

Once again, here are the whole polynomial to/from vector programs:

```
C-x ( Z `  [ ] t 1  0 TAB
           Z (  TAB RET 0 s l x RET M-TAB ! /  s | 1
                a d x RET
```


```
         1 Z ) r 1
      Z '
C-x )

C-x (  1 + 0 RET 1 C-u v x ' x RET TAB V M ^ *  C-x )
```

Programming Tutorial Exercise 11

First we define a dummy program to go on the z s key. The true z s key is supposed to take two numbers from the stack and return one number, so DEL as a dummy definition will make sure the stack comes out right.

```
2:  4          1:  4                          2:  4
1:  2              .                          1:  2
    .                                             .

  4 RET 2       C-x ( DEL C-x )  Z K s RET       2
```

The last step replaces the 2 that was eaten during the creation of the dummy z s command. Now we move on to the real definition. The recurrence needs to be rewritten slightly, to the form s(n,m) = s(n-1,m-1) - (n-1) s(n-1,m).

(Because this definition is long, it will be repeated in concise form below. You can use M-# m to load it from there.)

```
2:  4        4:  4       3:  4       2:  4
1:  2        3:  2       2:  2       1:  2
    .        2:  4       1:  0           .
             1:  2           .
                 .

  C-x (       M-2 RET        a =         Z [  DEL DEL 1  Z :


4:  4       2:  4                  2:  3      4:  3    4:  3    3:  3
3:  2       1:  2                  1:  2      3:  2    3:  2    2:  2
2:  2           .                      .      2:  3    2:  3    1:  3
1:  0                                         1:  2    1:  1        .
    .                                             .        .

  RET 0   a = Z [  DEL DEL 0  Z :  TAB 1 - TAB   M-2 RET     1 -      z s
```

(Note that the value 3 that our dummy z s produces is not correct; it is merely a placeholder that will do just as well for now.)

```
3:  3             4:  3          3:  3       2:  3      1:  -6
2:  3             3:  3          2:  3       1:  9          .
1:  2             2:  3          1:  3           .
    .             1:  2              .
```


```
                          .

 M-TAB M-TAB     TAB RET M-TAB          z s           *           -


1:  -6                          2:  4          1:  11      2:  11
    .                           1:  2              .       1:  11
                                    .                          .

  Z ] Z ] C-x )   Z K s RET      DEL 4 RET 2       z s      M-RET k s
```

Even though the result that we got during the definition was highly bogus, once the definition is complete the z s command gets the right answers.

Here's the full program once again:

```
C-x (  M-2 RET a =
       Z [  DEL DEL 1
       Z :  RET 0 a =
            Z [  DEL DEL 0
            Z :  TAB 1 - TAB M-2 RET 1 - z s
                 M-TAB M-TAB TAB RET M-TAB z s * -
            Z ]
       Z ]
C-x )
```

You can read this definition using M-# m (`read-kbd-macro`) followed by Z K s, without having to make a dummy definition first, because `read-kbd-macro` doesn't need to execute the definition as it reads it in. For this reason, `M-# m` is often the easiest way to create recursive programs in Calc.

Programming Tutorial Exercise 12

This turns out to be a much easier way to solve the problem. Let's denote Stirling numbers as calls of the function `s'.

First, we store the rewrite rules corresponding to the definition of Stirling numbers in a convenient variable:

```
s e StirlingRules RET
[ s(n,n) := 1  :: n >= 0,
  s(n,0) := 0  :: n > 0,
  s(n,m) := s(n-1,m-1) - (n-1) s(n-1,m) :: n >= m :: m >= 1 ]
C-c C-c
```

Now, it's just a matter of applying the rules:

```
2:  4          1:  s(4, 2)              1:  11
1:  2              .                        .
    .
```

```
4 RET 2       C-x (   ' s($$,$) RET     a r StirlingRules RET  C-x )
```

As in the case of the `fib` rules, it would be useful to put these rules in `EvalRules` and to add a `:: remember' condition to the last rule.

Go to the previous, next section.

Go to the previous, next section.

Introduction

This chapter is the beginning of the Calc reference manual. It covers basic concepts such as the stack, algebraic and numeric entry, undo, numeric prefix arguments, etc.

Basic Commands

To start the Calculator in its standard interface, type M-x calc. By default this creates a pair of small windows, `*Calculator*' and `*Calc Trail*'. The former displays the contents of the Calculator stack and is manipulated exclusively through Calc commands. It is possible (though not usually necessary) to create several Calc Mode buffers each of which has an independent stack, undo list, and mode settings. There is exactly one Calc Trail buffer; it records a list of the results of all calculations that have been done. The Calc Trail buffer uses a variant of Calc Mode, so Calculator commands still work when the trail buffer's window is selected. It is possible to turn the trail window off, but the `*Calc Trail*' buffer itself still exists and is updated silently. See section Trail Commands.

In most installations, the M-# c key sequence is a more convenient way to start the Calculator. Also, M-# M-# and M-# # are synonyms for M-# c unless you last used Calc in its "keypad" mode.

Most Calc commands use one or two keystrokes. Lower- and upper-case letters are distinct. Commands may also be entered in full M-x form; for some commands this is the only form. As a convenience, the x key (`calc-execute-extended-command`) is like M-x except that it enters the initial string `calc-' for you. For example, the following key sequences are equivalent: S, M-x calc-sin RET, x sin RET.

The Calculator exists in many parts. When you type M-# c, the Emacs "auto-load" mechanism will bring in only the first part, which contains the basic arithmetic functions. The other parts will be auto-loaded the first time you use the more advanced commands like trig functions or matrix operations. This is done to improve the response time of the Calculator in the common case when all you need to do is a little arithmetic. If for some reason the Calculator fails to load an extension module automatically, you can force it to load all the extensions by using the M-# L (`calc-load-everything`) command. See section Mode Settings.

If you type M-x calc or M-# c with any numeric prefix argument, the Calculator is loaded if necessary, but it is not actually started. If the argument is positive, the `` `calc-ext' `` extensions are also loaded if necessary. User-written Lisp code that wishes to make use of Calc's arithmetic routines can use `(calc 0)' or `(calc 1)' to auto-load the Calculator.

If you type M-# b, then next time you use M-# c you will get a Calculator that uses the full height of the Emacs screen. When full-screen mode is on, M-# c runs the `full-calc` command instead of `calc`. From the Unix shell you can type `emacs -f full-calc' to start a new Emacs specifically for use as a calculator. When Calc is started from the Emacs command line like this, Calc's normal "quit" commands actually quit Emacs itself.

The M-# o command is like M-# c except that the Calc window is not actually selected. If you are already in the Calc window, M-# o switches you out of it. (The regular Emacs C-x o command would also work for this, but it has a tendency to drop you into the Calc Trail window instead, which M-# o takes care not to do.)

For one quick calculation, you can type M-# q (`quick-calc`) which prompts you for a formula (like `2+3/4'). The result is displayed at the bottom of the Emacs screen without ever creating any special Calculator windows. See section "Quick Calculator" Mode.

Finally, if you are using the X window system you may want to try M-# k (`calc-keypad`) which runs Calc with a "calculator keypad" picture as well as a stack display. Click on the keys with the mouse to operate the calculator. See section "Keypad" Mode.

The q key (`calc-quit`) exits Calc Mode and closes the Calculator's window(s). It does not delete the Calculator buffers. If you type M-x calc again, the Calculator will reappear with the contents of the stack intact. Typing M-# c or M-# M-# again from inside the Calculator buffer is equivalent to executing `calc-quit`; you can think of M-# M-# as toggling the Calculator on and off.

The M-# x command also turns the Calculator off, no matter which user interface (standard, Keypad, or Embedded) is currently active. It also cancels `calc-edit` mode if used from there.

The d SPC key sequence (`calc-refresh`) redraws the contents of the Calculator buffer from memory. Use this if the contents of the buffer have been damaged somehow.

The o key (`calc-realign`) moves the cursor back to its "home" position at the bottom of the Calculator buffer.

The < and > keys are bound to `calc-scroll-left` and `calc-scroll-right`. These are just like the normal horizontal scrolling commands except that they scroll one half-screen at a time by default. (Calc formats its output to fit within the bounds of the window whenever it can.)

The { and } keys are bound to `calc-scroll-down` and `calc-scroll-up`. They scroll up or down by one-half the height of the Calc window.

The M-# 0 command (`calc-reset`; that's M-# followed by a zero) resets the Calculator to its default state. This clears the stack, resets all the modes, clears the caches (see section Caches), and so on. (It does *not* erase the values of any variables.) With a numeric prefix argument, M-# 0 preserves the contents of the stack but resets everything else.

The M-x calc-version command displays the current version number of Calc and the name of the person who installed it on your system. (This information is also present in the `*Calc Trail*' buffer, and in the output of the h h command.)

Help Commands

The ? key (`calc-help`) displays a series of brief help messages. Some keys (such as b and d) are prefix keys, like Emacs' ESC and C-x prefixes. You can type ? after a prefix to see a list of commands beginning with that prefix. (If the message includes `[MORE]', press ? again to see additional commands

for that prefix.)

The h h (`calc-full-help`) command displays all the ? responses at once. When printed, this makes a nice, compact (three pages) summary of Calc keystrokes.

In general, the h key prefix introduces various commands that provide help within Calc. Many of the h key functions are Calc-specific analogues to the C-h functions for Emacs help.

The h i (`calc-info`) command runs the Emacs Info system to read this manual on-line. This is basically the same as typing C-h i (the regular way to run the Info system), then, if Info is not already in the Calc manual, selecting the beginning of the manual. The M-# i command is another way to read the Calc manual; it is different from h i in that it works any time, not just inside Calc. The plain i key is also equivalent to h i, though this key is obsolete and may be replaced with a different command in a future version of Calc.

The h t (`calc-tutorial`) command runs the Info system on the Tutorial section of the Calc manual. It is like h i, except that it selects the starting node of the tutorial rather than the beginning of the whole manual. (It actually selects the node "Interactive Tutorial" which tells a few things about using the Info system before going on to the actual tutorial.) The M-# t key is equivalent to h t (but it works at all times).

The h s (`calc-info-summary`) command runs the Info system on the Summary node of the Calc manual. See section Calc Summary. The M-# s key is equivalent to h s.

The h k (`calc-describe-key`) command looks up a key sequence in the Calc manual. For example, h k H a S looks up the documentation on the H a S (`calc-solve-for`) command. This works by looking up the textual description of the key(s) in the Key Index of the manual, then jumping to the node indicated by the index.

Most Calc commands do not have traditional Emacs documentation strings, since the h k command is both more convenient and more instructive. This means the regular Emacs C-h k (`describe-key`) command will not be useful for Calc keystrokes.

The h c (`calc-describe-key-briefly`) command reads a key sequence and displays a brief one-line description of it at the bottom of the screen. It looks for the key sequence in the Summary node of the Calc manual; if it doesn't find the sequence there, it acts just like its regular Emacs counterpart C-h c (`describe-key-briefly`). For example, h c H a S gives the description:

```
H a S runs calc-solve-for:  a `H a S' v  => fsolve(a,v)  (?=notes)
```

which means the command H a S or H M-x calc-solve-for takes a value a from the stack, prompts for a value v, then applies the algebraic function `fsolve` to these values. The `?=notes' message means you can now type ? to see additional notes from the summary that apply to this command.

The h f (`calc-describe-function`) command looks up an algebraic function or a command name in the Calc manual. The prompt initially contains `calcFunc-'; follow this with an algebraic function name to look up that function in the Function Index. Or, backspace and enter a command name beginning with `calc-' to look it up in the Command Index. This command will also look up operator symbols that can appear in algebraic formulas, like `%' and `=>'.

The h v (`calc-describe-variable`) command looks up a variable in the Calc manual. The pr initially contains the `var-' prefix; just add a variable name like `pi` or `PlotRejects`.

The h b (`calc-describe-bindings`) command is just like C-h b, except that only local (Calc-related) key bindings are listed.

The h n or h C-n (`calc-view-news`) command displays the "news" or change history of Calc. This is kept in the file `` `README' ``, which Calc looks for in the same directory as the Calc source files.

The h C-c, h C-d, and h C-w keys display copying, distribution, and warranty information about Calc. These work by pulling up the appropriate parts of the "Copying" or "Reporting Bugs" sections of the manual.

Stack Basics

Calc uses RPN notation. If you are not familar with RPN, see section RPN Calculations and the Stack.

To add the numbers 1 and 2 in Calc you would type the keys: 1 RET 2 +. (RET corresponds to the ENTER key on most calculators.) The first three keystrokes "push" the numbers 1 and 2 onto the stack. The + key always "pops" the top two numbers from the stack, adds them, and pushes the result (3) back onto the stack. This number is ready for further calculations: 5 - pushes 5 onto the stack, then pops the 3 and 5, subtracts them, and pushes the result (-*2*).

Note that the "top" of the stack actually appears at the *bottom* of the buffer. A line containing a single `.' character signifies the end of the buffer; Calculator commands operate on the number(s) directly above this line. The d t (`calc-truncate-stack`) command allows you to move the `.' marker up and down in the stack; see section Truncating the Stack.

Stack elements are numbered consecutively, with number 1 being the top of the stack. These line numbers are ordinarily displayed on the lefthand side of the window. The d l (`calc-line-numbering`) command controls whether these numbers appear. (Line numbers may be turned off since they slow the Calculator down a bit and also clutter the display.)

The unshifted letter o (`calc-realign`) command repositions the cursor to its top-of-stack "home" position. It also undoes any horizontal scrolling in the window. If you give it a numeric prefix argument, it instead moves the cursor to the specified stack element.

The RET (or equivalent SPC) key is only required to separate two consecutive numbers. (After all, if you typed 1 2 by themselves the Calculator would enter the number 12.) If you press RET or SPC *not* right after typing a number, the key duplicates the number on the top of the stack. RET * is thus a handy way to square a number.

The DEL key pops and throws away the top number on the stack. The TAB key swaps the top two objects on the stack. See section Stack and Trail Commands, for descriptions of these and other stack-related commands.

Numeric Entry

Pressing a digit or other numeric key begins numeric entry using the minibuffer. The number is pushed on the stack when you press the RET or SPC keys. If you press any other non-numeric key, the number is pushed onto the stack and the appropriate operation is performed. If you press a numeric key which is not valid, the key is ignored.

There are three different concepts corresponding to the word "minus," typified by a-b (subtraction), -x (change-sign), and -5 (negative number). Calc uses three different keys for these operations, respectively: -, n, and _ (the underscore). The - key subtracts the two numbers on the top of the stack. The n key changes the sign of the number on the top of the stack or the number currently being entered. The _ key begins entry of a negative number or changes the sign of the number currently being entered. The following sequences all enter the number -5 onto the stack: 0 RET 5 -, 5 n RET, 5 RET n, _ 5 RET, 5 _ RET.

Some other keys are active during numeric entry, such as # for non-decimal numbers, : for fractions, and @ for HMS forms. These notations are described later in this manual with the corresponding data types. See section Data Types.

During numeric entry, the only editing key available is DEL.

Algebraic Entry

Calculations can also be entered in algebraic form. This is accomplished by typing the apostrophe key, ', followed by the expression in standard format: ' 2+3*4 RET computes 2+(3*4) = 14 and pushes that on the stack. If you wish you can ignore the RPN aspect of Calc altogether and simply enter algebraic expressions in this way. You may want to use DEL every so often to clear previous results off the stack.

You can press the apostrophe key during normal numeric entry to switch the half-entered number into algebraic entry mode. One reason to do this would be to use the full Emacs cursor motion and editing keys, which are available during algebraic entry but not during numeric entry.

In the same vein, during either numeric or algebraic entry you can press ` (backquote) to switch to `calc-edit` mode, where you complete your half-finished entry in a separate buffer. See section Editing Stack Entries.

If you prefer algebraic entry, you can use the command m a (`calc-algebraic-mode`) to set Algebraic mode. In this mode, digits and other keys that would normally start numeric entry instead start full algebraic entry; as long as your formula begins with a digit you can omit the apostrophe. Open parentheses and square brackets also begin algebraic entry. You can still do RPN calculations in this mode, but you will have to press RET to terminate every number: 2 RET 3 RET * 4 RET + would accomplish the same thing as 2*3+4 RET.

If you give a numeric prefix argument like C-u to the m a command, it enables Incomplete Algebraic mode; this is like regular Algebraic mode except that it applies to the (and [keys only. Numeric keys still begin a numeric entry in this mode.

The m t (`calc-total-algebraic-mode`) gives you an even stronger algebraic-entry mode, in which *all* regular letter and punctuation keys begin algebraic entry. Use this if you prefer typing sqrt( ) instead of Q, factor( ) instead of a f, and so on. To type regular Calc commands when you are in "total" algebraic mode, hold down the META key. Thus M-q is the command to quit Calc, M-p sets the precision, and M-m t (or M-m M-t, if you prefer) turns total algebraic mode back off again. Meta keys also terminate algebraic entry, so that 2+3 M-S is equivalent to 2+3 RET M-S. The symbol `Alg*' will appear in the mode line whenever you are in this mode.

Pressing ' (the apostrophe) a second time re-enters the previous algebraic formula. You can then use the normal Emacs editing keys to modify this formula to your liking before pressing RET.

Within a formula entered from the keyboard, the symbol $ represents the number on the top of the stack. If an entered formula contains any $ characters, the Calculator replaces the top of stack with that formula rather than simply pushing the formula onto the stack. Thus, ' 1+2 RET pushes 3 on the stack, and $*2 RET replaces it with 6. Note that the $ key always initiates algebraic entry; the ' is unnecessary if $ is the first character in the new formula.

Higher stack elements can be accessed from an entered formula with the symbols $$, $$$, and so on. The number of stack elements removed (to be replaced by the entered values) equals the number of dollar signs in the longest such symbol in the formula. For example, `$$+$$$' adds the second and third stack elements, replacing the top three elements with the answer. (All information about the top stack element is thus lost since no single `$' appears in this formula.)

A slightly different way to refer to stack elements is with a dollar sign followed by a number: `$1', `$2', and so on are much like `$', `$$', etc., except that stack entries referred to numerically are not replaced by the algebraic entry. That is, while `$+1' replaces 5 on the stack with 6, `$1+1' leaves the 5 on the stack and pushes an additional 6.

If a sequence of formulas are entered separated by commas, each formula is pushed onto the stack in turn. For example, `1,2,3' pushes those three numbers onto the stack (leaving the 3 at the top), and `$+1,$-1' replaces a 5 on the stack with 4 followed by 6. Also, `$,$$' exchanges the top two elements of the stack, just like the TAB key.

You can finish an algebraic entry with M-= or M-RET instead of RET. This uses = to evaluate the variables in each formula that goes onto the stack. (Thus ' pi RET pushes the variable `pi', but ' pi M-RET pushes 3.1415.)

If you finish your algebraic entry by pressing LFD (or C-j) instead of RET, Calc disables the default simplifications (as if by m O; see section Simplification Modes) while the entry is being pushed on the stack. Thus ' 1+2 RET pushes 3 on the stack, but ' 1+2 LFD pushes the formula 1+2; you might then press = when it is time to evaluate this formula.

"Quick Calculator" Mode

There is another way to invoke the Calculator if all you need to do is make one or two quick calculations. Type M-# q (or M-x quick-calc), then type any formula as an algebraic entry. The Calculator will compute the result and display it in the echo area, without ever actually putting up a Calc window.

You can use the $ character in a Quick Calculator formula to refer to the previous Quick Calculator result. Older results are not retained; the Quick Calculator has no effect on the full Calculator's stack or trail. If you compute a result and then forget what it was, just run `M-# q` again and enter `$' as the formula.

If this is the first time you have used the Calculator in this Emacs session, the M-# q command will create the `*Calculator*` buffer and perform all the usual initializations; it simply will refrain from putting that buffer up in a new window. The Quick Calculator refers to the `*Calculator*` buffer for all mode settings. Thus, for example, to set the precision that the Quick Calculator uses, simply run the full Calculator momentarily and use the regular p command.

If you use `M-# q` from inside the Calculator buffer, the effect is the same as pressing the apostrophe key (algebraic entry).

The result of a Quick calculation is placed in the Emacs "kill ring" as well as being displayed. A subsequent C-y command will yank the result into the editing buffer. You can also use this to yank the result into the next M-# q input line as a more explicit alternative to $ notation, or to yank the result into the Calculator stack after typing M-# c.

If you finish your formula by typing LFD (or C-j) instead of RET, the result is inserted immediately into the current buffer rather than going into the kill ring.

Quick Calculator results are actually evaluated as if by the = key (which replaces variable names by their stored values, if any). If the formula you enter is an assignment to a variable using the `:=' operator, say, `foo := 2 + 3' or `foo := foo + 1', then the result of the evaluation is stored in that Calc variable. See section Storing and Recalling.

If the result is an integer and the current display radix is decimal, the number will also be displayed in hex and octal formats. If the integer is in the range from 1 to 126, it will also be displayed as an ASCII character.

For example, the quoted character `"x"' produces the vector result `[120]' (because 120 is the ASCII code of the lower-case `x'; see section Strings). Since this is a vector, not an integer, it is displayed only according to the current mode settings. But running Quick Calc again and entering `120' will produce the result `120 (16#78, 8#170, x)' which shows the number in its decimal, hexadecimal, octal, and ASCII forms.

Please note that the Quick Calculator is not any faster at loading or computing the answer than the full Calculator; the name "quick" merely refers to the fact that it's much less hassle to use for small calculations.

Numeric Prefix Arguments

Many Calculator commands use numeric prefix arguments. Some, such as d s (`calc-sci-notation`), set a parameter to the value of the prefix argument or use a default if you don't use a prefix. Others (like d f (`calc-fix-notation`)) require an argument and prompt for a number if you don't give one as a prefix.

As a rule, stack-manipulation commands accept a numeric prefix argument which is interpreted as an index into the stack. A positive argument operates on the top n stack entries; a negative argument operates on the nth stack entry in isolation; and a zero argument operates on the entire stack.

Most commands that perform computations (such as the arithmetic and scientific functions) accept a numeric prefix argument that allows the operation to be applied across many stack elements. For unary operations (that is, functions of one argument like absolute value or complex conjugate), a positive prefix argument applies that function to the top n stack entries simultaneously, and a negative argument applies it to the nth stack entry only. For binary operations (functions of two arguments like addition, GCD, and vector concatenation), a positive prefix argument "reduces" the function across the top n stack elements (for example, C-u 5 + sums the top 5 stack entries; see section Reducing and Mapping Vectors), and a negative argument maps the next-to-top n stack elements with the top stack element as a second argument (for example, 7 c-u -5 + adds 7 to the top 5 stack elements). This feature is not available for operations which use the numeric prefix argument for some other purpose.

Numeric prefixes are specified the same way as always in Emacs: Press a sequence of META-digits, or press ESC followed by digits, or press C-u followed by digits. Some commands treat plain C-u (without any actual digits) specially.

You can type ~ (`calc-num-prefix`) to pop an integer from the top of the stack and enter it as the numeric prefix for the next command. For example, C-u 16 p sets the precision to 16 digits; an alternate (silly) way to do this would be 2 RET 4 ^ ~ p, i.e., compute 2 to the fourth power and set the precision to that value.

Conversely, if you have typed a numeric prefix argument the ~ key pushes it onto the stack in the form of an integer.

Undoing Mistakes

The shift-U key (`calc-undo`) undoes the most recent operation. If that operation added or dropped objects from the stack, those objects are removed or restored. If it was a "store" operation, you are queried whether or not to restore the variable to its original value. The U key may be pressed any number of times to undo successively farther back in time; with a numeric prefix argument it undoes a specified number of operations. The undo history is cleared only by the q (`calc-quit`) command. (Recall that M-# c is synonymous with `calc-quit` while inside the Calculator; this also clears the undo history.)

Currently the mode-setting commands (like `calc-precision`) are not undoable. You can undo past a point where you changed a mode, but you will need to reset the mode yourself.

The shift-D key (`calc-redo`) redoes an operation that was mistakenly undone. Pressing U with a negative prefix argument is equivalent to executing `calc-redo`. You can redo any number of times, up to the number of recent consecutive undo commands. Redo information is cleared whenever you give any command that adds new undo information, i.e., if you undo, then enter a number on the stack or make any other change, then it will be too late to redo.

The M-RET key (`calc-last-args`) is like undo in that it restores the arguments of the most recent command onto the stack; however, it does not remove the result of that command. Given a numeric

prefix argument, this command applies to the nth most recent command which removed items from t stack; it pushes those items back onto the stack.

The K (`calc-keep-args`) command provides a related function to M-RET. See section Stack and Trail Commands.

It is also possible to recall previous results or inputs using the trail. See section Trail Commands.

The standard Emacs C-_ undo key is recognized as a synonym for U.

Error Messages

Many situations that would produce an error message in other calculators simply create unsimplified formulas in the Emacs Calculator. For example, 1 RET 0 / pushes the formula 1 / 0; 0 L pushes the formula `ln(0)'. Floating-point overflow and underflow are also reasons for this to happen.

When a function call must be left in symbolic form, Calc usually produces a message explaining why. Messages that are probably surprising or indicative of user errors are displayed automatically. Other messages are simply kept in Calc's memory and are displayed only if you type w (`calc-why`). You can also press w if the same computation results in several messages. (The first message will end with `[w=more]' in this case.)

The d w (`calc-auto-why`) command controls when error messages are displayed automatically. (Calc effectively presses w for you after your computation finishes.) By default, this occurs only for "important" messages. The other possible modes are to report *all* messages automatically, or to report none automatically (so that you must always press w yourself to see the messages).

Multiple Calculators

It is possible to have any number of Calc Mode buffers at once. Usually this is done by executing M-x another-calc, which is similar to M-# c except that if a `*Calculator*' buffer already exists, a new, independent one with a name of the form `*Calculator*<n>' is created. You can also use the command `calc-mode` to put any buffer into Calculator mode, but this would ordinarily never be done.

The q (`calc-quit`) command does not destroy a Calculator buffer; it only closes its window. Use M-x kill-buffer to destroy a Calculator buffer.

Each Calculator buffer keeps its own stack, undo list, and mode settings such as precision, angular mode, and display formats. In Emacs terms, variables such as `calc-stack` are buffer-local variables. The global default values of these variables are used only when a new Calculator buffer is created. The `calc-quit` command saves the stack and mode settings of the buffer being quit as the new defaults.

There is only one trail buffer, `*Calc Trail*', used by all Calculator buffers.

Troubleshooting Commands

This section describes commands you can use in case a computation incorrectly fails or gives the wrong answer.

See section Reporting Bugs, if you find a problem that appears to be due to a bug or deficiency in Calc.

Autoloading Problems

The Calc program is split into many component files; components are loaded automatically as you use various commands that require them. Occasionally Calc may lose track of when a certain component is necessary; typically this means you will type a command and it won't work because some function you've never heard of was undefined.

If this happens, the easiest workaround is to type M-# L (`calc-load-everything`) to force all the parts of Calc to be loaded right away. This will cause Emacs to take up a lot more memory than it would otherwise, but it's guaranteed to fix the problem.

If you seem to run into this problem no matter what you do, or if even the M-# L command crashes, Calc may have been improperly installed. See section Installation, for details of the installation process.

Recursion Depth

Calc uses recursion in many of its calculations. Emacs Lisp keeps a variable `max-lisp-eval-depth` which limits the amount of recursion possible in an attempt to recover from program bugs. If a calculation ever halts incorrectly with the message "Computation got stuck or ran too long," use the M command (`calc-more-recursion-depth`) to increase this limit. (Of course, this will not help if the calculation really did get stuck due to some problem inside Calc.)

The limit is always increased (multiplied) by a factor of two. There is also an I M (`calc-less-recursion-depth`) command which decreases this limit by a factor of two, down to a minimum value of 200. The default value is 1000.

These commands also double or halve `max-specpdl-size`, another internal Lisp recursion limit. The minimum value for this limit is 600.

Caches

Calc saves certain values after they have been computed once. For example, the P (`calc-pi`) command initially "knows" the constant @c{π} pi to about 20 decimal places; if the current precision is greater than this, it will recompute @c{π} pi using a series approximation. This value will not need to be recomputed ever again unless you raise the precision still further. Many operations such as logarithms and sines make use of similarly cached values such as pi/4 and @c{$\ln 2$} ln(2). The visible effect of caching is that high-precision computations may seem to do extra work the first time. Other things cached include powers of two (for the binary arithmetic functions), matrix inverses and determinants, symbolic integrals, and data points computed by the graphing commands.

If you suspect a Calculator cache has become corrupt, you can use the `calc-flush-caches` command to reset all caches to the empty state. (This should only be necessary in the event of bugs in the Calculator.) The M-# 0 (with the zero key) command also resets caches along with all other aspects of the Calculator's state.

Debugging Calc

A few commands exist to help in the debugging of Calc commands. See section Programming, to see the various ways that you can write your own Calc commands.

The Z T (`calc-timing`) command turns on and off a mode in which the timing of slow commands is reported in the Trail. Any Calc command that takes two seconds or longer writes a line to the Trail showing how many seconds it took. This value is accurate only to within one second.

All steps of executing a command are included; in particular, time taken to format the result for display in the stack and trail is counted. Some prompts also count time taken waiting for them to be answered, while others do not; this depends on the exact implementation of the command. For best results, if you are timing a sequence that includes prompts or multiple commands, define a keyboard macro to run the whole sequence at once. Calc's X command (see section Programming with Keyboard Macros) will then report the time taken to execute the whole macro.

Another advantage of the X command is that while it is executing, the stack and trail are not updated from step to step. So if you expect the output of your test sequence to leave a result that may take a long time to format and you don't wish to count this formatting time, end your sequence with a DEL keystroke to clear the result from the stack. When you run the sequence with X, Calc will never bother to format the large result.

Another thing Z T does is to increase the Emacs variable `gc-cons-threshold` to a much higher value (two million; the usual default in Calc is 250,000) for the duration of each command. This generally prevents garbage collection during the timing of the command, though it may cause your Emacs process to grow abnormally large. (Garbage collection time is a major unpredictable factor in the timing of Emacs operations.)

Another command that is useful when debugging your own Lisp extensions to Calc is M-x calc-pass-errors, which disables the error handler that changes the "`max-lisp-eval-depth` exceeded" message to the much more friendly "Computation got stuck or ran too long." This handler interferes with the Emacs Lisp debugger's `debug-on-error` mode. Errors are reported in the handler itself rather than at the true location of the error. After you have executed `calc-pass-errors`, Lisp errors will be reported correctly but the user-friendly message will be lost.

Go to the previous, next section.

Go to the previous, next section.

Data Types

This chapter discusses the various types of objects that can be placed on the Calculator stack, how they are displayed, and how they are entered. (See section Data Type Formats, for information on how these data types are represented as underlying Lisp objects.)

Integers, fractions, and floats are various ways of describing real numbers. HMS forms also for many purposes act as real numbers. These types can be combined to form complex numbers, modulo forms, error forms, or interval forms. (But these last four types cannot be combined arbitrarily: error forms may not contain modulo forms, for example.) Finally, all these types of numbers may be combined into vectors, matrices, or algebraic formulas.

Integers

The Calculator stores integers to arbitrary precision. Addition, subtraction, and multiplication of integers always yields an exact integer result. (If the result of a division or exponentiation of integers is not an integer, it is expressed in fractional or floating-point form according to the current Fraction Mode. See section Fraction Mode.)

A decimal integer is represented as an optional sign followed by a sequence of digits. Grouping (see section Grouping Digits) can be used to insert a comma at every third digit for display purposes, but you must not type commas during the entry of numbers.

A non-decimal integer is represented as an optional sign, a radix between 2 and 36, a `#' symbol, and one or more digits. For radix 11 and above, the letters A through Z (upper- or lower-case) count as digits and do not terminate numeric entry mode. See section Radix Modes, for how to set the default radix for display of integers. Numbers of any radix may be entered at any time. If you press # at the beginning of a number, the current display radix is used.

Fractions

A fraction is a ratio of two integers. Fractions are traditionally written "2/3" but Calc uses the notation `2:3'. (The / key performs RPN division; the following two sequences push the number `2:3' on the stack: 2 : 3 RET, or 2 RET 3 / assuming Fraction Mode has been enabled.) When the Calculator produces a fractional result it always reduces it to simplest form, which may in fact be an integer.

Fractions may also be entered in a three-part form, where `2:3:4' represents two-and-three-quarters. See section Fraction Formats, for fraction display formats.

Non-decimal fractions are entered and displayed as `radix#num:denom' (or in the analogous three-part form). The numerator and denominator always use the same radix.

Floats

A floating-point number or float is a number stored in scientific notation. The number of significant digits in the fractional part is governed by the current floating precision (see section Precision). The range of acceptable values is from @c{$10^{-3999999}$} 10^-3999999 (inclusive) to @c{$10^{4000000}$} 10^4000000 (exclusive), plus the corresponding negative values and zero.

Calculations that would exceed the allowable range of values (such as `exp(exp(20))') are left in symbolic form by Calc. The messages "floating-point overflow" or "floating-point underflow" indicate that during the calculation a number would have been produced that was too large or too close to zero, respectively, to be represented by Calc. This does not necessarily mean the final result would have overflowed, just that an overflow occurred while computing the result. (In fact, it could report an underflow even though the final result would have overflowed!)

If a rational number and a float are mixed in a calculation, the result will in general be expressed as a float. Commands that require an integer value (such as k g [gcd]) will also accept integer-valued floats, i.e., floating-point numbers with nothing after the decimal point.

Floats are identified by the presence of a decimal point and/or an exponent. In general a float consists of an optional sign, digits including an optional decimal point, and an optional exponent consisting of an `e', an optional sign, and up to seven exponent digits. For example, `23.5e-2' is 23.5 times ten to the minus-second power, or 0.235.

Floating-point numbers are normally displayed in decimal notation with all significant figures shown. Exceedingly large or small numbers are displayed in scientific notation. Various other display options are available. See section Float Formats.

Floating-point numbers are stored in decimal, not binary. The result of each operation is rounded to the nearest value representable in the number of significant digits specified by the current precision, rounding away from zero in the case of a tie. Thus (in the default display mode) what you see is exactly what you get. Some operations such as square roots and transcendental functions are performed with several digits of extra precision and then rounded down, in an effort to make the final result accurate to the full requested precision. However, accuracy is not rigorously guaranteed. If you suspect the validity of a result, try doing the same calculation in a higher precision. The Calculator's arithmetic is not intended to be IEEE-conformant in any way.

While floats are always *stored* in decimal, they can be entered and displayed in any radix just like integers and fractions. The notation `radix#ddd.ddd' is a floating-point number whose digits are in the specified radix. Note that the `.' is more aptly referred to as a "radix point" than as a decimal point in this case. The number `8#123.4567' is defined as `8#1234567 * 8^-4'. If the radix is 14 or less, you can use `e' notation to write a non-decimal number in scientific notation. The exponent is written in decimal, and is considered to be a power of the radix: `8#1234567e-4'. If the radix is 15 or above, the letter `e' is a digit, so scientific notation must be written out, e.g., `16#123.4567*16^2'. The first two exercises of the Modes Tutorial explore some of the properties of non-decimal floats.

Complex Numbers

There are two supported formats for complex numbers: rectangular and polar. The default format is rectangular, displayed in the form `(real,imag)' where real is the real part and imag is the imaginary part, each of which may be any real number. Rectangular complex numbers can also be displayed in `a+bi' notation; see section Complex Formats.

Polar complex numbers are displayed in the form `(r;@c{θ} theta)' where r is the nonnegative magnitude and @c{θ} theta is the argument or phase angle. The range of @c{θ} theta depends on the current angular mode (see section Angular Modes); it is generally between *-180* and *+180* degrees or the equivalent range in radians.

Complex numbers are entered in stages using incomplete objects. See section Incomplete Objects.

Operations on rectangular complex numbers yield rectangular complex results, and similarly for polar complex numbers. Where the two types are mixed, or where new complex numbers arise (as for the square root of a negative real), the current Polar Mode is used to determine the type. See section Polar Mode.

A complex result in which the imaginary part is zero (or the phase angle is 0 or 180 degrees or @c{π} pi radians) is automatically converted to a real number.

Infinities

The word `inf` represents the mathematical concept of infinity. Calc actually has three slightly different infinity-like values: `inf`, `uinf`, and `nan`. These are just regular variable names (see section Variables); you should avoid using these names for your own variables because Calc gives them special treatment. Infinities, like all variable names, are normally entered using algebraic entry.

Mathematically speaking, it is not rigorously correct to treat "infinity" as if it were a number, but mathematicians often do so informally. When they say that `1 / inf = 0', what they really mean is that 1 / x, as x becomes larger and larger, becomes arbitrarily close to zero. So you can imagine that if x got "all the way to infinity," then 1 / x would go all the way to zero. Similarly, when they say that `exp(inf) = inf', they mean that @c{e^x} exp(x) grows without bound as x grows. The symbol `-inf' likewise stands for an infinitely negative real value; for example, we say that `exp(-inf) = 0'. You can have an infinity pointing in any direction on the complex plane: `sqrt(-inf) = i inf'.

The same concept of limits can be used to define 1 / 0. We really want the value that 1 / x approaches as x approaches zero. But if all we have is 1 / 0, we can't tell which direction x was coming from. If x was positive and decreasing toward zero, then we should say that `1 / 0 = inf'. But if x was negative and increasing toward zero, the answer is `1 / 0 = -inf'. In fact, x could be an imaginary number, giving the answer `i inf' or `-i inf'. Calc uses the special symbol `uinf' to mean undirected infinity, i.e., a value which is infinitely large but with an unknown sign (or direction on the complex plane).

Calc actually has three modes that say how infinities are handled. Normally, infinities never arise from calculations that didn't already have them. Thus, 1 / 0 is treated simply as an error and left unevaluated.

The m i (`calc-infinite-mode`) command (see section Infinite Mode) enables a mode in which evaluates to `uinf` instead. There is also an alternative type of infinite mode which says to treat zeros as if they were positive, so that `1 / 0 = inf'. While this is less mathematically correct, it may be the answer you want in some cases.

Since all infinities are "as large" as all others, Calc simplifies, e.g., `5 inf' to `inf'. Another example is `5 - inf = -inf', where the `-inf' is so large that adding a finite number like five to it does not affect it. Note that `a - inf' also results in `-inf'; Calc assumes that variables like `a` always stand for finite quantities. Just to show that infinities really are all the same size, note that `sqrt(inf) = inf^2 = exp(inf) = inf' in Calc's notation.

It's not so easy to define certain formulas like `0 * inf' and `inf / inf'. Depending on where these zeros and infinities came from, the answer could be literally anything. The latter formula could be the limit of x / x (giving a result of one), or 2 x / x (giving two), or x^2 / x (giving `inf`), or x / x^2 (giving zero). Calc uses the symbol `nan` to represent such an indeterminate value. (The name "nan" comes from analogy with the "NAN" concept of IEEE standard arithmetic; it stands for "Not A Number." This is somewhat of a misnomer, since `nan` *does* stand for some number or infinity, it's just that *which* number it stands for cannot be determined.) In Calc's notation, `0 * inf = nan' and `inf / inf = nan'. A few other common indeterminate expressions are `inf - inf' and `inf ^ 0'. Also, `0 / 0 = nan' if you have turned on "infinite mode" (as described above).

Infinities are especially useful as parts of intervals. See section Interval Forms.

Vectors and Matrices

The vector data type is flexible and general. A vector is simply a list of zero or more data objects. When these objects are numbers, the whole is a vector in the mathematical sense. When these objects are themselves vectors of equal (nonzero) length, the whole is a matrix. A vector which is not a matrix is referred to here as a plain vector.

A vector is displayed as a list of values separated by commas and enclosed in square brackets: `[1, 2, 3]'. Thus the following is a 2 row by 3 column matrix: `[[1, 2, 3], [4, 5, 6]]'. Vectors, like complex numbers, are entered as incomplete objects. See section Incomplete Objects. During algebraic entry, vectors are entered all at once in the usual brackets-and-commas form. Matrices may be entered algebraically as nested vectors, or using the shortcut notation `[1, 2, 3; 4, 5, 6]', with rows separated by semicolons. The commas may usually be omitted when entering vectors: `[1 2 3]'. Curly braces may be used in place of brackets: `{1, 2, 3}', but the commas are required in this case.

Traditional vector and matrix arithmetic is also supported; see section Basic Arithmetic and see section Vector/Matrix Functions. Many other operations are applied to vectors element-wise. For example, the complex conjugate of a vector is a vector of the complex conjugates of its elements.

Algebraic functions for building vectors include `vec(a, b, c)' to build `[a, b, c]', `cvec(a, n, m)' to build an @c{$n\times m$} nxm matrix of `a's, and `index(n)' to build a vector of integers from 1 to `n'.

Strings

Character strings are not a special data type in the Calculator. Rather, a string is represented simply as a vector all of whose elements are integers in the range 0 to 255 (ASCII codes). You can enter a string at any time by pressing the " key. Quotation marks and backslashes are written `\"' and `\\', respectively, inside strings. Other notations introduced by backslashes are:

`\a`	7	`\^@`	0
`\b`	8	`\^a-z`	1-26
`\e`	27	`\^[`	27
`\f`	12	`\^\\`	28
`\n`	10	`\^]`	29
`\r`	13	`\^^`	30
`\t`	9	`\^_`	31
		`\^?`	127

Finally, a backslash followed by three octal digits produces any character from its ASCII code.

Strings are normally displayed in vector-of-integers form. The d " (`calc-display-strings`) command toggles a mode in which any vectors of small integers are displayed as quoted strings instead.

The backslash notations shown above are also used for displaying strings. Characters 128 and above are not translated by Calc; unless you have an Emacs modified for 8-bit fonts, these will show up in backslash-octal-digits notation. For characters below 32, and for character 127, Calc uses the backslash-letter combination if there is one, or otherwise uses a `\^' sequence.

The only Calc feature that uses strings is compositions; see section Compositions. Strings also provide a convenient way to do conversions between ASCII characters and integers.

There is a `string` function which provides a different display format for strings. Basically, `string(s)', where s is a vector of integers in the proper range, is displayed as the corresponding string of characters with no surrounding quotation marks or other modifications. Thus `string("ABC")' (or `string([65 66 67])') will look like `ABC' on the stack. This happens regardless of whether d " has been used. The only way to turn it off is to use d U (unformatted language mode) which will display `string("ABC")' instead.

Control characters are displayed somewhat differently by `string`. Characters below 32, and character 127, are shown using `^' notation (same as shown above, but without the backslash). The quote and backslash characters are left alone, as are characters 128 and above.

The `bstring` function is just like `string` except that the resulting string is breakable across multiple lines if it doesn't fit all on one line. Potential break points occur at every space character in the string.

HMS Forms

HMS stands for Hours-Minutes-Seconds; when used as an angular argument, the interpretation is Degrees-Minutes-Seconds. All functions that operate on angles accept HMS forms. These are interpreted as degrees regardless of the current angular mode. It is also possible to use HMS as the angular mode so that calculated angles are expressed in degrees, minutes, and seconds.

The default format for HMS values is `hours@ mins' secs"'. During entry, the letters `h' (for "hours") or `o' (approximating the "degrees" symbol) are accepted as well as `@', `m' is accepted in place of `', and `s' is accepted in place of `"'. The hours value is an integer (or integer-valued float). The mins value is an integer or integer-valued float between 0 and 59. The secs value is a real number between 0 (inclusive) and 60 (exclusive). A positive HMS form is interpreted as hours + mins/60 + secs/3600. A negative HMS form is interpreted as - *hours* - mins/60 - secs/3600. Display format for HMS forms is quite flexible. See section HMS Formats.

HMS forms can be added and subtracted. When they are added to numbers, the numbers are interpreted according to the current angular mode. HMS forms can also be multiplied and divided by real numbers. Dividing two HMS forms produces a real-valued ratio of the two angles.

Just for kicks, M-x calc-time pushes the current time of day on the stack as an HMS form.

Date Forms

A date form represents a date and possibly an associated time. Simple date arithmetic is supported: Adding a number to a date produces a new date shifted by that many days; adding an HMS form to a date shifts it by that many hours. Subtracting two date forms computes the number of days between them (represented as a simple number). Many other operations, such as multiplying two date forms, are nonsensical and are not allowed by Calc.

Date forms are entered and displayed enclosed in `< >' brackets. The default format is, e.g., `<Wed Jan 9, 1991>' for dates, or `<3:32:20pm Wed Jan 9, 1991>' for dates with times. Input is flexible; date forms can be entered in any of the usual notations for dates and times. See section Date Formats.

Date forms are stored internally as numbers, specifically the number of days since midnight on the morning of January 1 of the year 1 AD. If the internal number is an integer, the form represents a date only; if the internal number is a fraction or float, the form represents a date and time. For example, `<6:00am Wed Jan 9, 1991>' is represented by the number 726842.25. The standard precision of 12 decimal digits is enough to ensure that a (reasonable) date and time can be stored without roundoff error.

If the current precision is greater than 12, date forms will keep additional digits in the seconds position. For example, if the precision is 15, the seconds will keep three digits after the decimal point. Decreasing the precision below 12 may cause the time part of a date form to become inaccurate. This can also happen if astronomically high years are used, though this will not be an issue in everyday (or even everymillenium) use. Note that date forms without times are stored as exact integers, so roundoff is never an issue for them.

You can use the v p (`calc-pack`) and v u (`calc-unpack`) commands to get at the numerical representation of a date form. See section Packing and Unpacking.

Date forms can go arbitrarily far into the future or past. Negative year numbers represent years BC. Calc uses a combination of the Gregorian and Julian calendars, following the history of Great Britain and the British colonies. This is the same calendar that is used by the `cal` program in most Unix implementations.

Some historical background: The Julian calendar was created by Julius Caesar in the year 46 BC as an attempt to fix the gradual drift caused by the lack of leap years in the calendar used until that time. The Julian calendar introduced an extra day in all years divisible by four. After some initial confusion, the calendar was adopted around the year we call 8 AD. Some centuries later it became apparent that the Julian year of 365.25 days was itself not quite right. In 1582 Pope Gregory XIII introduced the Gregorian calendar, which added the new rule that years divisible by 100, but not by 400, were not to be considered leap years despite being divisible by four. Many countries delayed adoption of the Gregorian calendar because of religious differences; in Britain it was put off until the year 1752, by which time the Julian calendar had fallen eleven days behind the true seasons. So the switch to the Gregorian calendar in early September 1752 introduced a discontinuity: The day after Sep 2, 1752 is Sep 14, 1752. Calc follows this convention. To take another example, Russia waited until 1918 before adopting the new calendar, and thus needed to remove thirteen days (between Feb 1, 1918 and Feb 14, 1918). This means that Calc's reckoning will be inconsistent with Russian history between 1752 and 1918, and similarly for various other countries.

Today's timekeepers introduce an occasional "leap second" as well, but Calc does not take these minor effects into account. (If it did, it would have to report a non-integer number of days between, say, `<12:00am Mon Jan 1, 1900>' and `<12:00am Sat Jan 1, 2000>'.)

Calc uses the Julian calendar for all dates before the year 1752, including dates BC when the Julian calendar technically had not yet been invented. Thus the claim that day number *-10000* is called "August 16, 28 BC" should be taken with a grain of salt.

Please note that there is no "year 0"; the day before `<Sat Jan 1, +1>' is `<Fri Dec 31, -1>'. These are days 0 and *-1* respectively in Calc's internal numbering scheme.

Another day counting system in common use is, confusingly, also called "Julian." It was invented in 1583 by Joseph Justus Scaliger, who named it in honor of his father Julius Caesar Scaliger. For obscure reasons he chose to start his day numbering on Jan 1, 4713 BC at noon, which in Calc's scheme is *-1721423.5* (recall that Calc starts at midnight instead of noon). Thus to convert a Calc date code obtained by unpacking a date form into a Julian day number, simply add 1721423.5. The Julian code for `6:00am Jan 9, 1991' is 2448265.75. The built-in t J command performs this conversion for you.

The Unix operating system measures time as an integer number of seconds since midnight, Jan 1, 1970. To convert a Calc date value into a Unix time stamp, first subtract 719164 (the code for `<Jan 1, 1970>'), then multiply by 86400 (the number of seconds in a day) and press R to round to the nearest integer. If you have a date form, you can simply subtract the day `<Jan 1, 1970>' instead of unpacking and subtracting 719164. Likewise, divide by 86400 and add `<Jan 1, 1970>' to convert from Unix time to a Calc date form. (Note that Unix normally maintains the time in the GMT time zone; you may need to subtract five hours to get New York time, or eight hours for California time. The same is usually true of

Julian day counts.) The built-in t U command performs these conversions.

Modulo Forms

A modulo form is a real number which is taken modulo (i.e., within an integer multiple of) some value M. Arithmetic modulo M often arises in number theory. Modulo forms are written `*a* `mod` *M*', where a and M are real numbers or HMS forms, and 0 <= a < M. In many applications a and M will be integers but this is not required.

Modulo forms are not to be confused with the modulo operator `%'. The expression `27 % 10' means to compute 27 modulo 10 to produce the result 7. Further computations treat this 7 as just a regular integer. The expression `27 mod 10' produces the result `7 mod 10'; further computations with this value are again reduced modulo 10 so that the result always lies in the desired range.

When two modulo forms with identical M's are added or multiplied, the Calculator simply adds or multiplies the values, then reduces modulo M. If one argument is a modulo form and the other a plain number, the plain number is treated like a compatible modulo form. It is also possible to raise modulo forms to powers; the result is the value raised to the power, then reduced modulo M. (When all values involved are integers, this calculation is done much more efficiently than actually computing the power and then reducing.)

Two modulo forms `*a* `mod` *M*' and `*b* `mod` *M*' can be divided if a, b, and M are all integers. The result is the modulo form which, when multiplied by `*b* `mod` *M*', produces `*a* `mod` *M*'. If there is no solution to this equation (which can happen only when M is non-prime), or if any of the arguments are non-integers, the division is left in symbolic form. Other operations, such as square roots, are not yet supported for modulo forms. (Note that, although `(*a* `mod` *M*)^.5' will compute a "modulo square root" in the sense of reducing @c{$\sqrt a$} sqrt(a) modulo M, this is not a useful definition from the number-theoretical point of view.)

To create a modulo form during numeric entry, press the shift-M key to enter the word `mod'. As a special convenience, pressing shift-M a second time automatically enters the value of M that was most recently used before. During algebraic entry, either type `mod' by hand or press M-m (that's META-m). Once again, pressing this a second time enters the current modulo.

You can also use v p and % to modify modulo forms. See section Building Vectors. See section Basic Arithmetic.

It is possible to mix HMS forms and modulo forms. For example, an HMS form modulo 24 could be used to manipulate clock times; an HMS form modulo 360 would be suitable for angles. Making the modulo M also be an HMS form eliminates troubles that would arise if the angular mode were inadvertently set to Radians, in which case `2@ 0' 0" mod 24' would be interpreted as two degrees modulo 24 radians!

Modulo forms cannot have variables or formulas for components. If you enter the formula `(x + 2) mod 5', Calc propagates the modulus to each of the coefficients: `(1 mod 5) x + (2 mod 5)'.

The algebraic function `makemod(a, m)' builds the modulo form `a mod m'.

Error Forms

An error form is a number with an associated standard deviation, as in `2.3 +/- 0.12'. The notation `*x* +/- @c{σ} sigma' stands for an uncertain value which follows a normal or Gaussian distribution of mean x and standard deviation or "error" @c{σ} sigma. Both the mean and the error can be either numbers or formulas. Generally these are real numbers but the mean may also be complex. If the error is negative or complex, it is changed to its absolute value. An error form with zero error is converted to a regular number by the Calculator.

All arithmetic and transcendental functions accept error forms as input. Operations on the mean-value part work just like operations on regular numbers. The error part for any function f(x) (such as @c{$\sin x$} sin(x)) is defined by the error of x times the derivative of f evaluated at the mean value of x. For a two-argument function f(x,y) (such as addition) the error is the square root of the sum of the squares of the errors due to x and y. Note that this definition assumes the errors in x and y are uncorrelated. A side effect of this definition is that `(2 +/- 1) * (2 +/- 1)' is not the same as `(2 +/- 1)^2'; the former represents the product of two independent values which happen to have the same probability distributions, and the latter is the product of one random value with itself. The former will produce an answer with less error, since on the average the two independent errors can be expected to cancel out.

Consult a good text on error analysis for a discussion of the proper use of standard deviations. Actual errors often are neither Gaussian-distributed nor uncorrelated, and the above formulas are valid only when errors are small. As an example, the error arising from `sin(*x* +/- @c{$\sigma$} *sigma*)' is `@c{σ\nobreak} *sigma* abs(cos(*x*))'. When x is close to zero, cos(x) is close to one so the error in the sine is close to @c{σ} sigma; this makes sense, since @c{$\sin x$} sin(x) is approximately x near zero, so a given error in x will produce about the same error in the sine. Likewise, near 90 degrees @c{$\cos x$} cos(x) is nearly zero and so the computed error is small: The sine curve is nearly flat in that region, so an error in x has relatively little effect on the value of @c{$\sin x$} sin(x). However, consider `sin(90 +/- 1000)'. The cosine of 90 is zero, so Calc will report zero error! We get an obviously wrong result because we have violated the small-error approximation underlying the error analysis. If the error in x had been small, the error in @c{$\sin x$} sin(x) would indeed have been negligible.

To enter an error form during regular numeric entry, use the p ("plus-or-minus") key to type the `+/-' symbol. (If you try actually typing `+/-' the + key will be interpreted as the Calculator's + command!) Within an algebraic formula, you can press M-p to type the `+/-' symbol, or type it out by hand.

Error forms and complex numbers can be mixed; the formulas shown above are used for complex numbers, too; note that if the error part evaluates to a complex number its absolute value (or the square root of the sum of the squares of the absolute values of the two error contributions) is used. Mathematically, this corresponds to a radially symmetric Gaussian distribution of numbers on the complex plane. However, note that Calc considers an error form with real components to represent a real number, not a complex distribution around a real mean.

Error forms may also be composed of HMS forms. For best results, both the mean and the error should be HMS forms if either one is.

The algebraic function `sdev(a, b)' builds the error form `a +/- b'.

Interval Forms

An interval is a subset of consecutive real numbers. For example, the interval `[2 .. 4]' represents all the numbers from 2 to 4, inclusive. If you multiply it by the interval `[0.5 .. 2]' you obtain `[1 .. 8]'. This calculation represents the fact that if you multiply some number in the range `[2 .. 4]' by some other number in the range `[0.5 .. 2]', your result will lie in the range from 1 to 8. Interval arithmetic is used to get a worst-case estimate of the possible range of values a computation will produce, given the set of possible values of the input.

The lower and upper limits of an interval must be either real numbers (or HMS or date forms), or symbolic expressions which are assumed to be real-valued, or `-inf' and `inf'. In general the lower limit must be less than the upper limit. A closed interval containing only one value, `[3 .. 3]', is converted to a plain number (3) automatically. An interval containing no values at all (such as `[3 .. 2]' or `[2 .. 2)') can be represented but is not guaranteed to behave well when used in arithmetic. Note that the interval `[3 .. inf)' represents all real numbers greater than or equal to 3, and `(-inf .. inf)' represents all real numbers. In fact, `[-inf .. inf]' represents all real numbers including the real infinities.

Intervals are entered in the notation shown here, either as algebraic formulas, or using incomplete forms. (See section Incomplete Objects.) In algebraic formulas, multiple periods in a row are collected from left to right, so that `1...1e2' is interpreted as `1.0 .. 1e2' rather than `1 .. 0.1e2'. Add spaces or zeros if you want to get the other interpretation. If you omit the lower or upper limit, a default of `-inf' or `inf' (respectively) is furnished.

"Infinite mode" also affects operations on intervals (see section Infinities). Calc will always introduce an open infinity, as in `1 / (0 .. 2] = [0.5 .. inf)'. But closed infinities, `1 / [0 .. 2] = [0.5 .. inf]', arise only in infinite mode; otherwise they are left unevaluated. Note that the "direction" of a zero is not an issue in this case since the zero is always assumed to be continuous with the rest of the interval. For intervals that contain zero inside them Calc is forced to give the result, `1 / (-2 .. 2) = [-inf .. inf]'.

While it may seem that intervals and error forms are similar, they are based on entirely different concepts of inexact quantities. An error form `*x* +/- @c{$\sigma$} *sigma*' means a variable is random, and its value could be anything but is "probably" within one @c{$\sigma$} *sigma* of the mean value x. An interval `[*a* .. *b*]' means a variable's value is unknown, but guaranteed to lie in the specified range. Error forms are statistical or "average case" approximations; interval arithmetic tends to produce "worst case" bounds on an answer.

Intervals may not contain complex numbers, but they may contain HMS forms or date forms.

See section Set Operations using Vectors, for commands that interpret interval forms as subsets of the set of real numbers.

The algebraic function `intv(n, a, b)' builds an interval form from `a' to `b'; `n' is an integer code which must be 0 for `(..)', 1 for `(..]', 2 for `[..)', or 3 for `[..]'.

Please note that in fully rigorous interval arithmetic, care would be taken to make sure that the computation of the lower bound rounds toward minus infinity, while upper bound computations round toward plus infinity. Calc's arithmetic always uses a round-to-nearest mode, which means that roundoff

errors could creep into an interval calculation to produce intervals slightly smaller than they ought to. For example, entering `[1..2]' and pressing Q 2 ^ should yield the interval `[1..2]' again, but in fact it yields the (slightly too small) interval `[1..1.9999999]' due to roundoff error.

Incomplete Objects

When (or [is typed to begin entering a complex number or vector, respectively, the effect is to push an incomplete complex number or vector onto the stack. The , key adds the value(s) at the top of the stack onto the current incomplete object. The) and] keys "close" the incomplete object after adding any values on the top of the stack in front of the incomplete object.

As a result, the sequence of keystrokes [2 , 3 RET 2 * , 9] pushes the vector `[2, 6, 9]' onto the stack. Likewise, ( 1 , 2 Q ) pushes the complex number `(1, 1.414)' (approximately).

If several values lie on the stack in front of the incomplete object, all are collected and appended to the object. Thus the , key is redundant: [2 RET 3 RET 2 * 9]. Some people prefer the equivalent SPC key to RET.

As a special case, typing , immediately after (, [, or , adds a zero or duplicates the preceding value in the list being formed. Typing DEL during incomplete entry removes the last item from the list.

The ; key is used in the same way as , to create polar complex numbers: (1 ; 2). When entering a vector, ; is useful for creating a matrix. In particular, [[1 , 2 ; 3 , 4 ; 5 , 6]] is equivalent to [[1 , 2] , [3 , 4] , [5 , 6]].

Incomplete entry is also used to enter intervals. For example, [2 .. 4) enters a semi-open interval. Note that when you type the first period, it will be interpreted as a decimal point, but when you type a second period immediately afterward, it is re-interpreted as part of the interval symbol. Typing .. corresponds to executing the `calc-dots` command.

If you find incomplete entry distracting, you may wish to enter vectors and complex numbers as algebraic formulas by pressing the apostrophe key.

Variables

A variable is somewhere between a storage register on a conventional calculator, and a variable in a programming language. (In fact, a Calc variable is really just an Emacs Lisp variable that contains a Calc number or formula.) A variable's name is normally composed of letters and digits. Calc also allows apostrophes and # signs in variable names. The Calc variable `foo` corresponds to the Emacs Lisp variable `var-foo`. Commands like s s (`calc-store`) that operate on variables can be made to use any arbitrary Lisp variable simply by backspacing over the `var-' prefix in the minibuffer.

In a command that takes a variable name, you can either type the full name of a variable, or type a single digit to use one of the special convenience variables `var-q0` through `var-q9`. For example, 3 s s 2 stores the number 3 in variable `var-q2`, and 3 s s foo RET stores that number in variable `var-foo`.

To push a variable itself (as opposed to the variable's value) on the stack, enter its name as an algebraic

expression using the apostrophe (') key. Variable names in algebraic formulas implicitly have `var-' prefixed to their names. The `#' character in variable names used in algebraic formulas corresponds to dash `-' in the Lisp variable name. If the name contains any dashes, the prefix `var-' is *not* automatically added. Thus the two formulas `foo + 1' and `var#foo + 1' both refer to the same variable.

The = (`calc-evaluate`) key "evaluates" a formula by replacing all variables in the formula which have been given values by a `calc-store` or `calc-let` command by their stored values. Other variables are left alone. Thus a variable that has not been stored acts like an abstract variable in algebra; a variable that has been stored acts more like a register in a traditional calculator. With a positive numeric prefix argument, = evaluates the top n stack entries; with a negative argument, = evaluates the nth stack entry.

A few variables are called special constants. Their names are `e', `pi', `i', `phi', and `gamma'. (See section Scientific Functions.) When they are evaluated with =, their values are calculated if necessary according to the current precision or complex polar mode. If you wish to use these symbols for other purposes, simply undefine or redefine them using `calc-store`.

The variables `inf', `uinf', and `nan' stand for infinite or indeterminate values. It's best not to use them as regular variables, since Calc uses special algebraic rules when it manipulates them. Calc displays a warning message if you store a value into any of these special variables.

See section Storing and Recalling, for a discussion of commands dealing with variables.

Formulas

When you press the apostrophe key you may enter any expression or formula in algebraic form. (Calc uses the terms "expression" and "formula" interchangeably.) An expression is built up of numbers, variable names, and function calls, combined with various arithmetic operators. Parentheses may be used to indicate grouping. Spaces are ignored within formulas, except that spaces are not permitted within variable names or numbers. Arithmetic operators, in order from highest to lowest precedence, and with their equivalent function names, are:

`_' [`subscr`] (subscripts);

postfix `%' [`percent`] (as in `25% = 0.25');

prefix `+' and `-' [`neg`] (as in `-x') and prefix `!' [`lnot`] (logical "not," as in `!x');

`+/-' [`sdev`] (the standard deviation symbol) and `mod' [`makemod`] (the symbol for modulo forms);

postfix `!' [`fact`] (factorial, as in `n!') and postfix `!!' [`dfact`] (double factorial);

`^' [`pow`] (raised-to-the-power-of);

`*' [`mul`];

`/' [`div`], `%' [`mod`] (modulo), and `\' [`idiv`] (integer division);

infix `+' [`add`] and `-' [`sub`] (as in `x-y');

`|' [`vconcat`] (vector concatenation);

relations `=' [`eq`], `!=' [`neq`], `<' [`lt`], `>' [`gt`], `<=' [`leq`], and `>=' [`geq`];

`&&' [`land`] (logical "and");

`||' [`lor`] (logical "or");

the C-style "if" operator `a?b:c' [`if`];

`!!!' [`pnot`] (rewrite pattern "not");

`&&&' [`pand`] (rewrite pattern "and");

`|||' [`por`] (rewrite pattern "or");

`:=' [`assign`] (for assignments and rewrite rules);

`::' [`condition`] (rewrite pattern condition);

`=>' [`evalto`].

Note that, unlike in usual computer notation, multiplication binds more strongly than division: `a*b/c*d' is equivalent to @c{$a b \over c d$} (a*b)/(c*d).

The multiplication sign `*' may be omitted in many cases. In particular, if the righthand side is a number, variable name, or parenthesized expression, the `*' may be omitted. Implicit multiplication has the same precedence as the explicit `*' operator. The one exception to the rule is that a variable name followed by a parenthesized expression, as in `f(x)', is interpreted as a function call, not an implicit `*'. In many cases you must use a space if you omit the `*': `2a' is the same as `2*a', and `a b' is the same as `a*b', but `ab' is a variable called `ab', *not* the product of `a' and `b'! Also note that `f (x)' is still a function call.

The rules are slightly different for vectors written with square brackets. In vectors, the space character is interpreted (like the comma) as a separator of elements of the vector. Thus `[ 2a b+c d ]' is equivalent to `[2*a, b+c, d]', whereas `2a b+c d' is equivalent to `2*a*b + c*d'. Note that spaces around the brackets, and around explicit commas, are ignored. To force spaces to be interpreted as multiplication you can enclose a formula in parentheses as in `[(a b) 2(c d)]', which is interpreted as `[a*b, 2*c*d]'. An implicit comma is also inserted between `][', as in the matrix `[[1 2][3 4]]'.

Vectors that contain commas (not embedded within nested parentheses or brackets) do not treat spaces specially: `[a b, 2 c d]' is a vector of two elements. Also, if it would be an error to treat spaces as separators, but not otherwise, then Calc will ignore spaces: `[a - b]' is a vector of one element, but `[a -b]' is a vector of two elements. Finally, vectors entered with curly braces instead of square brackets do not give spaces any special treatment. When Calc displays a vector that does not contain any commas, it will insert parentheses if necessary to make the meaning clear: `[(a b)]'.

The expression `5%-2' is ambiguous; is this five-percent minus two, or five modulo minus-two? Calc always interprets the leftmost symbol as an infix operator preferentially (modulo, in this case), so you would need to write `(5%)-2' to get the former interpretation.

A function call is, e.g., `sin(1+x)'. Function names follow the same rules as variable names except that

the default prefix `calcFunc-' is used (instead of `var-') for the internal Lisp form. Most mathematical Calculator commands like `calc-sin` have function equivalents like `sin`. If no Lisp function is def for a function called by a formula, the call is left as it is during algebraic manipulation: `f(x+y)' is left alone. Beware that many innocent-looking short names like `in` and `re` have predefined meanings which could surprise you; however, single letters or single letters followed by digits are always safe to use for your own function names. See section Index of Algebraic Functions.

In the documentation for particular commands, the notation H S (`calc-sinh`) [`sinh`] means that the key sequence H S, the command M-x calc-sinh, and the algebraic function `sinh(x)` all represent the same operation.

Commands that interpret ("parse") text as algebraic formulas include algebraic entry ('), editing commands like ` which parse the contents of the editing buffer when you finish, the M-# g and M-# r commands, the C-y command, the X window system "paste" mouse operation, and Embedded Mode. All of these operations use the same rules for parsing formulas; in particular, language modes (see section Language Modes) affect them all in the same way.

When you read a large amount of text into the Calculator (say a vector which represents a big set of rewrite rules; see section Rewrite Rules), you may wish to include comments in the text. Calc's formula parser ignores the symbol `%%' and anything following it on a line:

```
[ a + b,   %% the sum of "a" and "b"
  c + d,
  %% last line is coming up:
  e + f ]
```

This is parsed exactly the same as `[a + b, c + d, e + f]'.

See section Syntax Tables, for a way to create your own operators and other input notations. See section Compositions, for a way to create new display formats.

See section Algebra, for commands for manipulating formulas symbolically.

Go to the previous, next section.

Go to the previous, next section.

Stack and Trail Commands

This chapter describes the Calc commands for manipulating objects on the stack and in the trail buffer. (These commands operate on objects of any type, such as numbers, vectors, formulas, and incomplete objects.)

Stack Manipulation Commands

To duplicate the top object on the stack, press RET or SPC (two equivalent keys for the `calc-enter` command). Given a positive numeric prefix argument, these commands duplicate several elements at the top of the stack. Given a negative argument, these commands duplicate the specified element of the stack. Given an argument of zero, they duplicate the entire stack. For example, with `10 20 30' on the stack, RET creates `10 20 30 30', C-u 2 RET creates `10 20 30 20 30', C-u - 2 RET creates `10 20 30 20', and C-u 0 RET creates `10 20 30 10 20 30'.

The LFD (`calc-over`) command (on a key marked Line-Feed if you have it, else on C-j) is like `calc-enter` except that the sign of the numeric prefix argument is interpreted oppositely. Also, with no prefix argument the default argument is 2. Thus with `10 20 30' on the stack, LFD and C-u 2 LFD are both equivalent to C-u - 2 RET, producing `10 20 30 20'.

To remove the top element from the stack, press DEL (`calc-pop`). The C-d key is a synonym for DEL. (If the top element is an incomplete object with at least one element, the last element is removed from it.) Given a positive numeric prefix argument, several elements are removed. Given a negative argument, the specified element of the stack is deleted. Given an argument of zero, the entire stack is emptied. For example, with `10 20 30' on the stack, DEL leaves `10 20', C-u 2 DEL leaves `10', C-u - 2 DEL leaves `10 30', and C-u 0 DEL leaves an empty stack.

The M-DEL (`calc-pop-above`) command is to DEL what LFD is to RET: It interprets the sign of the numeric prefix argument in the opposite way, and the default argument is 2. Thus M-DEL by itself removes the second-from-top stack element, leaving the first, third, fourth, and so on; M-3 M-DEL deletes the third stack element.

To exchange the top two elements of the stack, press TAB (`calc-roll-down`). Given a positive numeric prefix argument, the specified number of elements at the top of the stack are rotated downward. Given a negative argument, the entire stack is rotated downward the specified number of times. Given an argument of zero, the entire stack is reversed top-for-bottom. For example, with `10 20 30 40 50' on the stack, TAB creates `10 20 30 50 40', C-u 3 TAB creates `10 20 50 30 40', C-u - 2 TAB creates `40 50 10 20 30', and C-u 0 TAB creates `50 40 30 20 10'.

The command M-TAB (`calc-roll-up`) is analogous to TAB except that it rotates upward instead of downward. Also, the default with no prefix argument is to rotate the top 3 elements. For example, with `10 20 30 40 50' on the stack, M-TAB creates `10 20 40 50 30', C-u 4 M-TAB creates `10 30 40 50 20', C-u - 2 M-TAB creates `30 40 50 10 20', and C-u 0 M-TAB creates `50 40 30 20 10'.

A good way to view the operation of TAB and M-TAB is in terms of moving a particular element to a new position in the stack. With a positive argument *n*, TAB moves the top stack element down to level *n*, making room for it by pulling all the intervening stack elements toward the top. M-TAB moves the element at level *n* up to the top. (Compare with LFD, which copies instead of moving the element in level *n*.)

With a negative argument *-n*, TAB rotates the stack to move the object in level *n* to the deepest place in the stack, and the object in level *n+1* to the top. M-TAB rotates the deepest stack element to be in level *n*, also putting the top stack element in level *n+1*.

See section Selecting Sub-Formulas, for a way to apply these commands to any portion of a vector or formula on the stack.

Editing Stack Entries

The backquote, ` (`calc-edit`) command creates a temporary buffer (`*Calc Edit*') for editing the top-of-stack value using regular Emacs commands. With a numeric prefix argument, it edits the specified number of stack entries at once. (An argument of zero edits the entire stack; a negative argument edits one specific stack entry.)

When you are done editing, press M-# M-# to finish and return to Calc. The RET and LFD keys also work to finish most sorts of editing, though in some cases Calc leaves RET with its usual meaning ("insert a newline") if it's a situation where you might want to insert new lines into the editing buffer. The traditional Emacs "finish" key sequence, C-c C-c, also works to finish editing and may be easier to type, depending on your keyboard.

When you finish editing, the Calculator parses the lines of text in the `*Calc Edit*' buffer as numbers or formulas, replaces the original stack elements in the original buffer with these new values, then kills the `*Calc Edit*' buffer. The original Calculator buffer continues to exist during editing, but for best results you should be careful not to change it until you have finished the edit. You can also cancel the edit by pressing M-# x.

The formula is normally reevaluated as it is put onto the stack. For example, editing `a + 2' to `3 + 2' and pressing M-# M-# will push 5 on the stack. If you use LFD to finish, Calc will put the result on the stack without evaluating it.

If you give a prefix argument to M-# M-# (or C-c C-c), Calc will not kill the `*Calc Edit*' buffer. You can switch back to that buffer and continue editing if you wish. However, you should understand that if you initiated the edit with `, the M-# M-# operation will be programmed to replace the top of the stack with the new edited value, and it will do this even if you have rearranged the stack in the meanwhile. This is not so much of a problem with other editing commands, though, such as s e (`calc-edit-variable`; see section Other Operations on Variables).

If the `calc-edit` command involves more than one stack entry, each line of the `*Calc Edit*' buffer is interpreted as a separate formula. Otherwise, the entire buffer is interpreted as one formula, with line breaks ignored. (You can use C-o or C-q C-j to insert a newline in the buffer without pressing RET.)

The ` key also works during numeric or algebraic entry. The text entered so far is moved to the `*Calc Edit*` buffer for more extensive editing than is convenient in the minibuffer.

Trail Commands

The commands for manipulating the Calc Trail buffer are two-key sequences beginning with the t prefix.

The t d (`calc-trail-display`) command turns display of the trail on and off. Normally the trail display is toggled on if it was off, off if it was on. With a numeric prefix of zero, this command always turns the trail off; with a prefix of one, it always turns the trail on. The other trail-manipulation commands described here automatically turn the trail on. Note that when the trail is off values are still recorded there; they are simply not displayed. To set Emacs to turn the trail off by default, type t d and then save the mode settings with m m (`calc-save-modes`).

The t i (`calc-trail-in`) and t o (`calc-trail-out`) commands switch the cursor into and out of the Calc Trail window. In practice they are rarely used, since the commands shown below are a more convenient way to move around in the trail, and they work "by remote control" when the cursor is still in the Calculator window.

There is a trail pointer which selects some entry of the trail at any given time. The trail pointer looks like a `>' symbol right before the selected number. The following commands operate on the trail pointer in various ways.

The t y (`calc-trail-yank`) command reads the selected value in the trail and pushes it onto the Calculator stack. It allows you to re-use any previously computed value without retyping. With a numeric prefix argument n, it yanks the value n lines above the current trail pointer.

The t < (`calc-trail-scroll-left`) and t > (`calc-trail-scroll-right`) commands horizontally scroll the trail window left or right by one half of its width.

The t n (`calc-trail-next`) and t p (`calc-trail-previous`) commands move the trail pointer down or up one line. The t f (`calc-trail-forward`) and t b (`calc-trail-backward`) commands move the trail pointer down or up one screenful at a time. All of these commands accept numeric prefix arguments to move several lines or screenfuls at a time.

The t [(`calc-trail-first`) and t] (`calc-trail-last`) commands move the trail pointer to the first or last line of the trail. The t h (`calc-trail-here`) command moves the trail pointer to the cursor position; unlike the other trail commands, t h works only when Calc Trail is the selected window.

The t m (`calc-trail-marker`) command allows you to enter a line of text of your own choosing into the trail. The text is inserted after the line containing the trail pointer; this usually means it is added to the end of the trail. Trail markers are useful mainly as the targets for later incremental searches in the trail.

The t k (`calc-trail-kill`) command removes the selected line from the trail. The line is saved in the Emacs kill ring suitable for yanking into another buffer, but it is not easy to yank the text back into the trail buffer. With a numeric prefix argument, this command kills the n lines below or above the selected one.

The t . (`calc-full-trail-vectors`) command is described elsewhere; see section Vector and Matrix Display Formats.

Keep Arguments

The K (`calc-keep-args`) command acts like a prefix for the following command. It prevents that command from removing its arguments from the stack. For example, after 2 RET 3 +, the stack contains the sole number 5, but after 2 RET 3 K +, the stack contains the arguments and the result: `2 3 5'.

This works for all commands that take arguments off the stack. As another example, K a s simplifies a formula, pushing the simplified version of the formula onto the stack after the original formula (rather than replacing the original formula).

Note that you could get the same effect by typing RET a s, copying the formula and then simplifying the copy. One difference is that for a very large formula the time taken to format the intermediate copy in RET a s could be noticeable; K a s would avoid this extra work.

Even stack manipulation commands are affected. TAB works by popping two values and pushing them back in the opposite order, so 2 RET 3 K TAB produces `2 3 3 2'.

A few Calc commands provide other ways of doing the same thing. For example, ' sin($) replaces the number on the stack with its sine using algebraic entry; to push the sine and keep the original argument you could use either ' sin($1) or K ' sin($). See section Algebraic Entry. Also, the s s command is effectively the same as K s t. See section Storing Variables.

Keyboard macros may interact surprisingly with the K prefix. If you have defined a keyboard macro to be, say, `Q +' to add one number to the square root of another, then typing K X will execute K Q +, probably not what you expected. The K prefix will apply to just the first command in the macro rather than the whole macro.

If you execute a command and then decide you really wanted to keep the argument, you can press M-RET (`calc-last-args`). This command pushes the last arguments that were popped by any command onto the stack. Note that the order of things on the stack will be different than with K: 2 RET 3 + M-RET leaves `5 2 3' on the stack instead of `2 3 5'. See section Undoing Mistakes.

Go to the previous, next section.

Go to the previous, next section.

Mode Settings

This chapter describes commands that set modes in the Calculator. They do not affect the contents of the stack, although they may change the *appearance* or *interpretation* of the stack's contents.

General Mode Commands

You can save all of the current mode settings in your `.emacs' file with the m m (`calc-save-modes`) command. This will cause Emacs to reestablish these modes each time it starts up. The modes saved in the file include everything controlled by the m and d prefix keys, the current precision and binary word size, whether or not the trail is displayed, the current height of the Calc window, and more. The current interface (used when you type M-# M-#) is also saved. If there were already saved mode settings in the file, they are replaced. Otherwise, the new mode information is appended to the end of the file.

The m R (`calc-mode-record-mode`) command tells Calc to record the new mode settings (as if by pressing m m) every time a mode setting changes. If Embedded Mode is enabled, other options are available; see section Mode Settings in Embedded Mode.

The m F (`calc-settings-file-name`) command allows you to choose a different place than your `.emacs' file for m m, Z P, and similar commands to save permanent information. You are prompted for a file name. All Calc modes are then reset to their default values, then settings from the file you named are loaded if this file exists, and this file becomes the one that Calc will use in the future for commands like m m. The default settings file name is `~/.emacs'. You can see the current file name by giving a blank response to the m F prompt. See also the discussion of the `calc-settings-file` variable; see section Installation.

If the file name you give contains the string `.emacs' anywhere inside it, m F will not automatically load the new file. This is because you are presumably switching to your `~/.emacs' file, which may contain other things you don't want to reread. You can give a numeric prefix argument of 1 to m F to force it to read the file no matter what its name. Conversely, an argument of *-1* tells m F *not* to read the new file. An argument of 2 or *-2* tells m F not to reset the modes to their defaults beforehand, which is useful if you intend your new file to have a variant of the modes present in the file you were using before.

The m x (`calc-always-load-extensions`) command enables a mode in which the first use of Calc loads the entire program, including all extensions modules. Otherwise, the extensions modules will not be loaded until the various advanced Calc features are used. Since this mode only has effect when Calc is first loaded, m x is usually followed by m m to make the mode-setting permanent. To load all of Calc just once, rather than always in the future, you can press M-# L.

The m S (`calc-shift-prefix`) command enables a mode in which all of Calc's letter prefix keys may be typed shifted as well as unshifted. If you are typing, say, a S (`calc-solve-for`) quite often

you might find it easier to turn this mode on so that you can type A S instead. When this mode is enabled, the commands that used to be on those single shifted letters (e.g., A (`calc-abs`)) can now be invoked by pressing the shifted letter twice: A A. Note that the v prefix key always works both shifted and unshifted, and the z and Z prefix keys are always distinct. Also, the h prefix is not affected by this mode. Press m S again to disable shifted-prefix mode.

Precision

The p (`calc-precision`) command controls the precision to which floating-point calculations are carried. The precision must be at least 3 digits and may be arbitrarily high, within the limits of memory and time. This affects only floats: Integer and rational calculations are always carried out with as many digits as necessary.

The p key prompts for the current precision. If you wish you can instead give the precision as a numeric prefix argument.

Many internal calculations are carried to one or two digits higher precision than normal. Results are rounded down afterward to the current precision. Unless a special display mode has been selected, floats are always displayed with their full stored precision, i.e., what you see is what you get. Reducing the current precision does not round values already on the stack, but those values will be rounded down before being used in any calculation. The c 0 through c 9 commands (see section Conversions) can be used to round an existing value to a new precision.

It is important to distinguish the concepts of precision and accuracy. In the normal usage of these words, the number 123.4567 has a precision of 7 digits but an accuracy of 4 digits. The precision is the total number of digits not counting leading or trailing zeros (regardless of the position of the decimal point). The accuracy is simply the number of digits after the decimal point (again not counting trailing zeros). In Calc you control the precision, not the accuracy of computations. If you were to set the accuracy instead, then calculations like `exp(100)' would generate many more digits than you would typically need, while `exp(-100)' would probably round to zero! In Calc, both these computations give you exactly 12 (or the requested number of) significant digits.

The only Calc features that deal with accuracy instead of precision are fixed-point display mode for floats (d f; see section Float Formats), and the rounding functions like `floor` and `round` (see section Integer Truncation). Also, c 0 through c 9 deal with both precision and accuracy depending on the magnitudes of the numbers involved.

If you need to work with a particular fixed accuracy (say, dollars and cents with two digits after the decimal point), one solution is to work with integers and an "implied" decimal point. For example, $8.99 divided by 6 would be entered 899 RET 6 /, yielding 149.833 (actually $1.49833 with our implied decimal point); pressing R would round this to 150 cents, i.e., $1.50.

See section Floats, for still more on floating-point precision and related issues.

Inverse and Hyperbolic Flags

There is no single-key equivalent to the `calc-arcsin` function. Instead, you must first press I (`calc-inverse`) to set the Inverse Flag, then press S (`calc-sin`). The I key actually toggles the Inverse Flag. When this flag is set, the word `Inv' appears in the mode line.

Likewise, the H key (`calc-hyperbolic`) sets or clears the Hyperbolic Flag, which transforms `calc-sin` into `calc-sinh`. If both of these flags are set at once, the effect will be `calc-arcsinh`. (The Hyperbolic flag is also used by some non-trigonometric commands; for example H L computes a base-10, instead of base-*e*, logarithm.)

Command names like `calc-arcsin` are provided for completeness, and may be executed with x or M-x. Their effect is simply to toggle the Inverse and/or Hyperbolic flags and then execute the corresponding base command (`calc-sin` in this case).

The Inverse and Hyperbolic flags apply only to the next Calculator command, after which they are automatically cleared. (They are also cleared if the next keystroke is not a Calc command.) Digits you type after I or H (or K) are treated as prefix arguments for the next command, not as numeric entries. The same is true of C-u, but not of the minus sign (K - means to subtract and keep arguments).

The third Calc prefix flag, K (keep-arguments), is discussed elsewhere. See section Keep Arguments.

Calculation Modes

The commands in this section are two-key sequences beginning with the m prefix. (That's the letter m, not the META key.) The `m a' (`calc-algebraic-mode`) command is described elsewhere (see section Algebraic Entry).

Angular Modes

The Calculator supports three notations for angles: radians, degrees, and degrees-minutes-seconds. When a number is presented to a function like `sin` that requires an angle, the current angular mode is used to interpret the number as either radians or degrees. If an HMS form is presented to `sin`, it is always interpreted as degrees-minutes-seconds.

Functions that compute angles produce a number in radians, a number in degrees, or an HMS form depending on the current angular mode. If the result is a complex number and the current mode is HMS, the number is instead expressed in degrees. (Complex-number calculations would normally be done in radians mode, though. Complex numbers are converted to degrees by calculating the complex result in radians and then multiplying by 180 over @c{π} pi.)

The m r (`calc-radians-mode`), m d (`calc-degrees-mode`), and m h (`calc-hms-mode`) commands control the angular mode. The current angular mode is displayed on the Emacs mode line. The default angular mode is degrees.

Polar Mode

The Calculator normally "prefers" rectangular complex numbers in the sense that rectangular form is used when the proper form can not be decided from the input. This might happen by multiplying a rectangular number by a polar one, by taking the square root of a negative real number, or by entering (2 SPC 3).

The m p (`calc-polar-mode`) command toggles complex-number preference between rectangular and polar forms. In polar mode, all of the above example situations would produce polar complex numbers.

Fraction Mode

Division of two integers normally yields a floating-point number if the result cannot be expressed as an integer. In some cases you would rather get an exact fractional answer. One way to accomplish this is to multiply fractions instead: 6 RET 1:4 * produces 3:2 even though 6 RET 4 / produces 1.5.

To set the Calculator to produce fractional results for normal integer divisions, use the m f (`calc-frac-mode`) command. For example, 8/4 produces 2 in either mode, but 6/4 produces 3:2 in Fraction Mode, 1.5 in Float Mode.

At any time you can use c f (`calc-float`) to convert a fraction to a float, or c F (`calc-fraction`) to convert a float to a fraction. See section Conversions.

Infinite Mode

The Calculator normally treats results like 1 / 0 as errors; formulas like this are left in unsimplified form. But Calc can be put into a mode where such calculations instead produce "infinite" results.

The m i (`calc-infinite-mode`) command turns this mode on and off. When the mode is off, infinities do not arise except in calculations that already had infinities as inputs. (One exception is that infinite open intervals like `[0 .. inf)' can be generated; however, intervals closed at infinity (`[0 .. inf]') will not be generated when infinite mode is off.)

With infinite mode turned on, `1 / 0' will generate `uinf`, an undirected infinity. See section Infinities, for a discussion of the difference between `inf` and `uinf`. Also, 0 / 0 evaluates to `nan`, the "indeterminate" symbol. Various other functions can also return infinities in this mode; for example, `ln(0) = -inf', and `gamma(-7) = uinf'. Once again, note that `exp(inf) = inf' regardless of infinite mode because this calculation has infinity as an input.

The m i command with a numeric prefix argument of zero, i.e., C-u 0 m i, turns on a "positive infinite mode" in which zero is treated as positive instead of being directionless. Thus, `1 / 0 = inf' and `-1 / 0 = -inf' in this mode. Note that zero never actually has a sign in Calc; there are no separate representations for *+0* and *-0*. Positive infinite mode merely changes the interpretation given to the single symbol, `0'. One consequence of this is that, while you might expect `1 / -0 = -inf', actually `1 / -0' is equivalent to `1 / 0', which is equal to positive `inf`.

Symbolic Mode

Calculations are normally performed numerically wherever possible. For example, the `calc-sqrt` command, or `sqrt` function in an algebraic expression, produces a numeric answer if the argument is a number or a symbolic expression if the argument is an expression: 2 Q pushes 1.4142 but ' x+1 RET Q pushes `sqrt(x+1)'.

In symbolic mode, controlled by the m s (`calc-symbolic-mode`) command, functions which would produce inexact, irrational results are left in symbolic form. Thus 16 Q pushes 4, but 2 Q pushes `sqrt(2)'.

The shift-N (`calc-eval-num`) command evaluates numerically the expression at the top of the stack, by temporarily disabling `calc-symbolic-mode` and executing = (`calc-evaluate`). Given a numeric prefix argument, it also sets the floating-point precision to the specified value for the duration of the command.

To evaluate a formula numerically without expanding the variables it contains, you can use the key sequence m s a v m s (this uses `calc-alg-evaluate`, which resimplifies but doesn't evaluate variables.)

Matrix and Scalar Modes

Calc sometimes makes assumptions during algebraic manipulation that are awkward or incorrect when vectors and matrices are involved. Calc has two modes, matrix mode and scalar mode, which modify its behavior around vectors in useful ways.

Press m v (`calc-matrix-mode`) once to enter matrix mode. In this mode, all objects are assumed to be matrices unless provably otherwise. One major effect is that Calc will no longer consider multiplication to be commutative. (Recall that in matrix arithmetic, `A*B' is not the same as `B*A'.) This assumption affects rewrite rules and algebraic simplification. Another effect of this mode is that calculations that would normally produce constants like 0 and 1 (e.g., a - a and a / a, respectively) will now produce function calls that represent "generic" zero or identity matrices: `idn(0)', `idn(1)'. The `idn` function `idn(a,n)' returns a times an nxn identity matrix; if n is omitted, it doesn't know what dimension to use and so the `idn` call remains in symbolic form. However, if this generic identity matrix is later combined with a matrix whose size is known, it will be converted into a true identity matrix of the appropriate size. On the other hand, if it is combined with a scalar (as in `idn(1) + 2'), Calc will assume it really was a scalar after all and produce, e.g., 3.

Press m v a second time to get scalar mode. Here, objects are assumed *not* to be vectors or matrices unless provably so. For example, normally adding a variable to a vector, as in `[x, y, z] + a', will leave the sum in symbolic form because as far as Calc knows, `a' could represent either a number or another 3-vector. In scalar mode, `a' is assumed to be a non-vector, and the addition is evaluated to `[x+a, y+a, z+a]'.

Press m v a third time to return to the normal mode of operation.

If you press m v with a numeric prefix argument n, you get a special "dimensioned matrix mode" in which matrices of unknown size are assumed to be nxn square matrices. Then, the function call `idn(1)' will expand into an actual matrix rather than representing a "generic" matrix.

Of course these modes are approximations to the true state of affairs, which is probably that some quantities will be matrices and others will be scalars. One solution is to "declare" certain variables or functions to be scalar-valued. See section Declarations, to see how to make declarations in Calc.

There is nothing stopping you from declaring a variable to be scalar and then storing a matrix in it; however, if you do, the results you get from Calc may not be valid. Suppose you let Calc get the result `[x+a, y+a, z+a]' shown above, and then stored `[1, 2, 3]' in `a'. The result would not be the same as for `[x, y, z] + [1, 2, 3]', but that's because you have broken your earlier promise to Calc that `a' would be scalar.

Another way to mix scalars and matrices is to use selections (see section Selecting Sub-Formulas). Use matrix mode when operating on your formula normally; then, to apply scalar mode to a certain part of the formula without affecting the rest just select that part, change into scalar mode and press = to resimplify the part under this mode, then change back to matrix mode before deselecting.

Automatic Recomputation

The evaluates-to operator, `=>', has the special property that any `=>' formulas on the stack are recomputed whenever variable values or mode settings that might affect them are changed. See section The Evaluates-To Operator.

The m C (`calc-auto-recompute`) command turns this automatic recomputation on and off. If you turn it off, Calc will not update `=>' operators on the stack (nor those in the attached Embedded Mode buffer, if there is one). They will not be updated unless you explicitly do so by pressing = or until you press m C to turn recomputation back on. (While automatic recomputation is off, you can think of m C m C as a command to update all `=>' operators while leaving recomputation off.)

To update `=>' operators in an Embedded buffer while automatic recomputation is off, use M-# u. See section Embedded Mode.

Working Messages

Since the Calculator is written entirely in Emacs Lisp, which is not designed for heavy numerical work, many operations are quite slow. The Calculator normally displays the message `Working...' in the echo area during any command that may be slow. In addition, iterative operations such as square roots and trigonometric functions display the intermediate result at each step. Both of these types of messages can be disabled if you find them distracting.

Type m w (`calc-working`) with a numeric prefix of 0 to disable all "working" messages. Use a numeric prefix of 1 to enable only the plain `Working...' message. Use a numeric prefix of 2 to see intermediate results as well. With no numeric prefix this displays the current mode.

While it may seem that the "working" messages will slow Calc down considerably, experiments have shown that their impact is actually quite small. But if your terminal is slow you may find that it helps to turn the messages off.

Simplification Modes

The current simplification mode controls how numbers and formulas are "normalized" when being taken from or pushed onto the stack. Some normalizations are unavoidable, such as rounding floating-point results to the current precision, and reducing fractions to simplest form. Others, such as simplifying a formula like a+a (or 2+3), are done by default but can be turned off when necessary.

When you press a key like + when 2 and 3 are on the stack, Calc pops these numbers, normalizes them, creates the formula 2+3, normalizes it, and pushes the result. Of course the standard rules for normalizing 2+3 will produce the result 5.

Simplification mode commands consist of the lower-case m prefix key followed by a shifted letter.

The m O (`calc-no-simplify-mode`) command turns off all optional simplifications. These would leave a formula like 2+3 alone. In fact, nothing except simple numbers are ever affected by normalization in this mode.

The m N (`calc-num-simplify-mode`) command turns off simplification of any formulas except those for which all arguments are constants. For example, 1+2 is simplified to 3, and a+(2-2) is simplified to a+0 but no further, since one argument of the sum is not a constant. Unfortunately, (a+2)-2 is *not* simplified because the top-level `-' operator's arguments are not both constant numbers (one of them is the formula a+2). A constant is a number or other numeric object (such as a constant error form or modulo form), or a vector all of whose elements are constant.

The m D (`calc-default-simplify-mode`) command restores the default simplifications for all formulas. This includes many easy and fast algebraic simplifications such as a+0 to a, and a + 2 a to 3 a, as well as evaluating functions like `deriv`(x^2, x) to 2 x.

The m B (`calc-bin-simplify-mode`) mode applies the default simplifications to a result and then, if the result is an integer, uses the b c (`calc-clip`) command to clip the integer according to the current binary word size. See section Binary Number Functions. Real numbers are rounded to the nearest integer and then clipped; other kinds of results (after the default simplifications) are left alone.

The m A (`calc-alg-simplify-mode`) mode does algebraic simplification; it applies all the default simplifications, and also the more powerful (and slower) simplifications made by a s (`calc-simplify`). See section Algebraic Simplifications.

The m E (`calc-ext-simplify-mode`) mode does "extended" algebraic simplification, as by the a e (`calc-simplify-extended`) command. See section "Unsafe" Simplifications.

The m U (`calc-units-simplify-mode`) mode does units simplification; it applies the command u s (`calc-simplify-units`), which in turn is a superset of a s. In this mode, variable names which are identifiable as unit names (like `mm' for "millimeters") are simplified with their unit definitions in mind.

A common technique is to set the simplification mode down to the lowest amount of simplification you will allow to be applied automatically, then use manual commands like a s and c c (`calc-clean`) to perform higher types of simplifications on demand. See section Programming with Formulas, for another sample use of no-simplification mode.

Declarations

A declaration is a statement you make that promises you will use a certain variable or function in a restricted way. This may give Calc the freedom to do things that it couldn't do if it had to take the fully general situation into account.

Declaration Basics

The s d (`calc-declare-variable`) command is the easiest way to make a declaration for a variable. This command prompts for the variable name, then prompts for the declaration. The default at the declaration prompt is the previous declaration, if any. You can edit this declaration, or press C-k to erase it and type a new declaration. (Or, erase it and press RET to clear the declaration, effectively "undeclaring" the variable.)

A declaration is in general a vector of type symbols and range values. If there is only one type symbol or range value, you can write it directly rather than enclosing it in a vector. For example, s d foo RET real RET declares `foo` to be a real number, and s d bar RET [int, const, [1..6]] RET declares `bar` to be a constant integer between 1 and 6. (Actually, you can omit the outermost brackets and Calc will provide them for you: s d bar RET int, const, [1..6] RET.)

Declarations in Calc are kept in a special variable called `Decls`. This variable encodes the set of all outstanding declarations in the form of a matrix. Each row has two elements: A variable or vector of variables declared by that row, and the declaration specifier as described above. You can use the s D command to edit this variable if you wish to see all the declarations at once. See section Other Operations on Variables, for a description of this command and the s p command that allows you to save your declarations permanently if you wish.

Items being declared can also be function calls. The arguments in the call are ignored; the effect is to say that this function returns values of the declared type for any valid arguments. The s d command declares only variables, so if you wish to make a function declaration you will have to edit the `Decls` matrix yourself.

For example, the declaration matrix

```
[ [ foo,       real       ]
  [ [j, k, n], int        ]
  [ f(1,2,3),  [0 .. inf) ] ]
```

declares that `foo` represents a real number, `j`, `k` and `n` represent integers, and the function `f` always returns a real number in the interval shown.

If there is a declaration for the variable `All`, then that declaration applies to all variables that are not otherwise declared. It does not apply to function names. For example, using the row `[All, real]' says that all your variables are real unless they are explicitly declared without `real` in some other row. The s d command declares `All` if you give a blank response to the variable-name prompt.

Kinds of Declarations

The type-specifier part of a declaration (that is, the second prompt in the s d command) can be a type symbol, an interval, or a vector consisting of zero or more type symbols followed by zero or more intervals or numbers that represent the set of possible values for the variable.

```
[ [ a, [1, 2, 3, 4, 5] ]
  [ b, [1 .. 5]        ]
  [ c, [int, 1 .. 5]   ] ]
```

Here `a` is declared to contain one of the five integers shown; `b` is any number in the interval from 1 to 5 (any real number since we haven't specified), and `c` is any integer in that interval. Thus the declarations for `a` and `c` are nearly equivalent (see below).

The type-specifier can be the empty vector `[]' to say that nothing is known about a given variable's value. This is the same as not declaring the variable at all except that it overrides any `All` declaration which would otherwise apply.

The initial value of `Decls` is the empty vector `[]'. If `Decls` has no stored value or if the value stored in it is not valid, it is ignored and there are no declarations as far as Calc is concerned. (The s d command will replace such a malformed value with a fresh empty matrix, `[]', before recording the new declaration.) Unrecognized type symbols are ignored.

The following type symbols describe what sorts of numbers will be stored in a variable:

`int`
: Integers.

`numint`
: Numerical integers. (Integers or integer-valued floats.)

`frac`
: Fractions. (Rational numbers which are not integers.)

`rat`
: Rational numbers. (Either integers or fractions.)

`float`
: Floating-point numbers.

`real`
: Real numbers. (Integers, fractions, or floats. Actually, intervals and error forms with real components also count as reals here.)

`pos`
: Positive real numbers. (Strictly greater than zero.)

`nonneg`
: Nonnegative real numbers. (Greater than or equal to zero.)

`number`

Numbers. (Real or complex.)

Calc uses this information to determine when certain simplifications of formulas are safe. For example, `(x^y)^z' cannot be simplified to `x^(y z)' in general; for example, `((-3)^2)^1:2' is 3, but `(-3)^(2*1:2) = (-3)^1' is *-3*. However, this simplification *is* safe if `z` is known to be an integer, or if `x` is known to be a nonnegative real number. If you have given declarations that allow Calc to deduce either of these facts, Calc will perform this simplification of the formula.

Calc can apply a certain amount of logic when using declarations. For example, `(x^y)^(2n+1)' will be simplified if `n` has been declared `int`; Calc knows that an integer times an integer, plus an integer, must always be an integer. (In fact, Calc would simplify `(-x)^(2n+1)' to `-(x^(2n+1))' since it is able to determine that `2n+1' must be an odd integer.)

Similarly, `(abs(x)^y)^z' will be simplified to `abs(x)^(y z)' because Calc knows that the `abs` function always returns a nonnegative real. If you had a `myabs` function that also had this property, you could get Calc to recognize it by adding the row `[myabs(), nonneg]' to the `Decls` matrix.

One instance of this simplification is `sqrt(x^2)' (since the `sqrt` function is effectively a one-half power). Normally Calc leaves this formula alone. After the command s d x RET real RET, however, it can simplify the formula to `abs(x)'. And after s d x RET nonneg RET, Calc can simplify this formula all the way to `x'.

If there are any intervals or real numbers in the type specifier, they comprise the set of possible values that the variable or function being declared can have. In particular, the type symbol `real` is effectively the same as the range `[-inf .. inf]' (note that infinity is included in the range of possible values); `pos` is the same as `(0 .. inf]', and `nonneg` is the same as `[0 .. inf]'. Saying `[real, [-5 .. 5]]' is redundant because the fact that the variable is real can be deduced just from the interval, but `[int, [-5 .. 5]]' and `[rat, [-5 .. 5]]' are useful combinations.

Note that the vector of intervals or numbers is in the same format used by Calc's set-manipulation commands. See section Set Operations using Vectors.

The type specifier `[1, 2, 3]' is equivalent to `[numint, 1, 2, 3]', *not* to `[int, 1, 2, 3]'. In other words, the range of possible values means only that the variable's value must be numerically equal to a number in that range, but not that it must be equal in type as well. Calc's set operations act the same way; `in(2, [1., 2., 3.])' and `in(1.5, [1:2, 3:2, 5:2])' both report "true."

If you use a conflicting combination of type specifiers, the results are unpredictable. An example is `[pos, [0 .. 5]]', where the interval does not lie in the range described by the type symbol.

"Real" declarations mostly affect simplifications involving powers like the one described above. Another case where they are used is in the a P command which returns a list of all roots of a polynomial; if the variable has been declared real, only the real roots (if any) will be included in the list.

"Integer" declarations are used for simplifications which are valid only when certain values are integers (such as `(x^y)^z' shown above).

Another command that makes use of declarations is a s, when simplifying equations and inequalities. It will cancel `x` from both sides of `a x = b x' only if it is sure `x` is non-zero, say, because it has a `pos`

declaration. To declare specifically that `x` is real and non-zero, use `[[-inf .. 0), (0 .. inf]]'. (There is n way in the current notation to say that `x` is nonzero but not necessarily real.) The a e command does "unsafe" simplifications, including cancelling `x' from the equation when `x' is not known to be nonzero.

Another set of type symbols distinguish between scalars and vectors.

`scalar`

The value is not a vector.

`vector`

The value is a vector.

`matrix`

The value is a matrix (a rectangular vector of vectors).

These type symbols can be combined with the other type symbols described above; `[int, matrix]' describes an object which is a matrix of integers.

Scalar/vector declarations are used to determine whether certain algebraic operations are safe. For example, `[a, b, c] + x' is normally not simplified to `[a + x, b + x, c + x]', but it will be if `x` has been declared `scalar`. On the other hand, multiplication is usually assumed to be commutative, but the terms in `x y' will never be exchanged if both `x` and `y` are known to be vectors or matrices. (Calc currently never distinguishes between `vector` and `matrix` declarations.)

See section Matrix and Scalar Modes, for a discussion of "matrix mode" and "scalar mode," which are similar to declaring `[All, matrix]' or `[All, scalar]' but much more convenient.

One more type symbol that is recognized is used with the H a d command for taking total derivatives of a formula. See section Calculus.

`const`

The value is a constant with respect to other variables.

Calc does not check the declarations for a variable when you store a value in it. However, storing *-3.5* in a variable that has been declared `pos`, `int`, or `matrix` may have unexpected effects; Calc may evaluate `sqrt(x^2)' to 3.5 if it substitutes the value first, or to -3.5 if `x` was declared `pos` and the formula `sqrt(x^2)' is simplified to `x' before the value is substituted. Before using a variable for a new purpose, it is best to use s d or s D to check to make sure you don't still have an old declaration for the variable that will conflict with its new meaning.

Functions for Declarations

Calc has a set of functions for accessing the current declarations in a convenient manner. These functions return 1 if the argument can be shown to have the specified property, or 0 if the argument can be shown *not* to have that property; otherwise they are left unevaluated. These functions are suitable for use with rewrite rules (see section Conditional Rewrite Rules) or programming constructs (see section Conditionals in Keyboard Macros). They can be entered only using algebraic notation. See section Logical Operations, for functions that perform other tests not related to declarations.

For example, `dint(17)' returns 1 because 17 is an integer, as do `dint(n)' and `dint(2 n - 3)' if `n' has b declared `int', but `dint(2.5)' and `dint(n + 0.5)' return 0. Calc consults knowledge of its own built-in functions as well as your own declarations: `dint(floor(x))' returns 1.

The `dint` function checks if its argument is an integer. The `dnatnum` function checks if its argument is a natural number, i.e., a nonnegative integer. The `dnumint` function checks if its argument is numerically an integer, i.e., either an integer or an integer-valued float. Note that these and the other data type functions also accept vectors or matrices composed of suitable elements, and that real infinities `inf' and `-inf' are considered to be integers for the purposes of these functions.

The `drat` function checks if its argument is rational, i.e., an integer or fraction. Infinities count as rational, but intervals and error forms do not.

The `dreal` function checks if its argument is real. This includes integers, fractions, floats, real error forms, and intervals.

The `dimag` function checks if its argument is imaginary, i.e., is mathematically equal to a real number times i.

The `dpos` function checks for positive (but nonzero) reals. The `dneg` function checks for negative reals. The `dnonneg` function checks for nonnegative reals, i.e., reals greater than or equal to zero. Note that the a s command can simplify an expression like x > 0 to 1 or 0 using `dpos`, and that a s is effectively applied to all conditions in rewrite rules, so the actual functions `dpos`, `dneg`, and `dnonneg` are rarely necessary.

The `dnonzero` function checks that its argument is nonzero. This includes all nonzero real or complex numbers, all intervals that do not include zero, all nonzero modulo forms, vectors all of whose elements are nonzero, and variables or formulas whose values can be deduced to be nonzero. It does not include error forms, since they represent values which could be anything including zero. (This is also the set of objects considered "true" in conditional contexts.)

The `deven` function returns 1 if its argument is known to be an even integer (or integer-valued float); it returns 0 if its argument is known not to be even (because it is known to be odd or a non-integer). The a s command uses this to simplify a test of the form `x % 2 = 0'. There is also an analogous `dodd` function.

The `drange` function returns a set (an interval or a vector of intervals and/or numbers; see section Set Operations using Vectors) that describes the set of possible values of its argument. If the argument is a variable or a function with a declaration, the range is copied from the declaration. Otherwise, the possible signs of the expression are determined using a method similar to `dpos`, etc., and a suitable set like `[0 .. inf]' is returned. If the expression is not provably real, the `drange` function remains unevaluated.

The `dscalar` function returns 1 if its argument is provably scalar, or 0 if its argument is provably non-scalar. It is left unevaluated if this cannot be determined. (If matrix mode or scalar mode are in effect, this function returns 1 or 0, respectively, if it has no other information.) When Calc interprets a condition (say, in a rewrite rule) it considers an unevaluated formula to be "false." Thus, `dscalar(a)' is "true" only if `a` is provably scalar, and `!dscalar(a)' is "true" only if `a` is provably non-scalar; both are "false" if there is insufficient information to tell.

Display Modes

The commands in this section are two-key sequences beginning with the d prefix. The d l (`calc-line-numbering`) and d b (`calc-line-breaking`) commands are described elsewhere; see section Stack Basics and see section Normal Language Modes, respectively. Display formats for vectors and matrices are also covered elsewhere; see section Vector and Matrix Display Formats.

One thing all display modes have in common is their treatment of the H prefix. This prefix causes any mode command that would normally refresh the stack to leave the stack display alone. The word "Dirty" will appear in the mode line when Calc thinks the stack display may not reflect the latest mode settings.

The d RET (`calc-refresh-top`) command reformats the top stack entry according to all the current modes. Positive prefix arguments reformat the top n entries; negative prefix arguments reformat the specified entry, and a prefix of zero is equivalent to d SPC (`calc-refresh`), which reformats the entire stack. For example, H d s M-2 d RET changes to scientific notation but reformats only the top two stack entries in the new mode.

The I prefix has another effect on the display modes. The mode is set only temporarily; the top stack entry is reformatted according to that mode, then the original mode setting is restored. In other words, I d s is equivalent to H d s d RET H d (old mode).

Radix Modes

Calc normally displays numbers in decimal (base-10 or radix-10) notation. Calc can actually display in any radix from two (binary) to 36. When the radix is above 10, the letters `A` to `Z` are used as digits. When entering such a number, letter keys are interpreted as potential digits rather than terminating numeric entry mode.

The key sequences d 2, d 8, d 6, and d 0 select binary, octal, hexadecimal, and decimal as the current display radix, respectively. Numbers can always be entered in any radix, though the current radix is used as a default if you press # without any initial digits. A number entered without a # is *always* interpreted as decimal.

To set the radix generally, use d r (`calc-radix`) and enter an integer from 2 to 36. You can specify the radix as a numeric prefix argument; otherwise you will be prompted for it.

Integers normally are displayed with however many digits are necessary to represent the integer and no more. The d z (`calc-leading-zeros`) command causes integers to be padded out with leading zeros according to the current binary word size. (See section Binary Number Functions, for a discussion of word size.) If the absolute value of the word size is w, all integers are displayed with at least enough digits to represent @c{2^w-1} (2^w)-1 in the current radix. (Larger integers will still be displayed in their entirety.)

Grouping Digits

Long numbers can be hard to read if they have too many digits. For example, the factorial of 30 is 33 digits long! Press d g (`calc-group-digits`) to enable grouping mode, in which digits are displayed in clumps of 3 or 4 (depending on the current radix) separated by commas.

The d g command toggles grouping on and off. With a numerix prefix of 0, this command displays the current state of the grouping flag; with an argument of minus one it disables grouping; with a positive argument N it enables grouping on every N digits. For floating-point numbers, grouping normally occurs only before the decimal point. A negative prefix argument -N enables grouping every N digits both before and after the decimal point.

The d , (`calc-group-char`) command allows you to choose any character as the grouping separator. The default is the comma character. If you find it difficult to read vectors of large integers grouped with commas, you may wish to use spaces or some other character instead. This command takes the next character you type, whatever it is, and uses it as the digit separator. As a special case, d , \ selects `\,' (TeX's thin-space symbol) as the digit separator.

Please note that grouped numbers will not generally be parsed correctly if re-read in textual form, say by the use of M-# y and M-# g. (See section Kill and Yank Functions, for details on these commands.) One exception is the `\,' separator, which doesn't interfere with parsing because it is ignored by TeX language mode.

Float Formats

Floating-point quantities are normally displayed in standard decimal form, with scientific notation used if the exponent is especially high or low. All significant digits are normally displayed. The commands in this section allow you to choose among several alternative display formats for floats.

The d n (`calc-normal-notation`) command selects the normal display format. All significant figures in a number are displayed. With a positive numeric prefix, numbers are rounded if necessary to that number of significant digits. With a negative numerix prefix, the specified number of significant digits less than the current precision is used. (Thus C-u -2 d n displays 10 digits if the current precision is 12.)

The d f (`calc-fix-notation`) command selects fixed-point notation. The numeric argument is the number of digits after the decimal point, zero or more. This format will relax into scientific notation if a nonzero number would otherwise have been rounded all the way to zero. Specifying a negative number of digits is the same as for a positive number, except that small nonzero numbers will be rounded to zero rather than switching to scientific notation.

The d s (`calc-sci-notation`) command selects scientific notation. A positive argument sets the number of significant figures displayed, of which one will be before and the rest after the decimal point. A negative argument works the same as for d n format. The default is to display all significant digits.

The d e (`calc-eng-notation`) command selects engineering notation. This is similar to scientific notation except that the exponent is rounded down to a multiple of three, with from one to three digits before the decimal point. An optional numeric prefix sets the number of significant digits to display, as

for d s.

It is important to distinguish between the current *precision* and the current *display format*. After the commands C-u 10 p and C-u 6 d n the Calculator computes all results to ten significant figures but displays only six. (In fact, intermediate calculations are often carried to one or two more significant figures, but values placed on the stack will be rounded down to ten figures.) Numbers are never actually rounded to the display precision for storage, except by commands like C-k and M-# y which operate on the actual displayed text in the Calculator buffer.

The d . (`calc-point-char`) command selects the character used as a decimal point. Normally this is a period; users in some countries may wish to change this to a comma. Note that this is only a display style; on entry, periods must always be used to denote floating-point numbers, and commas to separate elements in a list.

Complex Formats

There are three supported notations for complex numbers in rectangular form. The default is as a pair of real numbers enclosed in parentheses and separated by a comma: `(a,b)'. The d c (`calc-complex-notation`) command selects this style.

The other notations are d i (`calc-i-notation`), in which numbers are displayed in `a+bi' form, and d j (`calc-j-notation`) which displays the form `a+bj' preferred in some disciplines.

Complex numbers are normally entered in `(a,b)' format. If you enter `2+3i' as an algebraic formula, it will be stored as the formula `2 + 3 * i'. However, if you use = to evaluate this formula and you have not changed the variable `i', the `i' will be interpreted as `(0,1)' and the formula will be simplified to `(2,3)'. Other commands (like `calc-sin`) will *not* interpret the formula `2 + 3 * i' as a complex number. See section Variables, under "special constants."

Fraction Formats

Display of fractional numbers is controlled by the d o (`calc-over-notation`) command. By default, a number like eight thirds is displayed in the form `8:3'. The d o command prompts for a one- or two-character format. If you give one character, that character is used as the fraction separator. Common separators are `:' and `/'. (During input of numbers, the : key must be used regardless of the display format; in particular, the / is used for RPN-style division, *not* for entering fractions.)

If you give two characters, fractions use "integer-plus-fractional-part" notation. For example, the format `+/' would display eight thirds as `2+2/3'. If two colons are present in a number being entered, the number is interpreted in this form (so that the entries 2:2:3 and 8:3 are equivalent).

It is also possible to follow the one- or two-character format with a number. For example: `:10' or `+/3'. In this case, Calc adjusts all fractions that are displayed to have the specified denominator, if possible. Otherwise it adjusts the denominator to be a multiple of the specified value. For example, in `:6' mode the fraction 1:6 will be unaffected, but 2:3 will be displayed as 4:6, 1:2 will be displayed as 3:6, and 1:8 will be displayed as 3:24. Integers are also affected by this mode: 3 is displayed as 18:6. Note that the format `:1' writes fractions the same as `:', but it writes integers as n:1.

The fraction format does not affect the way fractions or integers are stored, only the way they appear the screen. The fraction format never affects floats.

HMS Formats

The d h (`calc-hms-notation`) command controls the display of HMS (hours-minutes-seconds) forms. It prompts for a string which consists basically of an "hours" marker, optional punctuation, a "minutes" marker, more optional punctuation, and a "seconds" marker. Punctuation is zero or more spaces, commas, or semicolons. The hours marker is one or more non-punctuation characters. The minutes and seconds markers must be single non-punctuation characters.

The default HMS format is `@ ' "', producing HMS values of the form `23@ 30' 15.75"'. The format `deg, ms' would display this same value as `23deg, 30m15.75s'. During numeric entry, the h or o keys are recognized as synonyms for @ regardless of display format. The m and s keys are recognized as synonyms for ' and ", respectively, but only if an @ (or h or o) has already been typed; otherwise, they have their usual meanings (m- prefix and s- prefix). Thus, 5 ", 0 @ 5 ", and 0 h 5 s are some of the ways to enter the quantity "five seconds." The ' key is recognized as "minutes" only if @ (or h or o) has already been pressed; otherwise it means to switch to algebraic entry.

Date Formats

The d d (`calc-date-notation`) command controls the display of date forms (see section Date Forms). It prompts for a string which contains letters that represent the various parts of a date and time. To show which parts should be omitted when the form represents a pure date with no time, parts of the string can be enclosed in `< >' marks. If you don't include `< >' markers in the format, Calc guesses at which parts, if any, should be omitted when formatting pure dates.

The default format is: `<H:mm:SSpp >Www Mmm D, YYYY'. An example string in this format is `3:32pm Wed Jan 9, 1991'. If you enter a blank format string, this default format is reestablished.

Calc uses `< >' notation for nameless functions as well as for dates. See section Specifying Operators. To avoid confusion with nameless functions, your date formats should avoid using the `#' character.

Date Formatting Codes

When displaying a date, the current date format is used. All characters except for letters and `<' and `>' are copied literally when dates are formatted. The portion between `< >' markers is omitted for pure dates, or included for date/time forms. Letters are interpreted according to the table below.

When dates are read in during algebraic entry, Calc first tries to match the input string to the current format either with or without the time part. The punctuation characters (including spaces) must match exactly; letter fields must correspond to suitable text in the input. If this doesn't work, Calc checks if the input is a simple number; if so, the number is interpreted as a number of days since Jan 1, 1 AD. Otherwise, Calc tries a much more relaxed and flexible algorithm which is described in the next section.

Weekday names are ignored during reading.

Two-digit year numbers are interpreted as lying in the range from 1941 to 2039. Years outside that ra are always entered and displayed in full. Year numbers with a leading `+' sign are always interpreted exactly, allowing the entry and display of the years 1 through 99 AD.

Here is a complete list of the formatting codes for dates:

Y

Year: "91" for 1991, "7" for 2007, "+23" for 23 AD.

YY

Year: "91" for 1991, "07" for 2007, "+23" for 23 AD.

BY

Year: "91" for 1991, " 7" for 2007, "+23" for 23 AD.

YYY

Year: "1991" for 1991, "23" for 23 AD.

YYYY

Year: "1991" for 1991, "+23" for 23 AD.

aa

Year: "ad" or blank.

AA

Year: "AD" or blank.

aaa

Year: "ad " or blank. (Note trailing space.)

AAA

Year: "AD " or blank.

aaaa

Year: "a.d." or blank.

AAAA

Year: "A.D." or blank.

bb

Year: "bc" or blank.

BB

Year: "BC" or blank.

bbb

Year: " bc" or blank. (Note leading space.)

BBB

Year: " BC" or blank.

bbbb

Year: "b.c." or blank.

BBBB

Year: "B.C." or blank.

M

Month: "8" for August.

MM

Month: "08" for August.

BM

Month: " 8" for August.

MMM

Month: "AUG" for August.

Mmm

Month: "Aug" for August.

mmm

Month: "aug" for August.

MMMM

Month: "AUGUST" for August.

Mmmm

Month: "August" for August.

D

Day: "7" for 7th day of month.

DD

Day: "07" for 7th day of month.

BD

Day: " 7" for 7th day of month.

W

Weekday: "0" for Sunday, "6" for Saturday.

WWW

Weekday: "SUN" for Sunday.

Www

Weekday: "Sun" for Sunday.

www

Weekday: "sun" for Sunday.

WWWW

Weekday: "SUNDAY" for Sunday.

Wwww

Weekday: "Sunday" for Sunday.

d

Day of year: "34" for Feb. 3.

ddd
: Day of year: "034" for Feb. 3.

bdd
: Day of year: " 34" for Feb. 3.

h
: Hour: "5" for 5 AM; "17" for 5 PM.

hh
: Hour: "05" for 5 AM; "17" for 5 PM.

bh
: Hour: " 5" for 5 AM; "17" for 5 PM.

H
: Hour: "5" for 5 AM and 5 PM.

HH
: Hour: "05" for 5 AM and 5 PM.

BH
: Hour: " 5" for 5 AM and 5 PM.

p
: AM/PM: "a" or "p".

P
: AM/PM: "A" or "P".

pp
: AM/PM: "am" or "pm".

PP
: AM/PM: "AM" or "PM".

pppp
: AM/PM: "a.m." or "p.m.".

PPPP
: AM/PM: "A.M." or "P.M.".

m
: Minutes: "7" for 7.

mm
: Minutes: "07" for 7.

bm
: Minutes: " 7" for 7.

s
: Seconds: "7" for 7; "7.23" for 7.23.

ss

Seconds: "07" for 7; "07.23" for 7.23.

bs

Seconds: " 7" for 7; " 7.23" for 7.23.

SS

Optional seconds: "07" for 7; blank for 0.

BS

Optional seconds: " 7" for 7; blank for 0.

N

Numeric date/time: "726842.25" for 6:00am Wed Jan 9, 1991.

n

Numeric date: "726842" for any time on Wed Jan 9, 1991.

J

Julian date/time: "2448265.75" for 6:00am Wed Jan 9, 1991.

j

Julian date: "2448266" for any time on Wed Jan 9, 1991.

U

Unix time: "663400800" for 6:00am Wed Jan 9, 1991.

X

Brackets suppression. An "X" at the front of the format causes the surrounding `< >' delimiters to be omitted when formatting dates. Note that the brackets are still required for algebraic entry.

If "SS" or "BS" (optional seconds) is preceded by a colon, the colon is also omitted if the seconds part is zero.

If "bb," "bbb" or "bbbb" or their upper-case equivalents appear in the format, then negative year numbers are displayed without a minus sign. Note that "aa" and "bb" are mutually exclusive. Some typical usages would be `YYYY AABB'; `AAAYYYYBBB'; `YYYYBBB'.

The formats "YY," "YYYY," "MM," "DD," "ddd," "hh," "HH," "mm," "ss," and "SS" actually match any number of digits during reading unless several of these codes are strung together with no punctuation in between, in which case the input must have exactly as many digits as there are letters in the format.

The "j," "J," and "U" formats do not make any time zone adjustment. They effectively use `julian(x,0)' and `unixtime(x,0)' to make the conversion; see section Date Arithmetic.

Free-Form Dates

When reading a date form during algebraic entry, Calc falls back on the algorithm described here if the input does not exactly match the current date format. This algorithm generally "does the right thing" and you don't have to worry about it, but it is described here in full detail for the curious.

Calc does not distinguish between upper- and lower-case letters while interpreting dates.

First, the time portion, if present, is located somewhere in the text and then removed. The remaining is then interpreted as the date.

A time is of the form `hh:mm:ss', possibly with the seconds part omitted and possibly with an AM/PM indicator added to indicate 12-hour time. If the AM/PM is present, the minutes may also be omitted. The AM/PM part may be any of the words `am', `pm', `noon', or `midnight'; each of these may be abbreviated to one letter, and the alternate forms `a.m.', `p.m.', and `mid' are also understood. Obviously `noon' and `midnight' are allowed only on 12:00:00. The words `noon', `mid', and `midnight' are also recognized with no number attached.

If there is no AM/PM indicator, the time is interpreted in 24-hour format.

To read the date portion, all words and numbers are isolated from the string; other characters are ignored. All words must be either month names or day-of-week names (the latter of which are ignored). Names can be written in full or as three-letter abbreviations.

Large numbers, or numbers with `+' or `-' signs, are interpreted as years. If one of the other numbers is greater than 12, then that must be the day and the remaining number in the input is therefore the month. Otherwise, Calc assumes the month, day and year are in the same order that they appear in the current date format. If the year is omitted, the current year is taken from the system clock.

If there are too many or too few numbers, or any unrecognizable words, then the input is rejected.

If there are any large numbers (of five digits or more) other than the year, they are ignored on the assumption that they are something like Julian dates that were included along with the traditional date components when the date was formatted.

One of the words `ad', `a.d.', `bc', or `b.c.' may optionally be used; the latter two are equivalent to a minus sign on the year value.

If you always enter a four-digit year, and use a name instead of a number for the month, there is no danger of ambiguity.

Standard Date Formats

There are actually ten standard date formats, numbered 0 through 9. Entering a blank line at the d d command's prompt gives you format number 1, Calc's usual format. You can enter any digit to select the other formats.

To create your own standard date formats, give a numeric prefix argument from 0 to 9 to the d d command. The format you enter will be recorded as the new standard format of that number, as well as becoming the new current date format. You can save your formats permanently with the m m command (see section Mode Settings).

0

`N' (Numerical format)

1

`<H:mm:SSpp >Www Mmm D, YYYY' (American format)

2

`D Mmm YYYY<, h:mm:SS>' (European format)

3

`Www Mmm BD< hh:mm:ss> YYYY' (Unix written date format)

4

`M/D/Y< H:mm:SSpp>' (American slashed format)

5

`D.M.Y< h:mm:SS>' (European dotted format)

6

`M-D-Y< H:mm:SSpp>' (American dashed format)

7

`D-M-Y< h:mm:SS>' (European dashed format)

8

`j<, h:mm:ss>' (Julian day plus time)

9

`YYddd< hh:mm:ss>' (Year-day format)

Truncating the Stack

The d t (`calc-truncate-stack`) command moves the `.' line that marks the top-of-stack up or down in the Calculator buffer. The number right above that line is considered to the be at the top of the stack. Any numbers below that line are "hidden" from all stack operations. This is similar to the Emacs "narrowing" feature, except that the values below the `.' are *visible*, just temporarily frozen. This feature allows you to keep several independent calculations running at once in different parts of the stack, or to apply a certain command to an element buried deep in the stack.

Pressing d t by itself moves the `.' to the line the cursor is on. Thus, this line and all those below it become hidden. To un-hide these lines, move down to the end of the buffer and press d t. With a positive numeric prefix argument n, d t hides the bottom n values in the buffer. With a negative argument, it hides all but the top n values. With an argument of zero, it hides zero values, i.e., moves the `.' all the way down to the bottom.

The d [(`calc-truncate-up`) and d] (`calc-truncate-down`) commands move the `.' up or down one line at a time (or several lines with a prefix argument).

Justification

Values on the stack are normally left-justified in the window. You can control this arrangement by typing d < (`calc-left-justify`), d > (`calc-right-justify`), or d = (`calc-center-justify`). For example, in right-justification mode, stack entries are displayed flush-right against the right edge of the window.

If you change the width of the Calculator window you may have to type d SPC (`calc-refresh`) to re-align right-justified or centered text.

Right-justification is especially useful together with fixed-point notation (see `d f`; `calc-fix-notation`). With these modes together, the decimal points on numbers will always line up.

With a numeric prefix argument, the justification commands give you a little extra control over the display. The argument specifies the horizontal "origin" of a display line. It is also possible to specify a maximum line width using the d b command (see section Normal Language Modes). For reference, the precise rules for formatting and breaking lines are given below. Notice that the interaction between origin and line width is slightly different in each justification mode.

In left-justified mode, the line is indented by a number of spaces given by the origin (default zero). If the result is longer than the maximum line width, if given, or too wide to fit in the Calc window otherwise, then it is broken into lines which will fit; each broken line is indented to the origin.

In right-justified mode, lines are shifted right so that the rightmost character is just before the origin, or just before the current window width if no origin was specified. If the line is too long for this, then it is broken; the current line width is used, if specified, or else the origin is used as a width if that is specified, or else the line is broken to fit in the window.

In centering mode, the origin is the column number of the center of each stack entry. If a line width is specified, lines will not be allowed to go past that width; Calc will either indent less or break the lines if necessary. If no origin is specified, half the line width or Calc window width is used.

Note that, in each case, if line numbering is enabled the display is indented an additional four spaces to make room for the line number. The width of the line number is taken into account when positioning according to the current Calc window width, but not when positioning by explicit origins and widths. In the latter case, the display is formatted as specified, and then uniformly shifted over four spaces to fit the line numbers.

Labels

The d { (`calc-left-label`) command prompts for a string, then displays that string to the left of every stack entry. If the entries are left-justified (see section Justification), then they will appear immediately after the label (unless you specified an origin greater than the length of the label). If the entries are centered or right-justified, the label appears on the far left and does not affect the horizontal position of the stack entry.

Give a blank string (with d { RET) to turn the label off.

The d } (`calc-right-label`) command similarly adds a label on the righthand side. It does not affect positioning of the stack entries unless they are right-justified. Also, if both a line width and an origin are given in right-justified mode, the stack entry is justified to the origin and the righthand label is justified to the line width.

One application of labels would be to add equation numbers to formulas you are manipulating in Calc and then copying into a document (possibly using Embedded Mode). The equations would typically be centered, and the equation numbers would be on the left or right as you prefer.

Language Modes

The commands in this section change Calc to use a different notation for entry and display of formulas, corresponding to the conventions of some other common language such as Pascal or TeX. Objects displayed on the stack or yanked from the Calculator to an editing buffer will be formatted in the current language; objects entered in algebraic entry or yanked from another buffer will be interpreted according to the current language.

The current language has no effect on things written to or read from the trail buffer, nor does it affect numeric entry. Only algebraic entry is affected. You can make even algebraic entry ignore the current language and use the standard notation by giving a numeric prefix, e.g., C-u '.

For example, suppose the formula `2*a[1] + atan(a[2])' occurs in a C program; elsewhere in the program you need the derivatives of this formula with respect to `a[1]' and `a[2]'. First, type d C to switch to C notation. Now use `C-u M-# g` to grab the formula into the Calculator, a d a[1] RET to differentiate with respect to the first variable, and M-# y to yank the formula for the derivative back into your C program. Press U to undo the differentiation and repeat with a d a[2] RET for the other derivative.

Without being switched into C mode first, Calc would have misinterpreted the brackets in `a[1]' and `a[2]', would not have known that `atan` was equivalent to Calc's built-in `arctan` function, and would have written the formula back with notations (like implicit multiplication) which would not have been legal for a C program.

As another example, suppose you are maintaining a C program and a TeX document, each of which needs a copy of the same formula. You can grab the formula from the program in C mode, switch to TeX mode, and yank the formula into the document in TeX math-mode format.

Language modes are selected by typing the letter d followed by a shifted letter key.

Normal Language Modes

The d N (`calc-normal-language`) command selects the usual notation for Calc formulas, as described in the rest of this manual. Matrices are displayed in a multi-line tabular format, but all other objects are written in linear form, as they would be typed from the keyboard.

The d O (`calc-flat-language`) command selects a language identical with the normal one, except that matrices are written in one-line form along with everything else. In some applications this form may be more suitable for yanking data into other buffers.

Even in one-line mode, long formulas or vectors will still be split across multiple lines if they exceed the width of the Calculator window. The d b (`calc-line-breaking`) command turns this line-breaking feature on and off. (It works independently of the current language.) If you give a numeric prefix argument of five or greater to the d b command, that argument will specify the line width used when breaking long lines.

The d B (`calc-big-language`) command selects a language which uses textual approximations to various mathematical notations, such as powers, quotients, and square roots:


```
  _____________
  | a + 1    2
  | ----- + c
\|    b
```

in place of `sqrt((a+1)/b + c^2)'.

Subscripts like `a_i' are displayed as actual subscripts in "big" mode. Double subscripts, `a_i_j' (`subscr(subscr(a, i), j)') are displayed as `a' with subscripts separated by commas: `i, j'. They must still be entered in the usual underscore notation.

One slight ambiguity of Big notation is that

```
  3
- -
  4
```

can represent either the negative rational number -3:4, or the actual expression `-(3/4)'; but the latter formula would normally never be displayed because it would immediately be evaluated to -3:4 or -0.75, so this ambiguity is not a problem in typical use.

Non-decimal numbers are displayed with subscripts. Thus there is no way to tell the difference between `16#C2' and `C2_16', though generally you will know which interpretation is correct. Logarithms `log(x,b)' and `log10(x)' also use subscripts in Big mode.

In Big mode, stack entries often take up several lines. To aid readability, stack entries are separated by a blank line in this mode. You may find it useful to expand the Calc window's height using C-x ^ (`enlarge-window`) or to make the Calc window the only one on the screen with C-x 1 (`delete-other-windows`).

Long lines are currently not rearranged to fit the window width in Big mode, so you may need to use the < and > keys to scroll across a wide formula. For really big formulas, you may even need to use { and } to scroll up and down.

The d U (`calc-unformatted-language`) command altogether disables the use of operator notation in formulas. In this mode, the formula shown above would be displayed:

```
sqrt(add(div(add(a, 1), b), pow(c, 2)))
```

These four modes differ only in display format, not in the format expected for algebraic entry. The standard Calc operators work in all four modes, and unformatted notation works in any language mode (except that Mathematica mode expects square brackets instead of parentheses).

C, FORTRAN, and Pascal Modes

The d C (`calc-c-language`) command selects the conventions of the C language for display and entry of formulas. This differs from the normal language mode in a variety of (mostly minor) ways. In particular, C language operators and operator precedences are used in place of Calc's usual ones. For example, `a^b' means `xor(a,b)' in C mode; a value raised to a power is written as a function call, `pow(a,b)'.

In C mode, vectors and matrices use curly braces instead of brackets. Octal and hexadecimal values are written with leading `0' or `0x' rather than using the `#' symbol. Array subscripting is translated into `subscr` calls, so that `a[i]' in C mode is the same as `a_i' in normal mode. Assignments turn into the `assign` function, which Calc normally displays using the `:=' symbol.

The variables `var-pi` and `var-e` would be displayed `pi' and `e' in normal mode, but in C mode they are displayed as `M_PI' and `M_E', corresponding to the names of constants typically provided in the `` `<math.h>' `` header. Functions whose names are different in C are translated automatically for entry and display purposes. For example, entering `asin(x)' will push the formula `arcsin(x)' onto the stack; this formula will be displayed as `asin(x)' as long as C mode is in effect.

The d P (`calc-pascal-language`) command selects Pascal conventions. Like C mode, Pascal mode interprets array brackets and uses a different table of operators. Hexadecimal numbers are entered and displayed with a preceding dollar sign. (Thus the regular meaning of $2 during algebraic entry does not work in Pascal mode, though $ (and $$, etc.) not followed by digits works the same as always.) No special provisions are made for other non-decimal numbers, vectors, and so on, since there is no universally accepted standard way of handling these in Pascal.

The d F (`calc-fortran-language`) command selects FORTRAN conventions. Various function names are transformed into FORTRAN equivalents. Vectors are written as `/1, 2, 3/', and may be entered this way or using square brackets. Since FORTRAN uses round parentheses for both function calls and array subscripts, Calc displays both in the same way; `a(i)' is interpreted as a function call upon reading, and subscripts must be entered as `subscr(a, i)'. Also, if the variable `a` has been declared to have type `vector` or `matrix` then `a(i)' will be parsed as a subscript. (See section Declarations.) Usually it doesn't matter, though; if you enter the subscript expression `a(i)' and Calc interprets it as a function call, you'll never know the difference unless you switch to another language mode or replace `a` with an actual vector (or unless `a` happens to be the name of a built-in function!).

Underscores are allowed in variable and function names in all of these language modes. The underscore here is equivalent to the `#' in normal mode, or to hyphens in the underlying Emacs Lisp variable names.

FORTRAN and Pascal modes normally do not adjust the case of letters in formulas. Most built-in Calc names use lower-case letters. If you use a positive numeric prefix argument with d P or d F, these modes will use upper-case letters exclusively for display, and will convert to lower-case on input. With a negative prefix, these modes convert to lower-case for display and input.

TeX Language Mode

The d T (`calc-tex-language`) command selects the conventions of "math mode" in the TeX typesetting language, by Donald Knuth. Formulas are entered and displayed in TeX notation, as in `\sin\left( a \over b \right)'. Math formulas are usually enclosed by `$ $' signs in TeX; these should be omitted when interfacing with Calc. To Calc, the `$' sign has the same meaning it always does in algebraic formulas (a reference to an existing entry on the stack).

Complex numbers are displayed as in `3 + 4i'. Fractions and quotients are written using `\over`; binomial coefficients are written with `\choose`. Interval forms are written with `\ldots`, and error forms are written with `\pm`. Absolute values are written as in `|x + 1|', and the floor and ceiling functions are written with `\lfloor`, `\rfloor`, etc. The words `\left` and `\right` are ignored when reading formulas in TeX mode. Both `inf` and `uinf` are written as `\infty`; when read, `\infty` always translates to `inf`.

Function calls are written the usual way, with the function name followed by the arguments in parentheses. However, functions for which TeX has special names (like `\sin`) will use curly braces instead of parentheses for very simple arguments. During input, curly braces and parentheses work equally well for grouping, but when the document is formatted the curly braces will be invisible. Thus the printed result is sin 2x but @c{$\sin(2 + x)$} sin(2 + x).

Function and variable names not treated specially by TeX are simply written out as-is, which will cause them to come out in italic letters in the printed document. If you invoke d T with a positive numeric prefix argument, names of more than one character will instead be written `\hbox{name}'. The `\hbox{ }' notation is ignored during reading. If you use a negative prefix argument, such function names are written `\name', and function names that begin with \ during reading have the \ removed. (Note that in this mode, long variable names are still written with `\hbox`. However, you can always make an actual variable name like `\bar` in any TeX mode.)

During reading, text of the form `\matrix{ ... }' is replaced by `[...]'. The same also applies to `\pmatrix` and `\bmatrix`. The symbol `&' is interpreted as a comma, and the symbols `\cr' and `\\' are interpreted as semicolons. During output, matrices are displayed in `\matrix{ a & b \\ c & d}' format; you may need to edit this afterwards to change `\matrix` to `\pmatrix` or `\\` to `\cr`.

Accents like `\tilde` and `\bar` translate into function calls internally (`tilde(x)', `bar(x)'). The `\underline` sequence is treated as an accent. The `\vec` accent corresponds to the function name `Vec`, because `vec` is the name of a built-in Calc function. The following table shows the accents in Calc, TeX, and eqn (described in the next section):

@begingroup @let@calcindexershow=@calcindexernoshow
@let@calcindexersh=@calcindexernoshow @endgroup

Calc	TeX	eqn
acute	\acute	
bar	\bar	bar
breve	\breve	


```
check     \check
dot       \dot          dot
dotdot    \ddot         dotdot
dyad                    dyad
grave     \grave
hat       \hat          hat
Prime                   prime
tilde     \tilde        tilde
under     \underline    under
Vec       \vec          vec
```

The `=>' (evaluates-to) operator appears as a `\to` symbol: `{a \to b}'. TeX defines `\to` as an alias for `\rightarrow`. However, if the `=>' is the top-level expression being formatted, a slightly different notation is used: `\evalto a \to b'. The `\evalto` word is ignored by Calc's input routines, and is undefined in TeX. You will typically want to include one of the following definitions at the top of a TeX file that uses `\evalto`:

```
\def\evalto{}
\def\evalto#1\to{}
```

The first definition formats evaluates-to operators in the usual way. The second causes only the b part to appear in the printed document; the a part and the arrow are hidden. Another definition you may wish to use is `\let\to=\Rightarrow' which causes `\to` to appear more like Calc's `=>' symbol. See section The Evaluates-To Operator, for a discussion of `evalto`.

The complete set of TeX control sequences that are ignored during reading is:

```
\hbox  \mbox  \text  \left  \right
\,  \>  \:  \;  \!  \quad  \qquad  \hfil  \hfill
\displaystyle  \textstyle  \dsize  \tsize
\scriptstyle  \scriptscriptstyle  \ssize  \ssize
\rm  \bf  \it  \sl  \roman  \bold  \italic  \slanted
\cal  \mit  \Cal  \Bbb  \frak  \goth
\evalto
```

Note that, because these symbols are ignored, reading a TeX formula into Calc and writing it back out may lose spacing and font information.

Also, the "discretionary multiplication sign" `\*' is read the same as `*'.

Here are some examples of how various Calc formulas are formatted in TeX:

```
sin(a^2 / b_i)
\sin\left( {a^2 \over b_i} \right)

[(3, 4), 3:4, 3 +/- 4, [3 .. inf)]
```


```
[3 + 4i, {3 \over 4}, 3 \pm 4, [3 \ldots \infty)]

[abs(a), abs(a / b), floor(a), ceil(a / b)]
[|a|, \left| a \over b \right|,
 \lfloor a \rfloor, \left\lceil a \over b \right\rceil]

[sin(a), sin(2 a), sin(2 + a), sin(a / b)]
[\sin{a}, \sin{2 a}, \sin(2 + a),
 \sin\left( {a \over b} \right)]
```

First with plain d T, then with C-u d T, then finally with C-u - d T (using the example definition `\def\foo#1{\tilde F(#1)}':

```
[f(a), foo(bar), sin(pi)]
[f(a), foo(bar), \sin{\pi}]
[f(a), \hbox{foo}(\hbox{bar}), \sin{\pi}]
[f(a), \foo{\hbox{bar}}, \sin{\pi}]
```

First with `\def\evalto{}', then with `\def\evalto#1\to{}':

```
2 + 3 => 5
\evalto 2 + 3 \to 5
```

First with standard `\to`, then with `\let\to\Rightarrow':

```
[2 + 3 => 5, a / 2 => (b + c) / 2]
[{2 + 3 \to 5}, {{a \over 2} \to {b + c \over 2}}]
```

Matrices normally, then changing `\matrix` to `\pmatrix`:

```
[ [ a / b, 0 ], [ 0, 2^(x + 1) ] ]
\matrix{ {a \over b} & 0 \\ 0 & 2^{(x + 1)} }
\pmatrix{ {a \over b} & 0 \\ 0 & 2^{(x + 1)} }
```

Eqn Language Mode

Eqn is another popular formatter for math formulas. It is designed for use with the TROFF text formatter, and comes standard with many versions of Unix. The d E (`calc-eqn-language`) command selects eqn notation.

The eqn language's main idiosyncrasy is that whitespace plays a significant part in the parsing of the language. For example, `sqrt x+1 + y' treats `x+1' as the argument of the `sqrt` operator. Eqn also

understands more conventional grouping using curly braces: `sqrt{x+1} + y'. Braces are required only when the argument contains spaces.

In Calc's eqn mode, however, curly braces are required to delimit arguments of operators like `sqrt`. The first of the above examples would treat only the `x' as the argument of `sqrt`, and in fact `sin x+1' would be interpreted as `sin * x + 1', because `sin` is not a special operator in the eqn language. If you always surround the argument with curly braces, Calc will never misunderstand.

Calc also understands parentheses as grouping characters. Another peculiarity of eqn's syntax makes it advisable to separate words with spaces from any surrounding characters that aren't curly braces, so Calc writes `sin ( x + y )' in eqn mode. (The spaces around `sin` are important to make eqn recognize that `sin` should be typeset in a roman font, and the spaces around `x` and `y` are a good idea just in case the eqn document has defined special meanings for these names, too.)

Powers and subscripts are written with the `sub` and `sup` operators, respectively. Note that the caret symbol `^' is treated the same as a space in eqn mode, as is the `~' symbol (these are used to introduce spaces of various widths into the typeset output of eqn).

As in TeX mode, Calc's formatter omits parentheses around the arguments of functions like `ln` and `sin` if they are "simple-looking"; in this case Calc surrounds the argument with braces, separated by a `~' from the function name: `sin~{x}'.

Font change codes (like `roman x') and positioning codes (like `~' and `down n x') are ignored by the eqn reader. Also ignored are the words `left`, `right`, `mark`, and `lineup`. Quotation marks in eqn mode input are treated the same as curly braces: `sqrt "1+x"' is equivalent to `sqrt {1+x}'; this is only an approximation to the true meaning of quotes in eqn, but it is good enough for most uses.

Accent codes (`x dot') are handled by treating them as function calls (`dot(x)') internally. See section TeX Language Mode for a table of these accent functions. The `prime` accent is treated specially if it occurs on a variable or function name: `f prime prime ( x prime )' is stored internally as `f''(x')'. For example, taking the derivative of `f(2 x)' with a d x will produce `2 f'(2 x)', which eqn mode will display as `2 f prime (2 x)'.

Assignments are written with the `<-' (left-arrow) symbol, and `evalto` operators are written with `->' or `evalto ... ->' (see section TeX Language Mode, for a discussion of this). The regular Calc symbols `:=' and `=>' are also recognized for these operators during reading.

Vectors in eqn mode use regular Calc square brackets, but matrices are formatted as `matrix { ccol { a above b } ... }'. The words `lcol` and `rcol` are recognized as synonyms for `ccol` during input, and are generated instead of `ccol` if the matrix justification mode so specifies.

Mathematica Language Mode

The d M (`calc-mathematica-language`) command selects the conventions of Mathematica, a powerful and popular mathematical tool from Wolfram Research, Inc. Notable differences in Mathematica mode are that the names of built-in functions are capitalized, and function calls use square brackets instead of parentheses. Thus the Calc formula `sin(2 x)' is entered and displayed `Sin[2 x]' in Mathematica mode.

Vectors and matrices use curly braces in Mathematica. Complex numbers are written `3 + 4 I'. The standard special constants in Calc are written `Pi`, `E`, `I`, `GoldenRatio`, `EulerGamma`, `Infinity`, `ComplexInfinity`, and `Indeterminate` in Mathematica mode. Non-decimal numbers are written, e.g., `16^^7fff'. Floating-point numbers in scientific notation are written `1.23*10.^3'. Subscripts use double square brackets: `a[[i]]'.

Maple Language Mode

The d W (`calc-maple-language`) command selects the conventions of Maple, another mathematical tool from the University of Waterloo.

Maple's language is much like C. Underscores are allowed in symbol names; square brackets are used for subscripts; explicit `*'s for multiplications are required. Use either `^' or `**' to denote powers.

Maple uses square brackets for lists and curly braces for sets. Calc interprets both notations as vectors, and displays vectors with square brackets. This means Maple sets will be converted to lists when they pass through Calc. As a special case, matrices are written as calls to the function `matrix`, given a list of lists as the argument, and can be read in this form or with all-capitals `MATRIX`.

The Maple interval notation `2 .. 3' has no surrounding brackets; Calc reads `2 .. 3' as the closed interval `[2 .. 3]', and writes any kind of interval as `2 .. 3'. This means you cannot see the difference between an open and a closed interval while in Maple display mode.

Maple writes complex numbers as `3 + 4*I'. Its special constants are `Pi`, `E`, `I`, and `infinity` (all three of `inf`, `uinf`, and `nan` display as `infinity`). Floating-point numbers are written `1.23*10.^3'.

Among things not currently handled by Calc's Maple mode are the various quote symbols, procedures and functional operators, and inert (`&') operators.

Compositions

There are several composition functions which allow you to get displays in a variety of formats similar to those in Big language mode. Most of these functions do not evaluate to anything; they are placeholders which are left in symbolic form by Calc's evaluator but are recognized by Calc's display formatting routines.

Two of these, `string` and `bstring`, are described elsewhere. See section Strings. For example, `string("ABC")' is displayed as `ABC'. When viewed on the stack it will be indistinguishable from the variable `ABC`, but internally it will be stored as `string([65, 66, 67])' and can still be manipulated this way; for example, the selection and vector commands j 1 v v j u would select the vector portion of this object and reverse the elements, then deselect to reveal a string whose characters had been reversed.

The composition functions do the same thing in all language modes (although their components will of course be formatted in the current language mode). The one exception is Unformatted mode (d U), which does not give the composition functions any special treatment. The functions are discussed here because of their relationship to the language modes.

Composition Basics

Compositions are generally formed by stacking formulas together horizontally or vertically in various ways. Those formulas are themselves compositions. TeX users will find this analogous to TeX's "boxes." Each multi-line composition has a baseline; horizontal compositions use the baselines to decide how formulas should be positioned relative to one another. For example, in the Big mode formula

```
           2
      a + b
17 + ------
       c
```

the second term of the sum is four lines tall and has line three as its baseline. Thus when the term is combined with 17, line three is placed on the same level as the baseline of 17.

Another important composition concept is precedence. This is an integer that represents the binding strength of various operators. For example, `*' has higher precedence (195) than `+' (180), which means that `(a * b) + c' will be formatted without the parentheses, but `a * (b + c)' will keep the parentheses.

The operator table used by normal and Big language modes has the following precedences:

```
_       1200   (subscripts)
%       1100   (as in n%)
-       1000   (as in -n)
!       1000   (as in !n)
mod      400
+/-      300
!!       210    (as in n!!)
!        210    (as in n!)
^        200
*        195    (or implicit multiplication)
/ % \    190
+ -      180    (as in a+b)
|        170
< =      160    (and other relations)
&&       110
||       100
? :       90
!!!       85
&&&       80
|||       75
:=        50
::        45
=>        40
```

The general rule is that if an operator with precedence n occurs as an argument to an operator with precedence m, then the argument is enclosed in parentheses if n < m. Top-level expressions and

expressions which are function arguments, vector components, etc., are formatted with precedence z
(so that they normally never get additional parentheses).

For binary left-associative operators like `+', the righthand argument is actually formatted with one-higher precedence than shown in the table. This makes sure `(a + b) + c' omits the parentheses, but the unnatural form `a + (b + c)' keeps its parentheses. Right-associative operators like `^' format the lefthand argument with one-higher precedence.

The `cprec` function formats an expression with an arbitrary precedence. For example, `cprec(abc, 185)' will combine into sums and products as follows: `7 + abc', `7 (abc)' (because this `cprec` form has higher precedence than addition, but lower precedence than multiplication).

A final composition issue is line breaking. Calc uses two different strategies for "flat" and "non-flat" compositions. A non-flat composition is anything that appears on multiple lines (not counting line breaking). Examples would be matrices and Big mode powers and quotients. Non-flat compositions are displayed exactly as specified. If they come out wider than the current window, you must use horizontal scrolling (< and >) to view them.

Flat compositions, on the other hand, will be broken across several lines if they are too wide to fit the window. Certain points in a composition are noted internally as break points. Calc's general strategy is to fill each line as much as possible, then to move down to the next line starting at the first break point that didn't fit. However, the line breaker understands the hierarchical structure of formulas. It will not break an "inner" formula if it can use an earlier break point from an "outer" formula instead. For example, a vector of sums might be formatted as:

```
[ a + b + c, d + e + f,
  g + h + i, j + k + l, m ]
```

If the `m' can fit, then so, it seems, could the `g'. But Calc prefers to break at the comma since the comma is part of a "more outer" formula. Calc would break at a plus sign only if it had to, say, if the very first sum in the vector had itself been too large to fit.

Of the composition functions described below, only `choriz` generates break points. The `bstring` function (see section Strings) also generates breakable items: A break point is added after every space (or group of spaces) except for spaces at the very beginning or end of the string.

Composition functions themselves count as levels in the formula hierarchy, so a `choriz` that is a component of a larger `choriz` will be less likely to be broken. As a special case, if a `bstring` occurs as a component of a `choriz` or `choriz`-like object (such as a vector or a list of arguments in a function call), then the break points in that `bstring` will be on the same level as the break points of the surrounding object.

Horizontal Compositions

The `choriz` function takes a vector of objects and composes them horizontally. For example, `choriz([17, a b/c, d])' formats as `17a b / cd' in normal language mode, or as

```
a b
```


```
17--d
   c
```

in Big language mode. This is actually one case of the general function `choriz(vec, sep, prec)', where either or both of sep and prec may be omitted. Prec gives the precedence to use when formatting each of the components of vec. The default precedence is the precedence from the surrounding environment.

Sep is a string (i.e., a vector of character codes as might be entered with `" "` notation) which should separate components of the composition. Also, if sep is given, the line breaker will allow lines to be broken after each occurrence of sep. If sep is omitted, the composition will not be breakable (unless any of its component compositions are breakable).

For example, `2 choriz([a, b c, d = e], " + ", 180)' is formatted as `2 a + b c + (d = e)'. To get the `choriz` to have precedence 180 "outwards" as well as "inwards," enclose it in a `cprec` form: `2 cprec(choriz(...), 180)' formats as `2 (a + b c + (d = e))'.

The baseline of a horizontal composition is the same as the baselines of the component compositions, which are all aligned.

Vertical Compositions

The `cvert` function makes a vertical composition. Each component of the vector is centered in a column. The baseline of the result is by default the top line of the resulting composition. For example, `f(cvert([a, bb, ccc]), cvert([a^2 + 1, b^2]))' formats in Big mode as

```
f( a ,  2    )
  bb   a  + 1
  ccc     2
         b
```

There are several special composition functions that work only as components of a vertical composition. The `cbase` function controls the baseline of the vertical composition; the baseline will be the same as the baseline of whatever component is enclosed in `cbase`. Thus `f(cvert([a, cbase(bb), ccc]), cvert([a^2 + 1, cbase(b^2)]))' displays as

```
         2
       a  + 1
   a       2
f(bb ,   b   )
  ccc
```

There are also `ctbase` and `cbbase` functions which make the baseline of the vertical composition equal to the top or bottom line (rather than the baseline) of that component. Thus `cvert([cbase(a / b)]) + cvert([ctbase(a / b)]) + cvert([cbbase(a / b)])' gives

```
        a
a       -
```


```
- + a + b
b   -
    b
```

There should be only one `cbase`, `ctbase`, or `cbbase` function in a given vertical composition. These functions can also be written with no arguments: `ctbase()' is a zero-height object which means the baseline is the top line of the following item, and `cbbase()' means the baseline is the bottom line of the preceding item.

The `crule` function builds a "rule," or horizontal line, across a vertical composition. By itself `crule()' uses `-' characters to build the rule. You can specify any other character, e.g., `crule("=")'. The argument must be a character code or vector of exactly one character code. It is repeated to match the width of the widest item in the stack. For example, a quotient with a thick line is `cvert([a + 1, cbase(crule("=")), b^2])':

```
a + 1
=====
  2
 b
```

Finally, the functions `clvert` and `crvert` act exactly like `cvert` except that the items are left- or right-justified in the stack. Thus `clvert([a, bb, ccc]) + crvert([a, bb, ccc])' gives:

```
a   +   a
bb     bb
ccc   ccc
```

Like `choriz`, the vertical compositions accept a second argument which gives the precedence to use when formatting the components. Vertical compositions do not support separator strings.

Other Compositions

The `csup` function builds a superscripted expression. For example, `csup(a, b)' looks the same as `a^b' does in Big language mode. This is essentially a horizontal composition of `a' and `b', where `b' is shifted up so that its bottom line is one above the baseline.

Likewise, the `csub` function builds a subscripted expression. This shifts `b' down so that its top line is one below the bottom line of `a' (note that this is not quite analogous to `csup`). Other arrangements can be obtained by using `choriz` and `cvert` directly.

The `cflat` function formats its argument in "flat" mode, as obtained by `d O', if the current language mode is normal or Big. It has no effect in other language modes. For example, `a^(b/c)' is formatted by Big mode like `csup(a, cflat(b/c))' to improve its readability.

The `cspace` function creates horizontal space. For example, `cspace(4)' is effectively the same as `string("   ")'. A second string (i.e., vector of characters) argument is repeated instead of the space character. For example, `cspace(4, "ab")' looks like `abababab'. If the second argument is not a string, it is formatted in the normal way and then several copies of that are composed together: `cspace(4, a^2)'

yields

```
 2 2 2 2
a a a a
```

If the number argument is zero, this is a zero-width object.

The `cvspace` function creates vertical space, or a vertical stack of copies of a certain string or formatted object. The baseline is the center line of the resulting stack. A numerical argument of zero will produce an object which contributes zero height if used in a vertical composition.

There are also `ctspace` and `cbspace` functions which create vertical space with the baseline the same as the baseline of the top or bottom copy, respectively, of the second argument. Thus `cvspace(2, a/b) + ctspace(2, a/b) + cbspace(2, a/b)' displays as:

```
          a
          -
a         b
-    a    a
b + - + -
a    b    b
-    a
b    -
     b
```

Information about Compositions

The functions in this section are actual functions; they compose their arguments according to the current language and other display modes, then return a certain measurement of the composition as an integer.

The `cwidth` function measures the width, in characters, of a composition. For example, `cwidth(a + b)' is 5, and `cwidth(a / b)' is 5 in normal mode, 1 in Big mode, and 11 in TeX mode (for `{a \over b}'). The argument may involve the composition functions described in this section.

The `cheight` function measures the height of a composition. This is the total number of lines in the argument's printed form.

The functions `cascent` and `cdescent` measure the amount of the height that is above (and including) the baseline, or below the baseline, respectively. Thus `cascent(x) + cdescent(x)' always equals `cheight(x)'. For a one-line formula like `a + b', `cascent` returns 1 and `cdescent` returns 0. For `a / b' in Big mode, `cascent` returns 2 and `cdescent` returns 1. The only formula for which `cascent` will return zero is `cvspace(0)' or equivalents.

User-Defined Compositions

The Z C (`calc-user-define-composition`) command lets you define the display format for any algebraic function. You provide a formula containing a certain number of argument variables on the

stack. Any time Calc formats a call to the specified function in the current language mode and with the number of arguments, Calc effectively replaces the function call with that formula with the arguments replaced.

Calc builds the default argument list by sorting all the variable names that appear in the formula into alphabetical order. You can edit this argument list before pressing RET if you wish. Any variables in the formula that do not appear in the argument list will be displayed literally; any arguments that do not appear in the formula will not affect the display at all.

You can define formats for built-in functions, for functions you have defined with Z F (see section Programming with Formulas), or for functions which have no definitions but are being used as purely syntactic objects. You can define different formats for each language mode, and for each number of arguments, using a succession of Z C commands. When Calc formats a function call, it first searches for a format defined for the current language mode (and number of arguments); if there is none, it uses the format defined for the Normal language mode. If neither format exists, Calc uses its built-in standard format for that function (usually just `func(args)').

If you execute Z C with the number 0 on the stack instead of a formula, any defined formats for the function in the current language mode will be removed. The function will revert to its standard format.

For example, the default format for the binomial coefficient function `choose(n, m)' in the Big language mode is

```
 n
( )
 m
```

You might prefer the notation,

```
 C
n m
```

To define this notation, first make sure you are in Big mode, then put the formula

```
choriz([cvert([cvspace(1), n]), C, cvert([cvspace(1), m])])
```

on the stack and type Z C. Answer the first prompt with `choose`. The second prompt will be the default argument list of `(C m n)'. Edit this list to be `(n m)' and press RET. Now, try it out: For example, turn simplification off with m O and enter `choose(a,b) + choose(7,3)' as an algebraic entry.

```
 C  +  C
a b   7 3
```

As another example, let's define the usual notation for Stirling numbers of the first kind, `stir1(n, m)'. This is just like the regular format for binomial coefficients but with square brackets instead of parentheses.


```
choriz([string("["), cvert([n, cbase(cvspace(1)), m]), string("]")]
```

Now type Z C stir1 RET, edit the argument list to `(n m)', and type RET.

The formula provided to Z C usually will involve composition functions, but it doesn't have to. Putting the formula `a + b + c' onto the stack and typing Z C foo RET RET would define the function `foo(x,y,z)' to display like `x + y + z'. This "sum" will act exactly like a real sum for all formatting purposes (it will be parenthesized the same, and so on). However it will be computationally unrelated to a sum. For example, the formula `2 * foo(1, 2, 3)' will display as `2 (1 + 2 + 3)'. Operator precedences have caused the "sum" to be written in parentheses, but the arguments have not actually been summed. (Generally a display format like this would be undesirable, since it can easily be confused with a real sum.)

The special function `eval` can be used inside a Z C composition formula to cause all or part of the formula to be evaluated at display time. For example, if the formula is `a + eval(b + c)', then `foo(1, 2, 3)' will be displayed as `1 + 5'. Evaluation will use the default simplifications, regardless of the current simplification mode. There are also `evalsimp` and `evalextsimp` which simplify as if by a s and a e (respectively). Note that these "functions" operate only in the context of composition formulas (and also in rewrite rules, where they serve a similar purpose; see section Rewrite Rules). On the stack, a call to `eval` will be left in symbolic form.

It is not a good idea to use `eval` except as a last resort. It can cause the display of formulas to be extremely slow. For example, while `eval(a + b)' might seem quite fast and simple, there are several situations where it could be slow. For example, `a' and/or `b' could be polar complex numbers, in which case doing the sum requires trigonometry. Or, `a' could be the factorial `fact(100)' which is unevaluated because you have typed m O; `eval` will evaluate it anyway to produce a large, unwieldy integer.

You can save your display formats permanently using the Z P command (see section Creating User Keys).

Syntax Tables

Syntax tables do for input what compositions do for output: They allow you to teach custom notations to Calc's formula parser. Calc keeps a separate syntax table for each language mode.

(Note that the Calc "syntax tables" discussed here are completely unrelated to the syntax tables described in the Emacs manual.)

The Z S (`calc-edit-user-syntax`) command edits the syntax table for the current language mode. If you want your syntax to work in any language, define it in the normal language mode. Type M-# M-# to finish editing the syntax table, or M-# x to cancel the edit. The m m command saves all the syntax tables along with the other mode settings; see section General Mode Commands.

Syntax Table Basics

Parsing is the process of converting a raw string of characters, such as you would type in during algebraic entry, into a Calc formula. Calc's parser works in two stages. First, the input is broken down into tokens, such as words, numbers, and punctuation symbols like `+', `:=', and `+/-'. Space between tokens is ignored (except when it serves to separate adjacent words). Next, the parser matches this string

of tokens against various built-in syntactic patterns, such as "an expression followed by `+' followed another expression" or "a name followed by `(', zero or more expressions separated by commas, and

A syntax table is a list of user-defined syntax rules, which allow you to specify new patterns to define your own favorite input notations. Calc's parser always checks the syntax table for the current language mode, then the table for the normal language mode, before it uses its built-in rules to parse an algebraic formula you have entered. Each syntax rule should go on its own line; it consists of a pattern, a `:=' symbol, and a Calc formula with an optional condition. (Syntax rules resemble algebraic rewrite rules, but the notation for patterns is completely different.)

A syntax pattern is a list of tokens, separated by spaces. Except for a few special symbols, tokens in syntax patterns are matched literally, from left to right. For example, the rule,

```
foo ( ) := 2+3
```

would cause Calc to parse the formula `4+foo()*5' as if it were `4+(2+3)*5'. Notice that the parentheses were written as two separate tokens in the rule. As a result, the rule works for both `foo()' and `foo ( )'. If we had written the rule as `foo () := 2+3', then Calc would treat `()' as a single, indivisible token, so that `foo( )' would not be recognized by the rule. (It would be parsed as a regular zero-argument function call instead.) In fact, this rule would also make trouble for the rest of Calc's parser: An unrelated formula like `bar()' would now be tokenized into `bar ()' instead of `bar ( )', so that the standard parser for function calls would no longer recognize it!

While it is possible to make a token with a mixture of letters and punctuation symbols, this is not recommended. It is better to break it into several tokens, as we did with `foo()' above.

The symbol `#' in a syntax pattern matches any Calc expression. On the righthand side, the things that matched the `#'s can be referred to as `#1', `#2', and so on (where `#1' matches the leftmost `#' in the pattern). For example, these rules match a user-defined function, prefix operator, infix operator, and postfix operator, respectively:

```
foo ( # ) := myfunc(#1)
foo # := myprefix(#1)
# foo # := myinfix(#1,#2)
# foo := mypostfix(#1)
```

Thus `foo(3)' will parse as `myfunc(3)', and `2+3 foo' will parse as `mypostfix(2+3)'.

It is important to write the first two rules in the order shown, because Calc tries rules in order from first to last. If the pattern `foo #' came first, it would match anything that could match the `foo (#)' rule, since an expression in parentheses is itself a valid expression. Thus the `foo ( # )' rule would never get to match anything. Likewise, the last two rules must be written in the order shown or else `3 foo 4' will be parsed as `mypostfix(3) * 4'. (Of course, the best way to avoid these ambiguities is not to use the same symbol in more than one way at the same time! In case you're not convinced, try the following exercise: How will the above rules parse the input `foo(3,4)', if at all? Work it out for yourself, then try it in Calc and see.)

Calc is quite flexible about what sorts of patterns are allowed. The only rule is that every pattern must

begin with a literal token (like `foo' in the first two patterns above), or with a `#' followed by a literal token (as in the last two patterns). After that, any mixture is allowed, although putting two `#'s in a r will not be very useful since two expressions with nothing between them will be parsed as one expression that uses implicit multiplication.

As a more practical example, Maple uses the notation `sum(a(i), i=1..10)' for sums, which Calc's Maple mode doesn't recognize at present. To handle this syntax, we simply add the rule,

```
sum ( # , # = # .. # ) := sum(#1,#2,#3,#4)
```

to the Maple mode syntax table. As another example, C mode can't read assignment operators like `++' and `*='. We can define these operators quite easily:

```
# *= # := muleq(#1,#2)
# ++ := postinc(#1)
++ # := preinc(#1)
```

To complete the job, we would use corresponding composition functions and Z C to cause these functions to display in their respective Maple and C notations. (Note that the C example ignores issues of operator precedence, which are discussed in the next section.)

You can enclose any token in quotes to prevent its usual interpretation in syntax patterns:

```
# ":=" # := becomes(#1,#2)
```

Quotes also allow you to include spaces in a token, although once again it is generally better to use two tokens than one token with an embedded space. To include an actual quotation mark in a quoted token, precede it with a backslash. (This also works to include backslashes in tokens.)

```
# "bad token" # "/\"\\" # := silly(#1,#2,#3)
```

This will parse `3 bad token 4 /"\ 5' to `silly(3,4,5)'.

The token # has a predefined meaning in Calc's formula parser; it is not legal to use `"#"' in a syntax rule. However, longer tokens that include the `#' character are allowed. Also, while `"$"' and `"\""' are allowed as tokens, their presence in the syntax table will prevent those characters from working in their usual ways (referring to stack entries and quoting strings, respectively).

Finally, the notation `%%' anywhere in a syntax table causes the rest of the line to be ignored as a comment.

Precedence

Different operators are generally assigned different precedences. By default, an operator defined by a rule like

```
# foo # := foo(#1,#2)
```

will have an extremely low precedence, so that `2*3+4 foo 5 == 6' will be parsed as `(2*3+4) foo (5 == 6)'. To change the precedence of an operator, use the notation `#/p' in place of `#', where p is an integer precedence level. For example, 185 lies between the precedences for `+' and `*', so if we change this rule to

```
#/185 foo #/186 := foo(#1,#2)
```

then `2+3 foo 4*5' will be parsed as `2+(3 foo (4*5))'. Also, because we've given the righthand expression slightly higher precedence, our new operator will be left-associative: `1 foo 2 foo 3' will be parsed as `(1 foo 2) foo 3'. By raising the precedence of the lefthand expression instead, we can create a right-associative operator.

See section Composition Basics, for a table of precedences of the standard Calc operators. For the precedences of operators in other language modes, look in the Calc source file `calc-lang.el'.

Advanced Syntax Patterns

To match a function with a variable number of arguments, you could write

```
foo ( # ) := myfunc(#1)
foo ( # , # ) := myfunc(#1,#2)
foo ( # , # , # ) := myfunc(#1,#2,#3)
```

but this isn't very elegant. To match variable numbers of items, Calc uses some notations inspired regular expressions and the "extended BNF" style used by some language designers.

```
foo ( { # }*, ) := apply(myfunc,#1)
```

The token `{' introduces a repeated or optional portion. One of the three tokens `}*', `}+', or `}?' ends the portion. These will match zero or more, one or more, or zero or one copies of the enclosed pattern, respectively. In addition, `}*' and `}+' can be followed by a separator token (with no space in between, as shown above). Thus `{ # }*,' matches nothing, or one expression, or several expressions separated by commas.

A complete `{ ... }' item matches as a vector of the items that matched inside it. For example, the above rule will match `foo(1,2,3)' to get `apply(myfunc,[1,2,3])'. The Calc `apply` function takes a function name and a vector of arguments and builds a call to the function with those arguments, so the net result is the formula `myfunc(1,2,3)'.

If the body of a `{ ... }' contains several `#'s (or nested `{ ... }' constructs), then the items will be strung together into the resulting vector. If the body does not contain anything but literal tokens, the result will always be an empty vector.

```
foo ( { # , # }+, ) := bar(#1)
foo ( { { # }*, }*; ) := matrix(#1)
```

will parse `foo(1,2,3,4)' as `bar([1,2,3,4])', and `foo(1,2;3,4)' as `matrix([[1,2],[3,4]])'. Also, after some

thought it's easy to see how this pair of rules will parse `foo(1,2,3)' as `matrix([[1,2,3]])', since the first rule will only match an even number of arguments. The rule

```
foo ( # { , # , # }? ) := bar(#1,#2)
```

will parse `foo(2,3,4)' as `bar(2,[3,4])', and `foo(2)' as `bar(2,[])'.

The notation `{ ... }?.' (note the trailing period) works just the same as regular `{ ... }?', except that it does not count as an argument; the following two rules are equivalent:

```
foo ( # , { also }? # ) := bar(#1,#3)
foo ( # , { also }?. # ) := bar(#1,#2)
```

Note that in the first case the optional text counts as `#2', which will always be an empty vector, but in the second case no empty vector is produced.

Another variant is `{ ... }?$', which means the body is optional only at the end of the input formula. All built-in syntax rules in Calc use this for closing delimiters, so that during algebraic entry you can type [sqrt(2), sqrt(3 RET, omitting the closing parenthesis and bracket. Calc does this automatically for trailing `)', `]', and `>' tokens in syntax rules, but you can use `{ ... }?$' explicitly to get this effect with any token (such as `"}"' or `end'). Like `{ ... }?.', this notation does not count as an argument. Conversely, you can use quotes, as in `")"', to prevent a closing-delimiter token from being automatically treated as optional.

Calc's parser does not have full backtracking, which means some patterns will not work as you might expect:

```
foo ( { # , }? # , # ) := bar(#1,#2,#3)
```

Here we are trying to make the first argument optional, so that `foo(2,3)' parses as `bar([],2,3)'. Unfortunately, Calc first tries to match `2,' against the optional part of the pattern, finds a match, and so goes ahead to match the rest of the pattern. Later on it will fail to match the second comma, but it doesn't know how to go back and try the other alternative at that point. One way to get around this would be to use two rules:

```
foo ( # , # , # ) := bar([#1],#2,#3)
foo ( # , # ) := bar([],#1,#2)
```

More precisely, when Calc wants to match an optional or repeated part of a pattern, it scans forward attempting to match that part. If it reaches the end of the optional part without failing, it "finalizes" its choice and proceeds. If it fails, though, it backs up and tries the other alternative. Thus Calc has "partial" backtracking. A fully backtracking parser would go on to make sure the rest of the pattern matched before finalizing the choice.

Conditional Syntax Rules

It is possible to attach a condition to a syntax rule. For example, the rules


```
foo ( # ) := ifoo(#1) :: integer(#1)
foo ( # ) := gfoo(#1)
```

will parse `foo(3)' as `ifoo(3)', but will parse `foo(3.5)' and `foo(x)' as calls to `gfoo'. Any number of conditions may be attached; all must be true for the rule to succeed. A condition is "true" if it evaluates to a nonzero number. See section Logical Operations, for a list of Calc functions like `integer' that perform logical tests.

The exact sequence of events is as follows: When Calc tries a rule, it first matches the pattern as usual. It then substitutes `#1', `#2', etc., in the conditions, if any. Next, the conditions are simplified and evaluated in order from left to right, as if by the a s algebra command (see section Simplifying Formulas). Each result is true if it is a nonzero number, or an expression that can be proven to be nonzero (see section Declarations). If the results of all conditions are true, the expression (such as `ifoo(#1)') has its `#'s substituted, and that is the result of the parse. If the result of any condition is false, Calc goes on to try the next rule in the syntax table.

Syntax rules also support `let' conditions, which operate in exactly the same way as they do in algebraic rewrite rules. See section Other Features of Rewrite Rules, for details. A `let' condition is always true, but as a side effect it defines a variable which can be used in later conditions, and also in the expression after the `:=' sign:

```
foo ( # ) := hifoo(x) :: let(x := #1 + 0.5) :: dnumint(x)
```

The `dnumint' function tests if a value is numerically an integer, i.e., either a true integer or an integer-valued float. This rule will parse `foo' with a half-integer argument, like `foo(3.5)', to a call like `hifoo(4.)'.

The lefthand side of a syntax rule `let' must be a simple variable, not the arbitrary pattern that is allowed in rewrite rules.

The `matches' function is also treated specially in syntax rule conditions (again, in the same way as in rewrite rules). See section Matching Commands. If the matching pattern contains meta-variables, then those meta-variables may be used in later conditions and in the result expression. The arguments to `matches' are not evaluated in this situation.

```
sum ( # , # ) := sum(#1,a,b,c) :: matches(#2, a=[b..c])
```

This is another way to implement the Maple mode `sum' notation. In this approach, we allow `#2' to equal the whole expression `i=1..10'. Then, we use `matches' to break it apart into its components. If the expression turns out not to match the pattern, the syntax rule will fail. Note that Z S always uses Calc's normal language mode for editing expressions in syntax rules, so we must use regular Calc notation for the interval `[b..c]' that will correspond to the Maple mode interval `1..10'.

The Modes Variable

The m g (`calc-get-modes`) command pushes onto the stack a vector of numbers that describes the various mode settings that are in effect. With a numeric prefix argument, it pushes only the nth mode, i.e., the nth element of this vector. Keyboard macros can use the m g command to modify their behavior based on the current mode settings.

The modes vector is also available in the special variable `Modes`. In other words, m g is like s r Modes RET. It will not work to store into this variable; in fact, if you do, `Modes` will cease to track the current modes. (The m g command will continue to work, however.)

In general, each number in this vector is suitable as a numeric prefix argument to the associated mode-setting command. (Recall that the ~ key takes a number from the stack and gives it as a numeric prefix to the next command.)

The elements of the modes vector are as follows:

1. Current precision. Default is 12; associated command is p.
2. Binary word size. Default is 32; associated command is b w.
3. Stack size (not counting the value about to be pushed by m g). This is zero if m g is executed with an empty stack.
4. Number radix. Default is 10; command is d r.
5. Floating-point format. This is the number of digits, plus the constant 0 for normal notation, 10000 for scientific notation, 20000 for engineering notation, or 30000 for fixed-point notation. These codes are acceptable as prefix arguments to the d n command, but note that this may lose information: For example, d s and C-u 12 d s have similar (but not quite identical) effects if the current precision is 12, but they both produce a code of 10012, which will be treated by d n as C-u 12 d s. If the precision then changes, the float format will still be frozen at 12 significant figures.
6. Angular mode. Default is 1 (degrees). Other values are 2 (radians) and 3 (HMS). The m d command accepts these prefixes.
7. Symbolic mode. Value is 0 or 1; default is 0. Command is m s.
8. Fraction mode. Value is 0 or 1; default is 0. Command is m f.
9. Polar mode. Value is 0 (rectangular) or 1 (polar); default is 0. Command is m p.
10. Matrix/scalar mode. Default value is *-1*. Value is 0 for scalar mode, *-2* for matrix mode, or *N* for @c{$N\times N$} *NxN* matrix mode. Command is m v.
11. Simplification mode. Default is 1. Value is *-1* for off (m O), 0 for m N, 2 for m B, 3 for m A, 4 for m E, or 5 for m U. The m D command accepts these prefixes.
12. Infinite mode. Default is *-1* (off). Value is 1 if the mode is on, or 0 if the mode is on with positive zeros. Command is m i.

For example, the sequence M-1 m g RET 2 + ~ p increases the precision by two, leaving a copy of the old precision on the stack. Later, ~ p will restore the original precision using that stack value. (This sequence might be especially useful inside a keyboard macro.)

As another example, M-3 m g 1 - ~ DEL deletes all but the oldest (bottommost) stack entry.

Yet another example: The HP-48 "round" command rounds a number to the current displayed precisi
You could roughly emulate this in Calc with the sequence M-5 m g 10000 % ~ c c. (This would not work for fixed-point mode, but it wouldn't be hard to do a full emulation with the help of the Z [and Z] programming commands. See section Conditionals in Keyboard Macros.)

The Calc Mode Line

This section is a summary of all symbols that can appear on the Calc mode line, the highlighted bar that appears under the Calc stack window (or under an editing window in Embedded Mode).

The basic mode line format is:

```
--%%-Calc: 12 Deg other modes       (Calculator)
```

The `%%' is the Emacs symbol for "read-only"; it shows that regular Emacs commands are not allowed to edit the stack buffer as if it were text.

The word `Calc:' changes to `CalcEmbed:' if Embedded Mode is enabled. The words after this describe the various Calc modes that are in effect.

The first mode is always the current precision, an integer. The second mode is always the angular mode, either `Deg`, `Rad`, or `Hms`.

Here is a complete list of the remaining symbols that can appear on the mode line:

`Alg`
: Algebraic mode (m a; see section Algebraic Entry).

`Alg[(`
: Incomplete algebraic mode (C-u m a).

`Alg*`
: Total algebraic mode (m t).

`Symb`
: Symbolic mode (m s; see section Symbolic Mode).

`Matrix`
: Matrix mode (m v; see section Matrix and Scalar Modes).

`Matrixn`
: Dimensioned matrix mode (C-u n m v).

`Scalar`
: Scalar mode (m v; see section Matrix and Scalar Modes).

`Polar`
: Polar complex mode (m p; see section Polar Mode).

`Frac`

Fraction mode (m f; see section Fraction Mode).

`Inf`
: Infinite mode (m i; see section Infinite Mode).

`+Inf`
: Positive infinite mode (C-u 0 m i).

`NoSimp`
: Default simplifications off (m O; see section Simplification Modes).

`NumSimp`
: Default simplifications for numeric arguments only (m N).

`BinSimpw`
: Binary-integer simplification mode; word size w (m B, b w).

`AlgSimp`
: Algebraic simplification mode (m A).

`ExtSimp`
: Extended algebraic simplification mode (m E).

`UnitSimp`
: Units simplification mode (m U).

`Bin`
: Current radix is 2 (d 2; see section Radix Modes).

`Oct`
: Current radix is 8 (d 8).

`Hex`
: Current radix is 16 (d 6).

`Radixn`
: Current radix is n (d r).

`Zero`
: Leading zeros (d z; see section Radix Modes).

`Big`
: Big language mode (d B; see section Normal Language Modes).

`Flat`
: One-line normal language mode (d O).

`Unform`
: Unformatted language mode (d U).

`C`
: C language mode (d C; see section C, FORTRAN, and Pascal Modes).

`Pascal`

Pascal language mode (d P).

`Fortran`
: FORTRAN language mode (d F).

`TeX`
: TeX language mode (d T; see section TeX Language Mode).

`Eqn`
: Eqn language mode (d E; see section Eqn Language Mode).

`Math`
: Mathematica language mode (d M; see section Mathematica Language Mode).

`Maple`
: Maple language mode (d W; see section Maple Language Mode).

`Normn`
: Normal float mode with n digits (d n; see section Float Formats).

`Fixn`
: Fixed point mode with n digits after the point (d f).

`Sci`
: Scientific notation mode (d s).

`Scin`
: Scientific notation with n digits (d s).

`Eng`
: Engineering notation mode (d e).

`Engn`
: Engineering notation with n digits (d e).

`Leftn`
: Left-justified display indented by n (d <; see section Justification).

`Right`
: Right-justified display (d >).

`Rightn`
: Right-justified display with width n (d >).

`Center`
: Centered display (d =).

`Centern`
: Centered display with center column n (d =).

`Widn`
: Line breaking with width n (d b; see section Normal Language Modes).

`Wide`

No line breaking (d b).

`Break`
: Selections show deep structure (j b; see section Making Selections).

`Save`
: Record modes in `` `~/.emacs' `` (m R; see section General Mode Commands).

`Local`
: Record modes in Embedded buffer (m R).

`LocEdit`
: Record modes as editing-only in Embedded buffer (m R).

`LocPerm`
: Record modes as permanent-only in Embedded buffer (m R).

`Global`
: Record modes as global in Embedded buffer (m R).

`Manual`
: Automatic recomputation turned off (m C; see section Automatic Recomputation).

`Graph`
: GNUPLOT process is alive in background (see section Graphics).

`Sel`
: Top-of-stack has a selection (Embedded only; see section Making Selections).

`Dirty`
: The stack display may not be up-to-date (see section Display Modes).

`Inv`
: "Inverse" prefix was pressed (I; see section Inverse and Hyperbolic Flags).

`Hyp`
: "Hyperbolic" prefix was pressed (H).

`Keep`
: "Keep-arguments" prefix was pressed (K).

`Narrow`
: Stack is truncated (d t; see section Truncating the Stack).

In addition, the symbols `Active` and `~Active` can appear as minor modes on an Embedded buffer's mode line. See section Embedded Mode.

Go to the previous, next section.

Go to the previous, next section.

Arithmetic Functions

This chapter describes the Calc commands for doing simple calculations on numbers, such as addition, absolute value, and square roots. These commands work by removing the top one or two values from the stack, performing the desired operation, and pushing the result back onto the stack. If the operation cannot be performed, the result pushed is a formula instead of a number, such as `2/0' (because division by zero is illegal) or `sqrt(x)' (because the argument `x' is a formula).

Most of the commands described here can be invoked by a single keystroke. Some of the more obscure ones are two-letter sequences beginning with the f ("functions") prefix key.

See section Numeric Prefix Arguments, for a discussion of the effect of numeric prefix arguments on commands in this chapter which do not otherwise interpret a prefix argument.

Basic Arithmetic

The + (`calc-plus`) command adds two numbers. The numbers may be any of the standard Calc data types. The resulting sum is pushed back onto the stack.

If both arguments of + are vectors or matrices (of matching dimensions), the result is a vector or matrix sum. If one argument is a vector and the other a scalar (i.e., a non-vector), the scalar is added to each of the elements of the vector to form a new vector. If the scalar is not a number, the operation is left in symbolic form: Suppose you added `x' to the vector `[1,2]'. You may want the result `[1+x,2+x]', or you may plan to substitute a 2-vector for `x' in the future. Since the Calculator can't tell which interpretation you want, it makes the safest assumption. See section Reducing and Mapping Vectors, for a way to add `x' to every element of a vector.

If either argument of + is a complex number, the result will in general be complex. If one argument is in rectangular form and the other polar, the current Polar Mode determines the form of the result. If Symbolic Mode is enabled, the sum may be left as a formula if the necessary conversions for polar addition are non-trivial.

If both arguments of + are HMS forms, the forms are added according to the usual conventions of hours-minutes-seconds notation. If one argument is an HMS form and the other is a number, that number is converted from degrees or radians (depending on the current Angular Mode) to HMS format and then the two HMS forms are added.

If one argument of + is a date form, the other can be either a real number, which advances the date by a certain number of days, or an HMS form, which advances the date by a certain amount of time. Subtracting two date forms yields the number of days between them. Adding two date forms is meaningless, but Calc interprets it as the subtraction of one date form and the negative of the other. (The negative of a date form can be understood by remembering that dates are stored as the number of days before or after Jan 1, 1 AD.)

If both arguments of + are error forms, the result is an error form with an appropriately computed standard deviation. If one argument is an error form and the other is a number, the number is taken to have zero error. Error forms may have symbolic formulas as their mean and/or error parts; adding these will produce a symbolic error form result. However, adding an error form to a plain symbolic formula (as in `(a +/- b) + c') will not work, for the same reasons just mentioned for vectors. Instead you must write `(a +/- b) + (c +/- 0)'.

If both arguments of + are modulo forms with equal values of M, or if one argument is a modulo form and the other a plain number, the result is a modulo form which represents the sum, modulo M, of the two values.

If both arguments of + are intervals, the result is an interval which describes all possible sums of the possible input values. If one argument is a plain number, it is treated as the interval `[x .. x]'.

If one argument of + is an infinity and the other is not, the result is that same infinity. If both arguments are infinite and in the same direction, the result is the same infinity, but if they are infinite in different directions the result is `nan`.

The - (`calc-minus`) command subtracts two values. The top number on the stack is subtracted from the one behind it, so that the computation 5 RET 2 - produces 3, not *-3*. All options available for + are available for - as well.

The * (`calc-times`) command multiplies two numbers. If one argument is a vector and the other a scalar, the scalar is multiplied by the elements of the vector to produce a new vector. If both arguments are vectors, the interpretation depends on the dimensions of the vectors: If both arguments are matrices, a matrix multiplication is done. If one argument is a matrix and the other a plain vector, the vector is interpreted as a row vector or column vector, whichever is dimensionally correct. If both arguments are plain vectors, the result is a single scalar number which is the dot product of the two vectors.

If one argument of * is an HMS form and the other a number, the HMS form is multiplied by that amount. It is an error to multiply two HMS forms together, or to attempt any multiplication involving date forms. Error forms, modulo forms, and intervals can be multiplied; see the comments for addition of those forms. When two error forms or intervals are multiplied they are considered to be statistically independent; thus, `[-2 .. 3] * [-2 .. 3]' is `[-6 .. 9]', whereas `[-2 .. 3] ^ 2' is `[0 .. 9]'.

The / (`calc-divide`) command divides two numbers. When dividing a scalar B by a square matrix A, the computation performed is B times the inverse of A. This also occurs if B is itself a vector or matrix, in which case the effect is to solve the set of linear equations represented by B. If B is a matrix with the same number of rows as A, or a plain vector (which is interpreted here as a column vector), then the equation A X = B is solved for the vector or matrix X. Otherwise, if B is a non-square matrix with the same number of *columns* as A, the equation X A = B is solved. If you wish a vector B to be interpreted as a row vector to be solved as X A = B, make it into a one-row matrix with C-u 1 v p first. To force a left-handed solution with a square matrix B, transpose A and B before dividing, then transpose the result.

HMS forms can be divided by real numbers or by other HMS forms. Error forms can be divided in any combination of ways. Modulo forms where both values and the modulo are integers can be divided to get an integer modulo form result. Intervals can be divided; dividing by an interval that encompasses zero or has zero as a limit will result in an infinite interval.

The ^ (`calc-power`) command raises a number to a power. If the power is an integer, an exact res computed using repeated multiplications. For non-integer powers, Calc uses Newton's method or logarithms and exponentials. Square matrices can be raised to integer powers. If either argument is an error (or interval or modulo) form, the result is also an error (or interval or modulo) form.

If you press the I (inverse) key first, the I ^ command computes an Nth root: 125 RET 3 I ^ computes the number 5. (This is entirely equivalent to 125 RET 1:3 ^.)

The \ (`calc-idiv`) command divides two numbers on the stack to produce an integer result. It is equivalent to dividing with /, then rounding down with F (`calc-floor`), only a bit more convenient and efficient. Also, since it is an all-integer operation when the arguments are integers, it avoids problems that / F would have with floating-point roundoff.

The % (`calc-mod`) command performs a "modulo" (or "remainder") operation. Mathematically, `a%b = a - (a\b)*b', and is defined for all real numbers a and b (except b=0). For positive b, the result will always be between 0 (inclusive) and b (exclusive). Modulo does not work for HMS forms and error forms. If a is a modulo form, its modulo is changed to b, which must be positive real number.

The : (`calc-fdiv`) command [`fdiv` function in a formula] divides the two integers on the top of the stack to produce a fractional result. This is a convenient shorthand for enabling Fraction Mode (with m f) temporarily and using `/'. Note that during numeric entry the : key is interpreted as a fraction separator, so to divide 8 by 6 you would have to type 8 RET 6 RET :. (Of course, in this case, it would be much easier simply to enter the fraction directly as 8:6 RET!)

The n (`calc-change-sign`) command negates the number on the top of the stack. It works on numbers, vectors and matrices, HMS forms, date forms, error forms, intervals, and modulo forms.

The A (`calc-abs`) [`abs`] command computes the absolute value of a number. The result of `abs` is always a nonnegative real number: With a complex argument, it computes the complex magnitude. With a vector or matrix argument, it computes the Frobenius norm, i.e., the square root of the sum of the squares of the absolute values of the elements. The absolute value of an error form is defined by replacing the mean part with its absolute value and leaving the error part the same. The absolute value of a modulo form is undefined. The absolute value of an interval is defined in the obvious way.

The f A (`calc-abssqr`) [`abssqr`] command computes the absolute value squared of a number, vector or matrix, or error form.

The f s (`calc-sign`) [`sign`] command returns 1 if its argument is positive, *-1* if its argument is negative, or 0 if its argument is zero. In algebraic form, you can also write `sign(a,x)' which evaluates to `x * sign(a)', i.e., either `x', `-x', or zero depending on the sign of `a'.

The & (`calc-inv`) [`inv`] command computes the reciprocal of a number, i.e., 1 / x. Operating on a square matrix, it computes the inverse of that matrix.

The Q (`calc-sqrt`) [`sqrt`] command computes the square root of a number. For a negative real argument, the result will be a complex number whose form is determined by the current Polar Mode.

The f h (`calc-hypot`) [`hypot`] command computes the square root of the sum of the squares of two numbers. That is, `hypot(a,b)' is the length of the hypotenuse of a right triangle with sides a and b. If the arguments are complex numbers, their squared magnitudes are used.

The f Q (`calc-isqrt`) [`isqrt`] command computes the integer square root of an integer. This is the true square root of the number, rounded down to an integer. For example, `isqrt(10)' produces 3. Note that, like \ [`idiv`], this uses exact integer arithmetic throughout to avoid roundoff problems. If the input is a floating-point number or other non-integer value, this is exactly the same as `floor(sqrt(x))'.

The f n (`calc-min`) [`min`] and f x (`calc-max`) [`max`] commands take the minimum or maximum of two real numbers, respectively. These commands also work on HMS forms, date forms, intervals, and infinities. (In algebraic expressions, these functions take any number of arguments and return the maximum or minimum among all the arguments.)

The f M (`calc-mant-part`) [`mant`] function extracts the "mantissa" part m of its floating-point argument; f X (`calc-xpon-part`) [`xpon`] extracts the "exponent" part e. The original number is equal to @c{$m \times 10^e$} m * 10^e, where m is in the interval `[1.0 .. 10.0)' except that m=e=0 if the original number is zero. For integers and fractions, `mant` returns the number unchanged and `xpon` returns zero. The v u (`calc-unpack`) command can also be used to "unpack" a floating-point number; this produces an integer mantissa and exponent, with the constraint that the mantissa is not a multiple of ten (again except for the m=e=0 case).

The f S (`calc-scale-float`) [`scf`] function scales a number by a given power of ten. Thus, `scf(mant(x), xpon(x)) = x' for any real `x'. The second argument must be an integer, but the first may actually be any numeric value. For example, `scf(5,-2) = 0.05' or `1:20' depending on the current Fraction Mode.

The f [(`calc-decrement`) [`decr`] and f] (`calc-increment`) [`incr`] functions decrease or increase a number by one unit. For integers, the effect is obvious. For floating-point numbers, the change is by one unit in the last place. For example, incrementing `12.3456' when the current precision is 6 digits yields `12.3457'. If the current precision had been 8 digits, the result would have been `12.345601'. Incrementing `0.0' produces @c{10^{-p}} 10^-p, where p is the current precision. These operations are defined only on integers and floats. With numeric prefix arguments, they change the number by n units.

Note that incrementing followed by decrementing, or vice-versa, will almost but not quite always cancel out. Suppose the precision is 6 digits and the number `9.99999' is on the stack. Incrementing will produce `10.0000'; decrementing will produce `9.9999'. One digit has been dropped. This is an unavoidable consequence of the way floating-point numbers work.

Incrementing a date/time form adjusts it by a certain number of seconds. Incrementing a pure date form adjusts it by a certain number of days.

Integer Truncation

There are four commands for truncating a real number to an integer, differing mainly in their treatment of negative numbers. All of these commands have the property that if the argument is an integer, the result is the same integer. An integer-valued floating-point argument is converted to integer form.

If you press H (`calc-hyperbolic`) first, the result will be expressed as an integer-valued floating-point number.

The F (`calc-floor`) [`floor` or `ffloor`] command truncates a real number to the next lower int i.e., toward minus infinity. Thus 3.6 F produces 3, but _3.6 F produces -*4*.

The I F (`calc-ceiling`) [`ceil` or `fceil`] command truncates toward positive infinity. Thus 3.6 I F produces 4, and _3.6 I F produces -*3*.

The R (`calc-round`) [`round` or `fround`] command rounds to the nearest integer. When the fractional part is .5 exactly, this command rounds away from zero. (All other rounding in the Calculator uses this convention as well.) Thus 3.5 R produces 4 but 3.4 R produces 3; _3.5 R produces -*4*.

The I R (`calc-trunc`) [`trunc` or `ftrunc`] command truncates toward zero. In other words, it "chops off" everything after the decimal point. Thus 3.6 I R produces 3 and _3.6 I R produces -*3*.

These functions may not be applied meaningfully to error forms, but they do work for intervals. As a convenience, applying `floor` to a modulo form floors the value part of the form. Applied to a vector, these functions operate on all elements of the vector one by one. Applied to a date form, they operate on the internal numerical representation of dates, converting a date/time form into a pure date.

There are two more rounding functions which can only be entered in algebraic notation. The `roundu` function is like `round` except that it rounds up, toward plus infinity, when the fractional part is .5. This distinction matters only for negative arguments. Also, `rounde` rounds to an even number in the case of a tie, rounding up or down as necessary. For example, `round(3.5)' and `round(4.5)' both return 4, but `round(5.5)' returns 6. The advantage of round-to-even is that the net error due to rounding after a long calculation tends to cancel out to zero. An important subtle point here is that the number being fed to `rounde` will already have been rounded to the current precision before `rounde` begins. For example, `round(2.500001)' with a current precision of 6 will incorrectly, or at least surprisingly, yield 2 because the argument will first have been rounded down to 2.5 (which `rounde` sees as an exact tie between 2 and 3).

Each of these functions, when written in algebraic formulas, allows a second argument which specifies the number of digits after the decimal point to keep. For example, `round(123.4567, 2)' will produce the answer 123.46, and `round(123.4567, -1)' will produce 120 (i.e., the cutoff is one digit to the *left* of the decimal point). A second argument of zero is equivalent to no second argument at all.

To compute the fractional part of a number (i.e., the amount which, when added to `floor(*N*)', will produce N) just take N modulo 1 using the % command.

Note also the \ (integer quotient), f I (integer logarithm), and f Q (integer square root) commands, which are analogous to /, B, and Q, respectively, except that they take integer arguments and return the result rounded down to an integer.

Complex Number Functions

The J (`calc-conj`) [`conj`] command computes the complex conjugate of a number. For complex number a+bi, the complex conjugate is a-bi. If the argument is a real number, this command leaves it the same. If the argument is a vector or matrix, this command replaces each element by its complex conjugate.

The G (`calc-argument`) [`arg`] command computes the "argument" or polar angle of a complex number. For a number in polar notation, this is simply the second component of the pair `(r;@c{θ} *theta*)'. The result is expressed according to the current angular mode and will be in the range *-180* degrees (exclusive) to *+180* degrees (inclusive), or the equivalent range in radians.

The `calc-imaginary` command multiplies the number on the top of the stack by the imaginary number i = (0,1). This command is not normally bound to a key in Calc, but it is available on the IMAG button in Keypad Mode.

The f r (`calc-re`) [`re`] command replaces a complex number by its real part. This command has no effect on real numbers. (As an added convenience, `re` applied to a modulo form extracts the value part.)

The f i (`calc-im`) [`im`] command replaces a complex number by its imaginary part; real numbers are converted to zero. With a vector or matrix argument, these functions operate element-wise.

The v p (`calc-pack`) command can pack the top two numbers on the the stack into a composite object such as a complex number. With a prefix argument of *-1*, it produces a rectangular complex number; with an argument of *-2*, it produces a polar complex number. (Also, see section Building Vectors.)

The v u (`calc-unpack`) command takes the complex number (or other composite object) on the top of the stack and unpacks it into its separate components.

Conversions

The commands described in this section convert numbers from one form to another; they are two-key sequences beginning with the letter c.

The c f (`calc-float`) [`pfloat`] command converts the number on the top of the stack to floating-point form. For example, 23 is converted to 23.0, 3:2 is converted to 1.5, and 2.3 is left the same. If the value is a composite object such as a complex number or vector, each of the components is converted to floating-point. If the value is a formula, all numbers in the formula are converted to floating-point. Note that depending on the current floating-point precision, conversion to floating-point format may lose information.

As a special exception, integers which appear as powers or subscripts are not floated by c f. If you really want to float a power, you can use a j s command to select the power followed by c f. Because c f cannot examine the formula outside of the selection, it does not notice that the thing being floated is a power. See section Selecting Sub-Formulas.

The normal c f command is "pervasive" in the sense that it applies to all numbers throughout the formula. The `pfloat` algebraic function never stays around in a formula; `pfloat(a + 1)' changes to `a + 1.0' as soon as it is evaluated.

With the Hyperbolic flag, H c f [`float`] operates only on the number or vector of numbers at the top level of its argument. Thus, `float(1)' is 1.0, but `float(a + 1)' is left unevaluated because its argument is not a number.

You should use H c f if you wish to guarantee that the final value, once all the variables have been

assigned, is a float; you would use c f if you wish to do the conversion on the numbers that appear r
now.

The c F (`calc-fraction`) [`pfrac`] command converts a floating-point number into a fractional approximation. By default, it produces a fraction whose decimal representation is the same as the input number, to within the current precision. You can also give a numeric prefix argument to specify a tolerance, either directly, or, if the prefix argument is zero, by using the number on top of the stack as the tolerance. If the tolerance is a positive integer, the fraction is correct to within that many significant figures. If the tolerance is a non-positive integer, it specifies how many digits fewer than the current precision to use. If the tolerance is a floating-point number, the fraction is correct to within that absolute amount.

The `pfrac` function is pervasive, like `pfloat`. There is also a non-pervasive version, H c F [`frac`], which is analogous to H c f discussed above.

The c d (`calc-to-degrees`) [`deg`] command converts a number into degrees form. The value on the top of the stack may be an HMS form (interpreted as degrees-minutes-seconds), or a real number which will be interpreted in radians regardless of the current angular mode.

The c r (`calc-to-radians`) [`rad`] command converts an HMS form or angle in degrees into an angle in radians.

The c h (`calc-to-hms`) [`hms`] command converts a real number, interpreted according to the current angular mode, to an HMS form describing the same angle. In algebraic notation, the `hms` function also accepts three arguments: `hms(h, m, s)'. (The three-argument version is independent of the current angular mode.)

The `calc-from-hms` command converts the HMS form on the top of the stack into a real number according to the current angular mode.

The c p (`calc-polar`) command converts the complex number on the top of the stack from polar to rectangular form, or from rectangular to polar form, whichever is appropriate. Real numbers are left the same. This command is equivalent to the `rect` or `polar` functions in algebraic formulas, depending on the direction of conversion. (It uses `polar`, except that if the argument is already a polar complex number, it uses `rect` instead. The I c p command always uses `rect`.)

The c c (`calc-clean`) [`pclean`] command "cleans" the number on the top of the stack. Floating point numbers are re-rounded according to the current precision. Polar numbers whose angular components have strayed from the *-180* to *+180* degree range are normalized. (Note that results will be undesirable if the current angular mode is different from the one under which the number was produced!) Integers and fractions are generally unaffected by this operation. Vectors and formulas are cleaned by cleaning each component number (i.e., pervasively).

If the simplification mode is set below the default level, it is raised to the default level for the purposes of this command. Thus, c c applies the default simplifications even if their automatic application is disabled. See section Simplification Modes.

A numeric prefix argument to c c sets the floating-point precision to that value for the duration of the command. A positive prefix (of at least 3) sets the precision to the specified value; a negative or zero

prefix decreases the precision by the specified amount.

The keystroke sequences c 0 through c 9 are equivalent to c c with the corresponding negative prefix argument. If roundoff errors have changed 2.0 into 1.999999, typing c 1 to clip off one decimal place often conveniently does the trick.

The c c command with a numeric prefix argument, and the c 0 through c 9 commands, also "clip" very small floating-point numbers to zero. If the exponent is less than or equal to the negative of the specified precision, the number is changed to 0.0. For example, if the current precision is 12, then c 2 changes the vector `[1e-8, 1e-9, 1e-10, 1e-11]' to `[1e-8, 1e-9, 0, 0]'. Numbers this small generally arise from roundoff noise.

If the numbers you are using really are legitimately this small, you should avoid using the c 0 through c 9 commands. (The plain c c command rounds to the current precision but does not clip small numbers.)

One more property of c 0 through c 9, and of c c with a prefix argument, is that integer-valued floats are converted to plain integers, so that c 1 on `[1., 1.5, 2., 2.5, 3.]' produces `[1, 1.5, 2, 2.5, 3]'. This is not done for huge numbers (`1e100' is technically an integer-valued float, but you wouldn't want it automatically converted to a 100-digit integer).

With the Hyperbolic flag, H c c and H c 0 through H c 9 operate non-pervasively [`clean`].

Date Arithmetic

The commands described in this section perform various conversions and calculations involving date forms (see section Date Forms). They use the t (for time/date) prefix key followed by shifted letters.

The simplest date arithmetic is done using the regular + and - commands. In particular, adding a number to a date form advances the date form by a certain number of days; adding an HMS form to a date form advances the date by a certain amount of time; and subtracting two date forms produces a difference measured in days. The commands described here provide additional, more specialized operations on dates.

Many of these commands accept a numeric prefix argument; if you give plain C-u as the prefix, these commands will instead take the additional argument from the top of the stack.

Date Conversions

The t D (`calc-date`) [`date`] command converts a date form into a number, measured in days since Jan 1, 1 AD. The result will be an integer if date is a pure date form, or a fraction or float if date is a date/time form. Or, if its argument is a number, it converts this number into a date form.

With a numeric prefix argument, t D takes that many objects (up to six) from the top of the stack and interprets them in one of the following ways:

The `date(year, month, day)' function builds a pure date form out of the specified year, month, and day, which must all be integers. Year is a year number, such as 1991 (*not* the same as 91!). Month must be an integer in the range 1 to 12; day must be in the range 1 to 31. If the specified month has fewer than 31

days and day is too large, the equivalent day in the following month will be used.

The `date(month, day)' function builds a pure date form using the current year, as determined by the real-time clock.

The `date(year, month, day, hms)' function builds a date/time form using an hms form.

The `date(year, month, day, hour, minute, second)' function builds a date/time form. hour should be an integer in the range 0 to 23; minute should be an integer in the range 0 to 59; second should be any real number in the range `[0 .. 60)'. The last two arguments default to zero if omitted.

The t J (`calc-julian') [`julian'] command converts a date form into a Julian day count, which is the number of days since noon on Jan 1, 4713 BC. A pure date is converted to an integer Julian count representing noon of that day. A date/time form is converted to an exact floating-point Julian count, adjusted to interpret the date form in the current time zone but the Julian day count in Greenwich Mean Time. A numeric prefix argument allows you to specify the time zone; see section Time Zones. Use a prefix of zero to suppress the time zone adjustment. Note that pure date forms are never time-zone adjusted.

This command can also do the opposite conversion, from a Julian day count (either an integer day, or a floating-point day and time in the GMT zone), into a pure date form or a date/time form in the current or specified time zone.

The t U (`calc-unix-time') [`unixtime'] command converts a date form into a Unix time value, which is the number of seconds since midnight on Jan 1, 1970, or vice-versa. The numeric result will be an integer if the current precision is 12 or less; for higher precisions, the result may be a float with (precision-12) digits after the decimal. Just as for t J, the numeric time is interpreted in the GMT time zone and the date form is interpreted in the current or specified zone. Some systems use Unix-like numbering but with the local time zone; give a prefix of zero to suppress the adjustment if so.

The t C (`calc-convert-time-zones') [`tzconv'] command converts a date form from one time zone to another. You are prompted for each time zone name in turn; you can answer with any suitable Calc time zone expression (see section Time Zones). If you answer either prompt with a blank line, the local time zone is used for that prompt. You can also answer the first prompt with $ to take the two time zone names from the stack (and the date to be converted from the third stack level).

Date Functions

The t N (`calc-now') [`now'] command pushes the current date and time on the stack as a date form. The time is reported in terms of the specified time zone; with no numeric prefix argument, t N reports for the current time zone.

The t P (`calc-date-part') command extracts one part of a date form. The prefix argument specifies the part; with no argument, this command prompts for a part code from 1 to 9. The various part codes are described in the following paragraphs.

The M-1 t P [`year'] function extracts the year number from a date form as an integer, e.g., 1991. This and the following functions will also accept a real number for an argument, which is interpreted as a

standard Calc day number. Note that this function will never return zero, since the year 1 BC immediately precedes the year 1 AD.

The M-2 t P [`month`] function extracts the month number from a date form as an integer in the range 1 to 12.

The M-3 t P [`day`] function extracts the day number from a date form as an integer in the range 1 to 31.

The M-4 t P [`hour`] function extracts the hour from a date form as an integer in the range 0 (midnight) to 23. Note that 24-hour time is always used. This returns zero for a pure date form. This function (and the following two) also accept HMS forms as input.

The M-5 t P [`minute`] function extracts the minute from a date form as an integer in the range 0 to 59.

The M-6 t P [`second`] function extracts the second from a date form. If the current precision is 12 or less, the result is an integer in the range 0 to 59. For higher precisions, the result may instead be a floating-point number.

The M-7 t P [`weekday`] function extracts the weekday number from a date form as an integer in the range 0 (Sunday) to 6 (Saturday).

The M-8 t P [`yearday`] function extracts the day-of-year number from a date form as an integer in the range 1 (January 1) to 366 (December 31 of a leap year).

The M-9 t P [`time`] function extracts the time portion of a date form as an HMS form. This returns `0@ 0' 0"' for a pure date form.

The t M (`calc-new-month`) [`newmonth`] command computes a new date form that represents the first day of the month specified by the input date. The result is always a pure date form; only the year and month numbers of the input are retained. With a numeric prefix argument n in the range from 1 to 31, t M computes the nth day of the month. (If n is greater than the actual number of days in the month, or if n is zero, the last day of the month is used.)

The t Y (`calc-new-year`) [`newyear`] command computes a new pure date form that represents the first day of the year specified by the input. The month, day, and time of the input date form are lost. With a numeric prefix argument n in the range from 1 to 366, t Y computes the nth day of the year (366 is treated as 365 in non-leap years). A prefix argument of 0 computes the last day of the year (December 31). A negative prefix argument from *-1* to *-12* computes the first day of the nth month of the year.

The t W (`calc-new-week`) [`newweek`] command computes a new pure date form that represents the Sunday on or before the input date. With a numeric prefix argument, it can be made to use any day of the week as the starting day; the argument must be in the range from 0 (Sunday) to 6 (Saturday). This function always subtracts between 0 and 6 days from the input date.

Here's an example use of `newweek`: Find the date of the next Wednesday after a given date. Using M-3 t W or `newweek(d, 3)' will give you the *preceding* Wednesday, so `newweek(d+7, 3)' will give you the following Wednesday. A further look at the definition of `newweek` shows that if the input date is itself a Wednesday, this formula will return the Wednesday one week in the future. An exercise for the reader is to modify this formula to yield the same day if the input is already a Wednesday. Another interesting exercise is to preserve the time-of-day portion of the input (`newweek` resets the time to midnight; hint:

how can `newweek` be defined in terms of the `weekday` function?).

The `pwday(date)' function (not on any key) computes the day-of-month number of the Sunday on or before date. With two arguments, `pwday(date, day)' computes the day number of the Sunday on or before day number day of the month specified by date. The day must be in the range from 7 to 31; if the day number is greater than the actual number of days in the month, the true number of days is used instead. Thus `pwday(date, 7)' finds the first Sunday of the month, and `pwday(date, 31)' finds the last Sunday of the month. With a third weekday argument, `pwday` can be made to look for any day of the week instead of Sunday.

The t I (`calc-inc-month`) [`incmonth`] command increases a date form by one month, or by an arbitrary number of months specified by a numeric prefix argument. The time portion, if any, of the date form stays the same. The day also stays the same, except that if the new month has fewer days the day number may be reduced to lie in the valid range. For example, `incmonth(<Jan 31, 1991>)' produces `<Feb 28, 1991>'. Because of this, t I t I and M-2 t I do not always give the same results (`<Mar 28, 1991>' versus `<Mar 31, 1991>' in this case).

The `incyear(date, step)' function increases a date form by the specified number of years, which may be any positive or negative integer. Note that `incyear(d, n)' is equivalent to `incmonth(d, 12*n)', but these do not have simple equivalents in terms of day arithmetic because months and years have varying lengths. If the step argument is omitted, 1 year is assumed. There is no keyboard command for this function; use C-u 12 t I instead.

There is no `newday` function at all because F [`floor`] serves this purpose. Similarly, instead of `incday` and `incweek` simply use d + n or d + 7 n.

See section Basic Arithmetic, for the f] [`incr`] command which can adjust a date/time form by a certain number of seconds.

Business Days

Often time is measured in "business days" or "working days," where weekends and holidays are skipped. Calc's normal date arithmetic functions use calendar days, so that subtracting two consecutive Mondays will yield a difference of 7 days. By contrast, subtracting two consecutive Mondays would yield 5 business days (assuming two-day weekends and the absence of holidays).

The t + (`calc-business-days-plus`) [`badd`] and t - (`calc-business-days-minus`) [`bsub`] commands perform arithmetic using business days. For t +, one argument must be a date form and the other must be a real number (positive or negative). If the number is not an integer, then a certain amount of time is added as well as a number of days; for example, adding 0.5 business days to a time in Friday evening will produce a time in Monday morning. It is also possible to add an HMS form; adding `12@ 0' 0"' also adds half a business day. For t -, the arguments are either a date form and a number or HMS form, or two date forms, in which case the result is the number of business days between the two dates.

By default, Calc considers any day that is not a Saturday or Sunday to be a business day. You can define any number of additional holidays by editing the variable `Holidays`. (There is an s H convenience command for editing this variable.) Initially, `Holidays` contains the vector `[sat, sun]'. Entries in the

`Holidays` vector may be any of the following kinds of objects:

- Date forms (pure dates, not date/time forms). These specify particular days which are to be treated as holidays.
- Intervals of date forms. These specify a range of days, all of which are holidays (e.g., Christmas week). See section Interval Forms.
- Nested vectors of date forms. Each date form in the vector is considered to be a holiday.
- Any Calc formula which evaluates to one of the above three things. If the formula involves the variable y, it stands for a yearly repeating holiday; y will take on various year numbers like 1992. For example, `date(y, 12, 25)' specifies Christmas day, and `newweek(date(y, 11, 7), 4) + 21' specifies Thanksgiving (which is held on the fourth Thursday of November). If the formula involves the variable m, that variable takes on month numbers from 1 to 12: `date(y, m, 15)' is a holiday that takes place on the 15th of every month.
- A weekday name, such as `sat` or `sun`. This is really a variable whose name is a three-letter, lower-case day name.
- An interval of year numbers (integers). This specifies the span of years over which this holiday list is to be considered valid. Any business-day arithmetic that goes outside this range will result in an error message. Use this if you are including an explicit list of holidays, rather than a formula to generate them, and you want to make sure you don't accidentally go beyond the last point where the holidays you entered are complete. If there is no limiting interval in the `Holidays` vector, the default `[1 .. 2737]' is used. (This is the absolute range of years for which Calc's business-day algorithms will operate.)
- An interval of HMS forms. This specifies the span of hours that are to be considered one business day. For example, if this range is `[9@ 0' 0" .. 17@ 0' 0"]' (i.e., 9am to 5pm), then the business day is only eight hours long, so that 1.5 t + on `<4:00pm Fri Dec 13, 1991>' will add one business day and four business hours to produce `<12:00pm Tue Dec 17, 1991>'. Likewise, t - will now express differences in time as fractions of an eight-hour day. Times before 9am will be treated as 9am by business date arithmetic, and times at or after 5pm will be treated as 4:59:59pm. If there is no HMS interval in `Holidays`, the full 24-hour day `[0 0' 0" .. 24 0' 0"]' is assumed. (Regardless of the type of bounds you specify, the interval is treated as inclusive on the low end and exclusive on the high end, so that the work day goes from 9am up to, but not including, 5pm.)

If the `Holidays` vector is empty, then t + and t - will act just like + and - because there will then be no difference between business days and calendar days.

Calc expands the intervals and formulas you give into a complete list of holidays for internal use. This is done mainly to make sure it can detect multiple holidays. (For example, `<Jan 1, 1989>' is both New Year's Day and a Sunday, but Calc's algorithms take care to count it only once when figuring the number of holidays between two dates.)

Since the complete list of holidays for all the years from 1 to 2737 would be huge, Calc actually computes only the part of the list between the smallest and largest years that have been involved in business-day calculations so far. Normally, you won't have to worry about this. Keep in mind, however, that if you do one calculation for 1992, and another for 1792, even if both involve only a small range of years, Calc will still work out all the holidays that fall in that 200-year span.

If you add a (positive) number of days to a date form that falls on a weekend or holiday, the date form is treated as if it were the most recent business day. (Thus adding one business day to a Friday, Saturday, or Sunday will all yield the following Monday.) If you subtract a number of days from a weekend or holiday, the date is effectively on the following business day. (So subtracting one business day from Saturday, Sunday, or Monday yields the preceding Friday.) The difference between two dates one or both of which fall on holidays equals the number of actual business days between them. These conventions are consistent in the sense that, if you add n business days to any date, the difference between the result and the original date will come out to n business days. (It can't be completely consistent though; a subtraction followed by an addition might come out a bit differently, since t + is incapable of producing a date that falls on a weekend or holiday.)

There is a `holiday` function, not on any keys, that takes any date form and returns 1 if that date falls on a weekend or holiday, as defined in `Holidays`, or 0 if the date is a business day.

Time Zones

Time zones and daylight savings time are a complicated business. The conversions to and from Julian and Unix-style dates automatically compute the correct time zone and daylight savings adjustment to use, provided they can figure out this information. This section describes Calc's time zone adjustment algorithm in detail, in case you want to do conversions in different time zones or in case Calc's algorithms can't determine the right correction to use.

Adjustments for time zones and daylight savings time are done by t U, t J, t N, and t C, but not by any other commands. In particular, `<may 1 1991> - <apr 1 1991>' evaluates to exactly 30 days even though there is a daylight-savings transition in between. This is also true for Julian pure dates: `julian(<may 1 1991>) - julian(<apr 1 1991>)'. But Julian and Unix date/times will adjust for daylight savings time: `julian(<12am may 1 1991>) - julian(<12am apr 1 1991>)' evaluates to `29.95834' (that's 29 days and 23 hours) because one hour was lost when daylight savings commenced on April 7, 1991.

In brief, the idiom `julian(date1) - julian(date2)' computes the actual number of 24-hour periods between two dates, whereas `date1 - date2' computes the number of calendar days between two dates without taking daylight savings into account.

The `calc-time-zone` [`tzone`] command converts the time zone specified by its numeric prefix argument into a number of seconds difference from Greenwich mean time (GMT). If the argument is a number, the result is simply that value multiplied by 3600. Typical arguments for North America are 5 (Eastern) or 8 (Pacific). If Daylight Savings time is in effect, one hour should be subtracted from the normal difference.

If you give a prefix of plain C-u, `calc-time-zone` (like other date arithmetic commands that include a time zone argument) takes the zone argument from the top of the stack. (In the case of t J and t U, the normal argument is then taken from the second-to-top stack position.) This allows you to give a non-integer time zone adjustment. The time-zone argument can also be an HMS form, or it can be a variable which is a time zone name in upper- or lower-case. For example `tzone(PST) = tzone(8)' and `tzone(pdt) = tzone(7)' (for Pacific standard and daylight savings times, respectively).

North American and European time zone names are defined as follows; note that for each time zone there is one name for standard time, another for daylight savings time, and a third for "generalized" time in

which the daylight savings adjustment is computed from context.

```
YST   PST   MST   CST   EST   AST     NST     GMT   WET     MET     MEZ
 9     8     7     6     5     4       3.5     0     -1      -2      -2

YDT   PDT   MDT   CDT   EDT   ADT     NDT     BST   WETDST  METDST  MESZ
 8     7     6     5     4     3       2.5     -1    -2      -3      -3

YGT   PGT   MGT   CGT   EGT   AGT     NGT     BGT   WEGT    MEGT    MEGZ
9/8   8/7   7/6   6/5   5/4   4/3   3.5/2.5   0/-1 -1/-2   -2/-3   -2/-3
```

To define time zone names that do not appear in the above table, you must modify the Lisp variable `math-tzone-names`. This is a list of lists describing the different time zone names; its structure is best explained by an example. The three entries for Pacific Time look like this:

```
( ( "PST" 8 0 )    ; Name as an upper-case string, then standard
  ( "PDT" 8 -1 )   ; adjustment, then daylight savings adjustment.
  ( "PGT" 8 "PST" "PDT" ) )   ; Generalized time zone.
```

With no arguments, `calc-time-zone` or `tzone()' obtains an argument from the Calc variable `TimeZone` if a value has been stored for that variable. If not, Calc runs the Unix `date' command and looks for one of the above time zone names in the output; if this does not succeed, `tzone()' leaves itself unevaluated. The time zone name in the `date' output may be followed by a signed adjustment, e.g., `GMT+5' or `GMT+0500' which specifies a number of hours and minutes to be added to the base time zone. Calc stores the time zone it finds into `TimeZone` to speed later calls to `tzone()'.

The special time zone name `local` is equivalent to no argument, i.e., it uses the local time zone as obtained from the `date` command.

If the time zone name found is one of the standard or daylight savings zone names from the above table, and Calc's internal daylight savings algorithm says that time and zone are consistent (e.g., `PDT` accompanies a date that Calc's algorithm would also consider to be daylight savings, or `PST` accompanies a date that Calc would consider to be standard time), then Calc substitutes the corresponding generalized time zone (like `PGT`).

If your system does not have a suitable `date' command, you may wish to put a `(setq var-TimeZone ...)' in your Emacs initialization file to set the time zone. The easiest way to do this is to edit the `TimeZone` variable using Calc's s T command, then use the s p (`calc-permanent-variable`) command to save the value of `TimeZone` permanently.

The t J and `t U` commands with no numeric prefix arguments do the same thing as `tzone()'. If the current time zone is a generalized time zone, e.g., `EGT`, Calc examines the date being converted to tell whether to use standard or daylight savings time. But if the current time zone is explicit, e.g., `EST` or `EDT`, then that adjustment is used exactly and Calc's daylight savings algorithm is not consulted.

Some places don't follow the usual rules for daylight savings time. The state of Arizona, for example, does not observe daylight savings time. If you run Calc during the winter season in Arizona, the Unix `date` command will report `MST` time zone, which Calc will change to `MGT`. If you then convert a time

that lies in the summer months, Calc will apply an incorrect daylight savings time adjustment. To av this, set your `TimeZone` variable explicitly to `MST` to force the use of standard, non-daylight-savings time.

By default Calc always considers daylight savings time to begin at 2 a.m. on the first Sunday of April, and to end at 2 a.m. on the last Sunday of October. This is the rule that has been in effect in North America since 1987. If you are in a country that uses different rules for computing daylight savings time, you have two choices: Write your own daylight savings hook, or control time zones explicitly by setting the `TimeZone` variable and/or always giving a time-zone argument for the conversion functions.

The Lisp variable `math-daylight-savings-hook` holds the name of a function that is used to compute the daylight savings adjustment for a given date. The default is `math-std-daylight-savings`, which computes an adjustment (either 0 or *-1*) using the North American rules given above.

The daylight savings hook function is called with four arguments: The date, as a floating-point number in standard Calc format; a six-element list of the date decomposed into year, month, day, hour, minute, and second, respectively; a string which contains the generalized time zone name in upper-case, e.g., `"WEGT"`; and a special adjustment to be applied to the hour value when converting into a generalized time zone (see below).

The Lisp function `math-prev-weekday-in-month` is useful for daylight savings computations. This is an internal version of the user-level `pwday` function described in the previous section. It takes four arguments: The floating-point date value, the corresponding six-element date list, the day-of-month number, and the weekday number (0-6).

The default daylight savings hook ignores the time zone name, but a more sophisticated hook could use different algorithms for different time zones. It would also be possible to use different algorithms depending on the year number, but the default hook always uses the algorithm for 1987 and later. Here is a listing of the default daylight savings hook:

```
(defun math-std-daylight-savings (date dt zone bump)
  (cond ((< (nth 1 dt) 4) 0)
        ((= (nth 1 dt) 4)
         (let ((sunday (math-prev-weekday-in-month date dt 7 0)))
           (cond ((< (nth 2 dt) sunday) 0)
                 ((= (nth 2 dt) sunday)
                  (if (>= (nth 3 dt) (+ 3 bump)) -1 0))
                 (t -1))))
        ((< (nth 1 dt) 10) -1)
        ((= (nth 1 dt) 10)
         (let ((sunday (math-prev-weekday-in-month date dt 31 0)))
           (cond ((< (nth 2 dt) sunday) -1)
                 ((= (nth 2 dt) sunday)
                  (if (>= (nth 3 dt) (+ 2 bump)) 0 -1))
                 (t 0))))
        (t 0))
```

)

The `bump` parameter is equal to zero when Calc is converting from a date form in a generalized time zone into a GMT date value. It is *-1* when Calc is converting in the other direction. The adjustments shown above ensure that the conversion behaves correctly and reasonably around the 2 a.m. transition in each direction.

There is a "missing" hour between 2 a.m. and 3 a.m. at the beginning of daylight savings time; converting a date/time form that falls in this hour results in a time value for the following hour, from 3 a.m. to 4 a.m. At the end of daylight savings time, the hour from 1 a.m. to 2 a.m. repeats itself; converting a date/time form that falls in in this hour results in a time value for the first manifestion of that time (*not* the one that occurs one hour later).

If `math-daylight-savings-hook` is `nil`, then the daylight savings adjustment is always taken to be zero.

In algebraic formulas, `tzone(zone, date)' computes the time zone adjustment for a given zone name at a given date. The date is ignored unless zone is a generalized time zone. If date is a date form, the daylight savings computation is applied to it as it appears. If date is a numeric date value, it is adjusted for the daylight-savings version of zone before being given to the daylight savings hook. This odd-sounding rule ensures that the daylight-savings computation is always done in local time, not in the GMT time that a numeric date is typically represented in.

The `dsadj(date, zone)' function computes the daylight savings adjustment that is appropriate for date in time zone zone. If zone is explicitly in or not in daylight savings time (e.g., `PDT` or `PST`) the date is ignored. If zone is a generalized time zone, the algorithms described above are used. If zone is omitted, the computation is done for the current time zone.

See section Reporting Bugs, for the address of Calc's author, if you should wish to contribute your improved versions of `math-tzone-names` and `math-daylight-savings-hook` to the Calc distribution.

Financial Functions

Calc's financial or business functions use the b prefix key followed by a shifted letter. (The b prefix followed by a lower-case letter is used for operations on binary numbers.)

Note that the rate and the number of intervals given to these functions must be on the same time scale, e.g., both months or both years. Mixing an annual interest rate with a time expressed in months will give you very wrong answers!

It is wise to compute these functions to a higher precision than you really need, just to make sure your answer is correct to the last penny; also, you may wish to check the definitions at the end of this section to make sure the functions have the meaning you expect.

Percentages

The M-% (`calc-percent`) command takes a percentage value, say 5.4, and converts it to an equivalent actual number. For example, 5.4 M-% enters 0.054 on the stack. (That's the META or ESC key combined with %.)

Actually, M-% creates a formula of the form `5.4%'. You can enter `5.4%' yourself during algebraic entry. The `%' operator simply means, "the preceding value divided by 100." The `%' operator has very high precedence, so that `1+8%' is interpreted as `1+(8%)', not as `(1+8)%'. (The `%' operator is just a postfix notation for the `percent` function, just like `20!' is the notation for `fact(20)', or twenty-factorial.)

The formula `5.4%' would normally evaluate immediately to 0.054, but the M-% command suppresses evaluation as it puts the formula onto the stack. However, the next Calc command that uses the formula `5.4%' will evaluate it as its first step. The net effect is that you get to look at `5.4%' on the stack, but Calc commands see it as `0.054', which is what they expect.

In particular, `5.4%' and `0.054' are suitable values for the rate arguments of the various financial functions, but the number `5.4' is probably *not* suitable--it represents a rate of 540 percent!

The key sequence M-% * effectively means "percent-of." For example, 68 RET 25 M-% * computes 17, which is 25% of 68 (and also 68% of 25, which comes out to the same thing).

The c % (`calc-convert-percent`) command converts the value on the top of the stack from numeric to percentage form. For example, if 0.08 is on the stack, c % converts it to `8%'. The quantity is the same, it's just represented differently. (Contrast this with M-%, which would convert this number to `0.08%'.) The = key is a convenient way to convert a formula like `8%' back to numeric form, 0.08.

To compute what percentage one quantity is of another quantity, use / c %. For example, 17 RET 68 / c % displays `25%'.

The b % (`calc-percent-change`) [`relch`] command calculates the percentage change from one number to another. For example, 40 RET 50 b % produces the answer `25%', since 50 is 25% larger than 40. A negative result represents a decrease: 50 RET 40 b % produces `-20%', since 40 is 20% smaller than 50. (The answers are different in magnitude because, in the first case, we're increasing by 25% of 40, but in the second case, we're decreasing by 20% of 50.) The effect of 40 RET 50 b % is to compute (50-40)/40, converting the answer to percentage form as if by c %.

Future Value

The b F (`calc-fin-fv`) [`fv`] command computes the future value of an investment. It takes three arguments from the stack: `fv(rate, n, payment)'. If you give payments of payment every year for n years, and the money you have paid earns interest at rate per year, then this function tells you what your investment would be worth at the end of the period. (The actual interval doesn't have to be years, as long as n and rate are expressed in terms of the same intervals.) This function assumes payments occur at the *end* of each interval.

The I b F [`fvb`] command does the same computation, but assuming your payments are at the beginning

of each interval. Suppose you plan to deposit $1000 per year in a savings account earning 5.4% inter starting right now. How much will be in the account after five years? `fvb(5.4%, 5, 1000) = 5870.73`. Thus you will have earned $870 worth of interest over the years. Using the stack, this calculation would have been 5.4 M-% 5 RET 1000 I b F. Note that the rate is expressed as a number between 0 and 1, *not* as a percentage.

The H b F [`fvl`] command computes the future value of an initial lump sum investment. Suppose you could deposit those five thousand dollars in the bank right now; how much would they be worth in five years? `fvl(5.4%, 5, 5000) = 6503.89`.

The algebraic functions `fv` and `fvb` accept an optional fourth argument, which is used as an initial lump sum in the sense of `fvl`. In other words, `fv(rate, n, payment, initial) = fv(rate, n, payment) + fvl(rate, n, initial)`.

To illustrate the relationships between these functions, we could do the `fvb` calculation "by hand" using `fvl`. The final balance will be the sum of the contributions of our five deposits at various times. The first deposit earns interest for five years: `fvl(5.4%, 5, 1000) = 1300.78`. The second deposit only earns interest for four years: `fvl(5.4%, 4, 1000) = 1234.13`. And so on down to the last deposit, which earns one year's interest: `fvl(5.4%, 1, 1000) = 1054.00`. The sum of these five values is, sure enough, $5870.73, just as was computed by `fvb` directly.

What does `fv(5.4%, 5, 1000) = 5569.96` mean? The payments are now at the ends of the periods. The end of one year is the same as the beginning of the next, so what this really means is that we've lost the payment at year zero (which contributed $1300.78), but we're now counting the payment at year five (which, since it didn't have a chance to earn interest, counts as $1000). Indeed, 5569.96 = 5870.73 - 1300.78 + 1000 (give or take a bit of roundoff error).

Present Value

The b P (`calc-fin-pv`) [`pv`] command computes the present value of an investment. Like `fv`, it takes three arguments: `pv(rate, n, payment)`. It computes the present value of a series of regular payments. Suppose you have the chance to make an investment that will pay $2000 per year over the next four years; as you receive these payments you can put them in the bank at 9% interest. You want to know whether it is better to make the investment, or to keep the money in the bank where it earns 9% interest right from the start. The calculation `pv(9%, 4, 2000)` gives the result 6479.44. If your initial investment must be less than this, say, $6000, then the investment is worthwhile. But if you had to put up $7000, then it would be better just to leave it in the bank.

Here is the interpretation of the result of `pv`: You are trying to compare the return from the investment you are considering, which is `fv(9%, 4, 2000) = 9146.26`, with the return from leaving the money in the bank, which is `fvl(9%, 4, x)` where x is the amount of money you would have to put up in advance. The `pv` function finds the break-even point, x = 6479.44, at which `fvl(9%, 4, 6479.44)` is also equal to 9146.26. This is the largest amount you should be willing to invest.

The I b P [`pvb`] command solves the same problem, but with payments occurring at the beginning of each interval. It has the same relationship to `fvb` as `pv` has to `fv`. For example `pvb(9%, 4, 2000) = 7062.59`, a larger number than `pv` produced because we get to start earning interest on the return from our investment sooner.

The H b P [pvl] command computes the present value of an investment that will pay off in one lump sum at the end of the period. For example, if we get our $8000 all at the end of the four years, `pvl(9%, 4, 8000) = 5667.40`. This is much less than pv reported, because we don't earn any interest on the return from this investment. Note that pvl and fvl are simple inverses: `fvl(9%, 4, 5667.40) = 8000`.

You can give an optional fourth lump-sum argument to pv and pvb; this is handled in exactly the same way as the fourth argument for fv and fvb.

The b N (`calc-fin-npv`) [npv] command computes the net present value of a series of irregular investments. The first argument is the interest rate. The second argument is a vector which represents the expected return from the investment at the end of each interval. For example, if the rate represents a yearly interest rate, then the vector elements are the return from the first year, second year, and so on.

Thus, `npv(9%, [2000,2000,2000,2000]) = pv(9%, 4, 2000) = 6479.44`. Obviously this function is more interesting when the payments are not all the same!

The npv function can actually have two or more arguments. Multiple arguments are interpreted in the same way as for the vector statistical functions like vsum. See section Single-Variable Statistics. Basically, if there are several payment arguments, each either a vector or a plain number, all these values are collected left-to-right into the complete list of payments. A numeric prefix argument on the b N command says how many payment values or vectors to take from the stack.

The I b N [npvb] command computes the net present value where payments occur at the beginning of each interval rather than at the end.

Related Financial Functions

The functions in this section are basically inverses of the present value functions with respect to the various arguments.

The b M (`calc-fin-pmt`) [pmt] command computes the amount of periodic payment necessary to amortize a loan. Thus `pmt(rate, n, amount)` equals the value of payment such that `pv(rate, n, payment) = amount`.

The I b M [pmtb] command does the same computation but using pvb instead of pv. Like pv and pvb, these functions can also take a fourth argument which represents an initial lump-sum investment.

The H b M key just invokes the fvl function, which is the inverse of pvl. There is no explicit pmtl function.

The b # (`calc-fin-nper`) [nper] command computes the number of regular payments necessary to amortize a loan. Thus `nper(rate, payment, amount)` equals the value of n such that `pv(rate, n, payment) = amount`. If payment is too small ever to amortize a loan for amount at interest rate rate, the nper function is left in symbolic form.

The I b # [nperb] command does the same computation but using pvb instead of pv. You can give a fourth lump-sum argument to these functions, but the computation will be rather slow in the

four-argument case.

The H b # [`nperl`] command does the same computation using `pvl`. By exchanging payment and amount you can also get the solution for `fvl`. For example, `nperl(8%, 2000, 1000) = 9.006`, so if you place $1000 in a bank account earning 8%, it will take nine years to grow to $2000.

The b T (`calc-fin-rate`) [`rate`] command computes the rate of return on an investment. This is also an inverse of `pv`: `rate(n, payment, amount)` computes the value of rate such that `pv(rate, n, payment) = amount`. The result is expressed as a formula like `6.3%'.

The I b T [`rateb`] and H b T [`ratel`] commands solve the analogous equations with `pvb` or `pvl` in place of `pv`. Also, `rate` and `rateb` can accept an optional fourth argument just like `pv` and `pvb`. To redo the above example from a different perspective, `ratel(9, 2000, 1000) = 8.00597%`, which says you will need an interest rate of 8% in order to double your account in nine years.

The b I (`calc-fin-irr`) [`irr`] command is the analogous function to `rate` but for net present value. Its argument is a vector of payments. Thus `irr(payments)` computes the rate such that `npv(rate, payments) = 0`; this rate is known as the internal rate of return.

The I b I [`irrb`] command computes the internal rate of return assuming payments occur at the beginning of each period.

Depreciation Functions

The functions in this section calculate depreciation, which is the amount of value that a possession loses over time. These functions are characterized by three parameters: cost, the original cost of the asset; salvage, the value the asset will have at the end of its expected "useful life"; and life, the number of years (or other periods) of the expected useful life.

There are several methods for calculating depreciation that differ in the way they spread the depreciation over the lifetime of the asset.

The b S (`calc-fin-sln`) [`sln`] command computes the "straight-line" depreciation. In this method, the asset depreciates by the same amount every year (or period). For example, `sln(12000, 2000, 5)' returns 2000. The asset costs $12000 initially and will be worth $2000 after five years; it loses $2000 per year.

The b Y (`calc-fin-syd`) [`syd`] command computes the accelerated "sum-of-years'-digits" depreciation. Here the depreciation is higher during the early years of the asset's life. Since the depreciation is different each year, b Y takes a fourth period parameter which specifies which year is requested, from 1 to life. If period is outside this range, the `syd` function will return zero.

The b D (`calc-fin-ddb`) [`ddb`] command computes an accelerated depreciation using the double-declining balance method. It also takes a fourth period parameter.

For symmetry, the `sln` function will accept a period parameter as well, although it will ignore its value except that the return value will as usual be zero if period is out of range.

For example, pushing the vector [1,2,3,4,5] (perhaps with v x 5) and then mapping V M '

[sln(12000,2000,5,$), syd(12000,2000,5,$), ddb(12000,2000,5,$)] RET produces a matrix that allows to compare the three depreciation methods:

```
[ [ 2000, 3333, 4800 ]
  [ 2000, 2667, 2880 ]
  [ 2000, 2000, 1728 ]
  [ 2000, 1333,  592 ]
  [ 2000,  667,   0  ] ]
```

(Values have been rounded to nearest integers in this figure.) We see that `sln` depreciates by the same amount each year, syd depreciates more at the beginning and less at the end, and ddb weights the depreciation even more toward the beginning.

Summing columns with V R : + yields [10000, 10000, 10000]; the total depreciation in any method is (by definition) the difference between the cost and the salvage value.

Definitions

For your reference, here are the actual formulas used to compute Calc's financial functions.

Calc will not evaluate a financial function unless the rate or n argument is known. However, payment or amount can be a variable. Calc expands these functions according to the formulas below for symbolic arguments only when you use the a " (`calc-expand-formula`) command, or when taking derivatives or integrals or solving equations involving the functions.

In `pmt` and `pmtb`, x=0 if omitted.

These functions accept any numeric objects, including error forms, intervals, and even (though not very usefully) complex numbers. The above formulas specify exactly the behavior of these functions with all sorts of inputs.

Note that if the first argument to the `log` in `nper` is negative, `nper` leaves itself in symbolic form rather than returning a (financially meaningless) complex number.

`rate(num, pmt, amt)' solves the equation `pv(rate, num, pmt) = amt' for `rate' using H a R (`calc-find-root`), with the interval `[.01% .. 100%]' for an initial guess. The `rateb` function is the same except that it uses `pvb`. Note that `ratel` can be solved directly; its formula is shown in the above list.

Similarly, `irr(pmts)' solves the equation `npv(rate, pmts) = 0' for `rate'.

If you give a fourth argument to `nper` or `nperb`, Calc will also use H a R to solve the equation using an initial guess interval of `[0 .. 100]'.

A fourth argument to `fv` simply sums the two components calculated from the above formulas for `fv` and `fvl`. The same is true of `fvb`, `pv`, and `pvb`.

The ddb function is computed iteratively; the "book" value starts out equal to cost, and decreases according to the above formula for the specified number of periods. If the book value would decrease

below salvage, it only decreases to salvage and the depreciation is zero for all subsequent periods. Th `ddb` function returns the amount the book value decreased in the specified period.

The Calc financial function names were borrowed mostly from Microsoft Excel and Borland's Quattro. The `ratel` function corresponds to `@CGR' in Borland's Reflex. The `nper` and `nperl` functions correspond to `@TERM' and `@CTERM' in Quattro, respectively. Beware that the Calc functions may take their arguments in a different order than the corresponding functions in your favorite spreadsheet.

Binary Number Functions

The commands in this chapter all use two-letter sequences beginning with the b prefix.

The "binary" operations actually work regardless of the currently displayed radix, although their results make the most sense in a radix like 2, 8, or 16 (as obtained by the d 2, d 8, or d 6 commands, respectively). You may also wish to enable display of leading zeros with d z. See section Radix Modes.

The Calculator maintains a current word size w, an arbitrary positive or negative integer. For a positive word size, all of the binary operations described here operate modulo 2^w. In particular, negative arguments are converted to positive integers modulo 2^w by all binary functions.

If the word size is negative, binary operations produce 2's complement integers from @c{-2^{-w-1}} -(2^(-w-1)) to @c{$2^{-w-1}-1$} 2^(-w-1)-1 inclusive. Either mode accepts inputs in any range; the sign of w affects only the results produced.

The b c (`calc-clip`) [`clip`] command can be used to clip a number by reducing it modulo 2^w. The commands described in this chapter automatically clip their results to the current word size. Note that other operations like addition do not use the current word size, since integer addition generally is not "binary." (However, see section Simplification Modes, `calc-bin-simplify-mode`.) For example, with a word size of 8 bits b c converts a number to the range 0 to 255; with a word size of *-8* b c converts to the range *-128* to 127.

The default word size is 32 bits. All operations except the shifts and rotates allow you to specify a different word size for that one operation by giving a numeric prefix argument: C-u 8 b c clips the top of stack to the range 0 to 255 regardless of the current word size. To set the word size permanently, use b w (`calc-word-size`). This command displays a prompt with the current word size; press RET immediately to keep this word size, or type a new word size at the prompt.

When the binary operations are written in symbolic form, they take an optional second (or third) word-size parameter. When a formula like `and(a,b)' is finally evaluated, the word size current at that time will be used, but when `and(a,b,-8)' is evaluated, a word size of *-8* will always be used. A symbolic binary function will be left in symbolic form unless the all of its argument(s) are integers or integer-valued floats.

If either or both arguments are modulo forms for which M is a power of two, that power of two is taken as the word size unless a numeric prefix argument overrides it. The current word size is never consulted when modulo-power-of-two forms are involved.

The b a (`calc-and`) [`and`] command computes the bitwise AND of the two numbers on the top of the

stack. In other words, for each of the w binary digits of the two numbers (pairwise), the correspondi of the result is 1 if and only if both input bits are 1: `and(2#1100, 2#1010) = 2#1000'.

The b o (`calc-or`) [`or`] command computes the bitwise inclusive OR of two numbers. A bit is 1 if either of the input bits, or both, are 1: `or(2#1100, 2#1010) = 2#1110'.

The b x (`calc-xor`) [`xor`] command computes the bitwise exclusive OR of two numbers. A bit is 1 if exactly one of the input bits is 1: `xor(2#1100, 2#1010) = 2#0110'.

The b d (`calc-diff`) [`diff`] command computes the bitwise difference of two numbers; this is defined by `diff(a,b) = and(a,not(b))', so that `diff(2#1100, 2#1010) = 2#0100'.

The b n (`calc-not`) [`not`] command computes the bitwise NOT of a number. A bit is 1 if the input bit is 0 and vice-versa.

The b l (`calc-lshift-binary`) [`lsh`] command shifts a number left by one bit, or by the number of bits specified in the numeric prefix argument. A negative prefix argument performs a logical right shift, in which zeros are shifted in on the left. In symbolic form, `lsh(a)' is short for `lsh(a,1)', which in turn is short for `lsh(a,n,w)'. Bits shifted "off the end," according to the current word size, are lost.

The H b l command also does a left shift, but it takes two arguments from the stack (the value to shift, and, at top-of-stack, the number of bits to shift). This version interprets the prefix argument just like the regular binary operations, i.e., as a word size. The Hyperbolic flag has a similar effect on the rest of the binary shift and rotate commands.

The b r (`calc-rshift-binary`) [`rsh`] command shifts a number right by one bit, or by the number of bits specified in the numeric prefix argument: `rsh(a,n) = lsh(a,-n)'.

The b L (`calc-lshift-arith`) [`ash`] command shifts a number left. It is analogous to `lsh`, except that if the shift is rightward (the prefix argument is negative), an arithmetic shift is performed as described below.

The b R (`calc-rshift-arith`) [`rash`] command performs an "arithmetic" shift to the right, in which the leftmost bit (according to the current word size) is duplicated rather than shifting in zeros. This corresponds to dividing by a power of two where the input is interpreted as a signed, twos-complement number. (The distinction between the `rsh' and `rash' operations is totally independent from whether the word size is positive or negative.) With a negative prefix argument, this performs a standard left shift.

The b t (`calc-rotate-binary`) [`rot`] command rotates a number one bit to the left. The leftmost bit (according to the current word size) is dropped off the left and shifted in on the right. With a numeric prefix argument, the number is rotated that many bits to the left or right.

See section Set Operations using Vectors, for the b p and b u commands that pack and unpack binary integers into sets. (For example, b u unpacks the number `2#11001' to the set of bit-numbers `[0, 3, 4]'.) Type b u V # to count the number of "1" bits in a binary integer.

Another interesting use of the set representation of binary integers is to reverse the bits in, say, a 32-bit integer. Type b u to unpack; type 31 TAB - to replace each bit-number in the set with 31 minus that bit-number; type b p to pack the set back into a binary integer.

TOC

Go to the previous, next section.

Go to the previous, next section.

Scientific Functions

The functions described here perform trigonometric and other transcendental calculations. They generally produce floating-point answers correct to the full current precision. The H (Hyperbolic) and I (Inverse) flag keys must be used to get some of these functions from the keyboard.

One miscellanous command is shift-P (`calc-pi`), which pushes the value of @c{π} pi (at the current precision) onto the stack. With the Hyperbolic flag, it pushes the value e, the base of natural logarithms. With the Inverse flag, it pushes Euler's constant @c{γ} gamma (about 0.5772). With both Inverse and Hyperbolic, it pushes the "golden ratio" @c{ϕ} phi (about 1.618). (At present, Euler's constant is not available to unlimited precision; Calc knows only the first 100 digits.) In Symbolic mode, these commands push the actual variables `pi', `e', `gamma', and `phi', respectively, instead of their values; see section Symbolic Mode.

The Q (`calc-sqrt`) [`sqrt`] function is described elsewhere; see section Basic Arithmetic. With the Inverse flag [`sqr`], this command computes the square of the argument.

See section Numeric Prefix Arguments, for a discussion of the effect of numeric prefix arguments on commands in this chapter which do not otherwise interpret a prefix argument.

Logarithmic Functions

The shift-L (`calc-ln`) [`ln`] command computes the natural logarithm of the real or complex number on the top of the stack. With the Inverse flag it computes the exponential function instead, although this is redundant with the E command.

The shift-E (`calc-exp`) [`exp`] command computes the exponential, i.e., e raised to the power of the number on the stack. The meanings of the Inverse and Hyperbolic flags follow from those for the `calc-ln` command.

The H L (`calc-log10`) [`log10`] command computes the common (base-10) logarithm of a number. (With the Inverse flag [`exp10`], it raises ten to a given power.) Note that the common logarithm of a complex number is computed by taking the natural logarithm and dividing by @c{$\ln10$} ln(10).

The B (`calc-log`) [`log`] command computes a logarithm to any base. For example, 1024 RET 2 B produces 10, since 2^10 = 1024. In certain cases like `log(3,9)', the result will be either 1:2 or 0.5 depending on the current Fraction Mode setting. With the Inverse flag [`alog`], this command is similar to ^ except that the order of the arguments is reversed.

The f I (`calc-ilog`) [`ilog`] command computes the integer logarithm of a number to any base. The number and the base must themselves be positive integers. This is the true logarithm, rounded down to an integer. Thus ilog(x,10) is 3 for all x in the range from 1000 to 9999. If both arguments are positive integers, exact integer arithmetic is used; otherwise, this is equivalent to `floor(log(x,b))'.

The f E (`calc-expm1`) [`expm1`] command computes exp(x)-1, but using an algorithm that produces more accurate answer when the result is close to zero, i.e., when @c{e^x} exp(x) is close to one.

The f L (`calc-lnp1`) [`lnp1`] command computes ln(x+1), producing a more accurate answer when x is close to zero.

Trigonometric/Hyperbolic Functions

The shift-S (`calc-sin`) [`sin`] command computes the sine of an angle or complex number. If the input is an HMS form, it is interpreted as degrees-minutes-seconds; otherwise, the input is interpreted according to the current angular mode. It is best to use Radians mode when operating on complex numbers.

Calc's "units" mechanism includes angular units like `deg`, `rad`, and `grad`. While `sin(45 deg)' is not evaluated all the time, the u s (`calc-simplify-units`) command will simplify `sin(45 deg)' by taking the sine of 45 degrees, regardless of the current angular mode. See section Basic Operations on Units.

Also, the symbolic variable `pi` is not ordinarily recognized in arguments to trigonometric functions, as in `sin(3 pi / 4)', but the a s (`calc-simplify`) command recognizes many such formulas when the current angular mode is radians *and* symbolic mode is enabled; this example would be replaced by `sqrt(2) / 2'. See section Symbolic Mode. Beware, this simplification occurs even if you have stored a different value in the variable `pi'; this is one reason why changing built-in variables is a bad idea. Arguments of the form x plus a multiple of @c{$\pi/2$} pi/2 are also simplified. Calc includes similar formulas for `cos` and `tan`.

The a s command knows all angles which are integer multiples of pi/12, @c{$\pi/10$} pi/10, or @c{$\pi/8$} pi/8 radians. In degrees mode, analogous simplifications occur for integer multiples of 15 or 18 degrees, and for arguments plus multiples of 90 degrees.

With the Inverse flag, `calc-sin` computes an arcsine. This is also available as the `calc-arcsin` command or `arcsin` algebraic function. The returned argument is converted to degrees, radians, or HMS notation depending on the current angular mode.

With the Hyperbolic flag, `calc-sin` computes the hyperbolic sine, also available as `calc-sinh` [`sinh`]. With the Hyperbolic and Inverse flags, it computes the hyperbolic arcsine (`calc-arcsinh`) [`arcsinh`].

The shift-C (`calc-cos`) [`cos`] command computes the cosine of an angle or complex number, and shift-T (`calc-tan`) [`tan`] computes the tangent, along with all the various inverse and hyperbolic variants of these functions.

The f T (`calc-arctan2`) [`arctan2`] command takes two numbers from the stack and computes the arc tangent of their ratio. The result is in the full range from *-180* (exclusive) to *+180* (inclusive) degrees, or the analogous range in radians. A similar result would be obtained with / followed by I T, but the value would only be in the range from *-90* to *+90* degrees since the division loses information about the signs of the two components, and an error might result from an explicit division by zero which

`arctan2` would avoid. By (arbitrary) definition, `arctan2(0,0)=0'.

The `calc-sincos` [`sincos`] command computes the sine and cosine of a number, returning them as a vector of the form `[cos, sin]'. With the Inverse flag [`arcsincos`], this command takes a two-element vector as an argument and computes `arctan2` of the elements. (This command does not accept the Hyperbolic flag.)

Advanced Mathematical Functions

Calc can compute a variety of less common functions that arise in various branches of mathematics. All of the functions described in this section allow arbitrary complex arguments and, except as noted, will work to arbitrarily large precisions. They can not at present handle error forms or intervals as arguments.

NOTE: These functions are still experimental. In particular, their accuracy is not guaranteed in all domains. It is advisable to set the current precision comfortably higher than you actually need when using these functions. Also, these functions may be impractically slow for some values of the arguments.

The f g (`calc-gamma`) [`gamma`] command computes the Euler gamma function. For positive integer arguments, this is related to the factorial function: `gamma(n+1) = fact(n)'. For general complex arguments the gamma function can be defined by the following definite integral: @c{$\Gamma(a) = \int_0^\infty t^{a-1} e^t dt$} gamma(a) = integ(t^(a-1) exp(t), t, 0, inf). (The actual implementation uses far more efficient computational methods.)

The f G (`calc-inc-gamma`) [`gammaP`] command computes the incomplete gamma function, denoted `P(a,x)'. This is defined by the integral, @c{$P(a,x) = \left( \int_0^x t^{a-1} e^t dt \right) / \Gamma(a)$} gammaP(a,x) = integ(t^(a-1) exp(t), t, 0, x) / gamma(a). This implies that `gammaP(a,inf) = 1' for any a (see the definition of the normal gamma function).

Several other varieties of incomplete gamma function are defined. The complement of P(a,x), called Q(a,x) = 1-P(a,x) by some authors, is computed by the I f G [`gammaQ`] command. You can think of this as taking the other half of the integral, from x to infinity.

The f b (`calc-beta`) [`beta`] command computes the Euler beta function, which is defined in terms of the gamma function as beta(a,b) = gamma(a) gamma(b) / gamma(a+b), or by beta(a,b) = integ(t^(a-1) (1-t)^(b-1), t, 0, 1).

The f B (`calc-inc-beta`) [`betaI`] command computes the incomplete beta function I(x,a,b). It is defined by betaI(x,a,b) = integ(t^(a-1) (1-t)^(b-1), t, 0, x) / beta(a,b). Once again, the H (hyperbolic) prefix gives the corresponding un-normalized version [`betaB`].

The f e (`calc-erf`) [`erf`] command computes the error function @c{$\hbox{erf}(x) = {2 \over \sqrt{\pi}} \int_0^x e^{-t^2} dt$} erf(x) = 2 integ(exp(-(t^2)), t, 0, x) / sqrt(pi). The complementary error function I f e (`calc-erfc`) [`erfc`] is the corresponding integral from `x' to infinity; the sum erf(x) + erfc(x) = 1.

The f j (`calc-bessel-J`) [`besJ`] and f y (`calc-bessel-Y`) [`besY`] commands compute the Bessel functions of the first and second kinds, respectively. In `besJ(n,x)' and `besY(n,x)' the "order" parameter n is often an integer, but is not required to be one. Calc's implementation of the Bessel

functions currently limits the precision to 8 digits, and may not be exact even to that precision. Use with care!

Branch Cuts and Principal Values

All of the logarithmic, trigonometric, and other scientific functions are defined for complex numbers as well as for reals. This section describes the values returned in cases where the general result is a family of possible values. Calc follows section 12.5.3 of Steele's Common Lisp, the Language, second edition, in these matters. This section will describe each function briefly; for a more detailed discussion (including some nifty diagrams), consult Steele's book.

Note that the branch cuts for `arctan` and `arctanh` were changed between the first and second editions of Steele. Versions of Calc starting with 2.00 follow the second edition.

The new branch cuts exactly match those of the HP-28/48 calculators. They also match those of Mathematica 1.2, except that Mathematica's `arctan` cut is always in the right half of the complex plane, and its `arctanh` cut is always in the top half of the plane. Calc's cuts are continuous with quadrants I and III for `arctan`, or II and IV for `arctanh`.

Note: The current implementations of these functions with complex arguments are designed with proper behavior around the branch cuts in mind, *not* efficiency or accuracy. You may need to increase the floating precision and wait a while to get suitable answers from them.

For `sqrt(a+bi)': When a<0 and b is small but positive or zero, the result is close to the +i axis. For b small and negative, the result is close to the -i axis. The result always lies in the right half of the complex plane.

For `ln(a+bi)': The real part is defined as `ln(abs(a+bi))'. The imaginary part is defined as `arg(a+bi) = arctan2(b,a)'. Thus the branch cuts for `sqrt` and `ln` both lie on the negative real axis.

The following table describes these branch cuts in another way. If the real and imaginary parts of z are as shown, then the real and imaginary parts of f(z) will be as shown. Here `eps` stands for a small positive value; each occurrence of `eps` may stand for a different small value.

```
     z           sqrt(z)       ln(z)
-----------------------------------------
   +,   0         +,  0       any, 0
   -,   0         0,  +       any, pi
   -, +eps      +eps, +      +eps, +
   -, -eps      +eps, -      +eps, -
```

For `z1^z2': This is defined by `exp(ln(z1)*z2)'. One interesting consequence of this is that `(-8)^1:3' does not evaluate to *-2* as you might expect, but to the complex number (1., 1.732). Both of these are valid cube roots of *-8* (as is (1., -1.732)); Calc chooses a perhaps less-obvious root for the sake of mathematical consistency.

For `arcsin(z)': This is defined by `-i*ln(i*z + sqrt(1-z^2))'. The branch cuts are on the real axis, less than

-1 and greater than 1.

For `arccos(z)': This is defined by `-i*ln(z + i*sqrt(1-z^2))', or equivalently by `pi/2 - arcsin(z)'. The branch cuts are on the real axis, less than *-1* and greater than 1.

For `arctan(z)': This is defined by `(ln(1+i*z) - ln(1-i*z)) / (2*i)'. The branch cuts are on the imaginary axis, below -i and above i.

For `arcsinh(z)': This is defined by `ln(z + sqrt(1+z^2))'. The branch cuts are on the imaginary axis, below -i and above i.

For `arccosh(z)': This is defined by `ln(z + (z+1)*sqrt((z-1)/(z+1)))'. The branch cut is on the real axis less than 1.

For `arctanh(z)': This is defined by `(ln(1+z) - ln(1-z)) / 2'. The branch cuts are on the real axis, less than *-1* and greater than 1.

The following tables for `arcsin`, `arccos`, and `arctan` assume the current angular mode is radians. The hyperbolic functions operate independently of the angular mode.

z	arcsin(z)	arccos(z)
(-1..1), 0	(-pi/2..pi/2), 0	(0..pi), 0
(-1..1), +eps	(-pi/2..pi/2), +eps	(0..pi), -eps
(-1..1), -eps	(-pi/2..pi/2), -eps	(0..pi), +eps
<-1, 0	-pi/2, +	pi, -
<-1, +eps	-pi/2 + eps, +	pi - eps, -
<-1, -eps	-pi/2 + eps, -	pi - eps, +
>1, 0	pi/2, -	0, +
>1, +eps	pi/2 - eps, +	+eps, -
>1, -eps	pi/2 - eps, -	+eps, +

z	arccosh(z)	arctanh(z)
(-1..1), 0	0, (0..pi)	any, 0
(-1..1), +eps	+eps, (0..pi)	any, +eps
(-1..1), -eps	+eps, (-pi..0)	any, -eps
<-1, 0	+, pi	-, pi/2
<-1, +eps	+, pi - eps	-, pi/2 - eps
<-1, -eps	+, -pi + eps	-, -pi/2 + eps
>1, 0	+, 0	+, -pi/2
>1, +eps	+, +eps	+, pi/2 - eps
>1, -eps	+, -eps	+, -pi/2 + eps

z	arcsinh(z)	arctan(z)
0, (-1..1)	0, (-pi/2..pi/2)	0, any


```
  0,   <-1      -,    -pi/2          -pi/2,    -
+eps,  <-1      +, -pi/2 + eps      pi/2 - eps, -
-eps,  <-1      -, -pi/2 + eps     -pi/2 + eps, -
  0,    >1      +,     pi/2           pi/2,    +
+eps,   >1      +,  pi/2 - eps      pi/2 - eps, +
-eps,   >1      -,  pi/2 - eps     -pi/2 + eps, +
```

Finally, the following identities help to illustrate the relationship between the complex trigonometric and hyperbolic functions. They are valid everywhere, including on the branch cuts.

```
sin(i*z)  = i*sinh(z)       arcsin(i*z)  = i*arcsinh(z)
cos(i*z)  =   cosh(z)       arcsinh(i*z) = i*arcsin(z)
tan(i*z)  = i*tanh(z)       arctan(i*z)  = i*arctanh(z)
sinh(i*z) = i*sin(z)        cosh(i*z)    =   cos(z)
```

The "advanced math" functions (gamma, Bessel, etc.) are also defined for general complex arguments, but their branch cuts and principal values are not rigorously specified at present.

Random Numbers

The k r (`calc-random`) [`random`] command produces random numbers of various sorts.

Given a positive numeric prefix argument M, it produces a random integer N in the range @c{$0 \le N < M$} 0 <= N < M. Each of the M values appears with equal probability.

With no numeric prefix argument, the k r command takes its argument from the stack instead. Once again, if this is a positive integer M the result is a random integer less than M. However, note that while numeric prefix arguments are limited to six digits or so, an M taken from the stack can be arbitrarily large. If M is negative, the result is a random integer in the range @c{$M < N \le 0$} M < N <= 0.

If the value on the stack is a floating-point number M, the result is a random floating-point number N in the range @c{$0 \le N < M$} 0 <= N < M or @c{$M < N \le 0$} M < N <= 0, according to the sign of M.

If M is zero, the result is a Gaussian-distributed random real number; the distribution has a mean of zero and a standard deviation of one. The algorithm used generates random numbers in pairs; thus, every other call to this function will be especially fast.

If M is an error form @c{m +/- σ} `m +/- s' where *m* and @c{σ} *s* are both real numbers, the result uses a Gaussian distribution with mean *m* and standard deviation @c{σ} *s*.

If M is an interval form, the lower and upper bounds specify the acceptable limits of the random numbers. If both bounds are integers, the result is a random integer in the specified range. If either bound is floating-point, the result is a random real number in the specified range. If the interval is open at either end, the result will be sure not to equal that end value. (This makes a big difference for integer intervals, but for floating-point intervals it's relatively minor: with a precision of 6, `random([1.0..2.0))' will return any of one million numbers from 1.00000 to 1.99999; `random([1.0..2.0])' may additionally return

2.00000, but the probability of this happening is extremely small.)

If M is a vector, the result is one element taken at random from the vector. All elements of the vector are given equal probabilities.

The sequence of numbers produced by k r is completely random by default, i.e., the sequence is seeded each time you start Calc using the current time and other information. You can get a reproducible sequence by storing a particular "seed value" in the Calc variable `RandSeed`. Any integer will do for a seed; integers of from 1 to 12 digits are good. If you later store a different integer into `RandSeed`, Calc will switch to a different pseudo-random sequence. If you "unstore" `RandSeed`, Calc will re-seed itself from the current time. If you store the same integer that you used before back into `RandSeed`, you will get the exact same sequence of random numbers as before.

The `calc-rrandom` command (not on any key) produces a random real number between zero and one. It is equivalent to `random(1.0)'.

The k a (`calc-random-again`) command produces another random number, re-using the most recent value of M. With a numeric prefix argument n, it produces n more random numbers using that value of M.

The k h (`calc-shuffle`) command produces a vector of several random values with no duplicates. The value on the top of the stack specifies the set from which the random values are drawn, and may be any of the M formats described above. The numeric prefix argument gives the length of the desired list. (If you do not provide a numeric prefix argument, the length of the list is taken from the top of the stack, and M from second-to-top.)

If M is a floating-point number, zero, or an error form (so that the random values are being drawn from the set of real numbers) there is little practical difference between using k h and using k r several times. But if the set of possible values consists of just a few integers, or the elements of a vector, then there is a very real chance that multiple k r's will produce the same number more than once. The k h command produces a vector whose elements are always distinct. (Actually, there is a slight exception: If M is a vector, no given vector element will be drawn more than once, but if several elements of M are equal, they may each make it into the result vector.)

One use of k h is to rearrange a list at random. This happens if the prefix argument is equal to the number of values in the list: [1, 1.5, 2, 2.5, 3] 5 k h might produce the permuted list `[2.5, 1, 1.5, 3, 2]'. As a convenient feature, if the argument n is negative it is replaced by the size of the set represented by M. Naturally, this is allowed only when M specifies a small discrete set of possibilities.

To do the equivalent of k h but with duplications allowed, given M on the stack and with n just entered as a numeric prefix, use v b to build a vector of copies of M, then use V M k r to "map" the normal k r function over the elements of this vector. See section Vector/Matrix Functions.

Random Number Generator

Calc's random number generator uses several methods to ensure that the numbers it produces are highly random. Knuth's *Art of Computer Programming*, Volume II, contains a thorough description of the theory of random number generators and their measurement and characterization.

If `RandSeed` has no stored value, Calc calls Emacs' built-in `random` function to get a stream of random numbers, which it then treats in various ways to avoid problems inherent in the simple random number generators that many systems use to implement `random`.

When Calc's random number generator is first invoked, it "seeds" the low-level random sequence using the time of day, so that the random number sequence will be different every time you use Calc.

Since Emacs Lisp doesn't specify the range of values that will be returned by its `random` function, Calc exercises the function several times to estimate the range. When Calc subsequently uses the `random` function, it takes only 10 bits of the result near the most-significant end. (It avoids at least the bottom four bits, preferably more, and also tries to avoid the top two bits.) This strategy works well with the linear congruential generators that are typically used to implement `random`.

If `RandSeed` contains an integer, Calc uses this integer to seed an "additive congruential" method (Knuth's algorithm 3.2.2A, computing @c{$X_{n-55} - X_{n-24}$} X_n-55 - X_n-24). This method expands the seed value into a large table which is maintained internally; the variable `RandSeed` is changed from, e.g., 42 to the vector [42] to indicate that the seed has been absorbed into this table. When `RandSeed` contains a vector, k r and related commands continue to use the same internal table as last time. There is no way to extract the complete state of the random number generator so that you can restart it from any point; you can only restart it from the same initial seed value. A simple way to restart from the same seed is to type s r RandSeed to get the seed vector, v u to unpack it back into a number, then s t RandSeed to reseed the generator with that number.

Calc uses a "shuffling" method as described in algorithm 3.2.2B of Knuth. It fills a table with 13 random 10-bit numbers. Then, to generate a new random number, it uses the previous number to index into the table, picks the value it finds there as the new random number, then replaces that table entry with a new value obtained from a call to the base random number generator (either the additive congruential generator or the `random` function supplied by the system). If there are any flaws in the base generator, shuffling will tend to even them out. But if the system provides an excellent `random` function, shuffling will not damage its randomness.

To create a random integer of a certain number of digits, Calc builds the integer three decimal digits at a time. For each group of three digits, Calc calls its 10-bit shuffling random number generator (which returns a value from 0 to 1023); if the random value is 1000 or more, Calc throws it out and tries again until it gets a suitable value.

To create a random floating-point number with precision p, Calc simply creates a random p-digit integer and multiplies by 10^-p. The resulting random numbers should be very clean, but note that relatively small numbers will have few significant random digits. In other words, with a precision of 12, you will occasionally get numbers on the order of @c{10^{-9}} 10^-9 or @c{10^{-10}} 10^-10, but those numbers will only have two or three random digits since they correspond to small integers times @c{10^{-12}} 10^-12.

To create a random integer in the interval `[0 .. m)', Calc counts the digits in m, creates a random integer with three additional digits, then reduces modulo m. Unless m is a power of ten the resulting values will be very slightly biased toward the lower numbers, but this bias will be less than 0.1%. (For example, if m is 42, Calc will reduce a random integer less than 100000 modulo 42 to get a result less than 42. It is easy to show that the numbers 40 and 41 will be only 2380/2381 as likely to result from this modulo operation

as numbers 39 and below.) If m is a power of ten, however, the numbers should be completely unbia

The Gaussian random numbers generated by `random(0.0)' use the "polar" method described in Knuth section 3.4.1C. This method generates a pair of Gaussian random numbers at a time, so only every other call to `random(0.0)' will require significant calculations.

Combinatorial Functions

Commands relating to combinatorics and number theory begin with the k key prefix.

The k g (`calc-gcd`) [`gcd`] command computes the Greatest Common Divisor of two integers. It also accepts fractions; the GCD of two fractions is defined by taking the GCD of the numerators, and the LCM of the denominators. This definition is consistent with the idea that `a / gcd(a,x)' should yield an integer for any `a' and `x'. For other types of arguments, the operation is left in symbolic form.

The k l (`calc-lcm`) [`lcm`] command computes the Least Common Multiple of two integers or fractions. The product of the LCM and GCD of two numbers is equal to the product of the numbers.

The k E (`calc-extended-gcd`) [`egcd`] command computes the GCD of two integers x and y and returns a vector [g, a, b] where @c{$g = \gcd(x,y) = a x + b y$} g = gcd(x,y) = a x + b y.

The ! (`calc-factorial`) [`fact`] command computes the factorial of the number at the top of the stack. If the number is an integer, the result is an exact integer. If the number is an integer-valued float, the result is a floating-point approximation. If the number is a non-integral real number, the generalized factorial is used, as defined by the Euler Gamma function. Please note that computation of large factorials can be slow; using floating-point format will help since fewer digits must be maintained. The same is true of many of the commands in this section.

The k d (`calc-double-factorial`) [`dfact`] command computes the "double factorial" of an integer. For an even integer, this is the product of even integers from 2 to N. For an odd integer, this is the product of odd integers from 3 to N. If the argument is an integer-valued float, the result is a floating-point approximation. This function is undefined for negative even integers. The notation N!! is also recognized for double factorials.

The k c (`calc-choose`) [`choose`] command computes the binomial coefficient N-choose-M, where M is the number on the top of the stack and N is second-to-top. If both arguments are integers, the result is an exact integer. Otherwise, the result is a floating-point approximation. The binomial coefficient is defined for all real numbers by @c{$N! \over M! (N-M)!\,$} N! / M! (N-M)!.

The k b (`calc-bernoulli-number`) [`bern`] command computes a given Bernoulli number. The value at the top of the stack is a nonnegative integer n that specifies which Bernoulli number is desired. The H k b command computes a Bernoulli polynomial, taking n from the second-to-top position and x from the top of the stack. If x is a variable or formula the result is a polynomial in x; if x is a number the result is a number.

The k e (`calc-euler-number`) [`euler`] command similarly computes an Euler number, and H k e computes an Euler polynomial. Bernoulli and Euler numbers occur in the Taylor expansions of several functions.

The k s (`calc-stirling-number`) [`stir1`] command computes a Stirling number of the first kind@c{ $n \brack m$} , given two integers n and m on the stack. The H k s [`stir2`] command computes a Stirling number of the second kind@c{ $n \brace m$} . These are the number of m-cycle permutations of n objects, and the number of ways to partition n objects into m non-empty sets, respectively.

The k p (`calc-prime-test`) command checks if the integer on the top of the stack is prime. For integers less than eight million, the answer is always exact and reasonably fast. For larger integers, a probabilistic method is used (see Knuth vol. II, section 4.5.4, algorithm P). The number is first checked against small prime factors (up to 13). Then, any number of iterations of the algorithm are performed. Each step either discovers that the number is non-prime, or substantially increases the certainty that the number is prime. After a few steps, the chance that a number was mistakenly described as prime will be less than one percent. (Indeed, this is a worst-case estimate of the probability; in practice even a single iteration is quite reliable.) After the k p command, the number will be reported as definitely prime or non-prime if possible, or otherwise "probably" prime with a certain probability of error.

The normal k p command performs one iteration of the primality test. Pressing k p repeatedly for the same integer will perform additional iterations. Also, k p with a numeric prefix performs the specified number of iterations. There is also an algebraic function `prime(n)' or `prime(n,iters)' which returns 1 if n is (probably) prime and 0 if not.

The k f (`calc-prime-factors`) [`prfac`] command attempts to decompose an integer into its prime factors. For numbers up to 25 million, the answer is exact although it may take some time. The result is a vector of the prime factors in increasing order. For larger inputs, prime factors above 5000 may not be found, in which case the last number in the vector will be an unfactored integer greater than 25 million (with a warning message). For negative integers, the first element of the list will be *-1*. For inputs *-1*, *0*, and *1*, the result is a list of the same number.

The k n (`calc-next-prime`) [`nextprime`] command finds the next prime above a given number. Essentially, it searches by calling `calc-prime-test` on successive integers until it finds one that passes the test. This is quite fast for integers less than eight million, but once the probabilistic test comes into play the search may be rather slow. Ordinarily this command stops for any prime that passes one iteration of the primality test. With a numeric prefix argument, a number must pass the specified number of iterations before the search stops. (This only matters when searching above eight million.) You can always use additional k p commands to increase your certainty that the number is indeed prime.

The I k n (`calc-prev-prime`) [`prevprime`] command analogously finds the next prime less than a given number.

The k t (`calc-totient`) [`totient`] command computes the Euler "totient" function@c{ $\phi(n)$} , the number of integers less than n which are relatively prime to n.

The k m (`calc-moebius`) [`moebius`] command computes the Moebius "mu" function. If the input number is a product of k distinct factors, this is (-1)^k. If the input number has any duplicate factors (i.e., can be divided by the same prime more than once), the result is zero.

Probability Distribution Functions

The functions in this section compute various probability distributions. For continuous distributions, this is the integral of the probability density function from x to infinity. (These are the "upper tail" distribution functions; there are also corresponding "lower tail" functions which integrate from minus infinity to x.) For discrete distributions, the upper tail function gives the sum from x to infinity; the lower tail function gives the sum from minus infinity up to, but not including, x.

To integrate from x to y, just use the distribution function twice and subtract. For example, the probability that a Gaussian random variable with mean 2 and standard deviation 1 will lie in the range from 2.5 to 2.8 is `utpn(2.5,2,1) - utpn(2.8,2,1)' ("the probability that it is greater than 2.5, but not greater than 2.8"), or equivalently `ltpn(2.8,2,1) - ltpn(2.5,2,1)'.

The k B (`calc-utpb`) [`utpb`] function uses the binomial distribution. Push the parameters n, p, and then x onto the stack; the result (`utpb(x,n,p)') is the probability that an event will occur x or more times out of n trials, if its probability of occurring in any given trial is p. The I k B [`ltpb`] function is the probability that the event will occur fewer than x times.

The other probability distribution functions similarly take the form k X (`calc-utpx`) [`utpx`] and I k X [`ltpx`], for various letters x. The arguments to the algebraic functions are the value of the random variable first, then whatever other parameters define the distribution. Note these are among the few Calc functions where the order of the arguments in algebraic form differs from the order of arguments as found on the stack. (The random variable comes last on the stack, so that you can type, e.g., 2 RET 1 RET 2.5 k N M-RET DEL 2.8 k N -, using M-RET DEL to recover the original arguments but substitute a new value for x.)

The `utpc(x,v)' function uses the chi-square distribution with v degrees of freedom. It is the probability that a model is correct if its chi-square statistic is x.

The `utpf(F,v1,v2)' function uses the F distribution, used in various statistical tests. The parameters @c{ν_1} v1 and @c{ν_2} v2 are the degrees of freedom in the numerator and denominator, respectively, used in computing the statistic F.

The `utpn(x,m,s)' function uses a normal (Gaussian) distribution with mean m and standard deviation @c{σ} s. It is the probability that such a normal-distributed random variable would exceed x.

The `utpp(n,x)' function uses a Poisson distribution with mean x. It is the probability that n or more such Poisson random events will occur.

The `utpt(t,v)' function uses the Student's "t" distribution with @c{ν} v degrees of freedom. It is the probability that a t-distributed random variable will be greater than t. (Note: This computes the distribution function @c{$A(t|\nu)$} A(t|v) where @c{$A(0|\nu) = 1$} A(0|v) = 1 and @c{$A(\infty|\nu) \to 0$} A(inf|v) -> 0. The `UTPT` operation on the HP-48 uses a different definition which returns half of Calc's value: `UTPT(t,v) = .5*utpt(t,v)'.)

While Calc does not provide inverses of the probability distribution functions, the a R command can be used to solve for the inverse. Since the distribution functions are monotonic, a R is guaranteed to be able to find a solution given any initial guess. See section Numerical Solutions.

Go to the previous, next section.

Go to the previous, next section.

Vector/Matrix Functions

Many of the commands described here begin with the v prefix. (For convenience, the shift-V prefix is equivalent to v.) The commands usually apply to both plain vectors and matrices; some apply only to matrices or only to square matrices. If the argument has the wrong dimensions the operation is left in symbolic form.

Vectors are entered and displayed using `[a,b,c]' notation. Matrices are vectors of which all elements are vectors of equal length. (Though none of the standard Calc commands use this concept, a three-dimensional matrix or rank-3 tensor could be defined as a vector of matrices, and so on.)

Packing and Unpacking

Calc's "pack" and "unpack" commands collect stack entries to build composite objects such as vectors and complex numbers. They are described in this chapter because they are most often used to build vectors.

The v p (`calc-pack`) [`pack`] command collects several elements from the stack into a matrix, complex number, HMS form, error form, etc. It uses a numeric prefix argument to specify the kind of object to be built; this argument is referred to as the "packing mode." If the packing mode is a nonnegative integer, a vector of that length is created. For example, C-u 5 v p will pop the top five stack elements and push back a single vector of those five elements. (C-u 0 v p simply creates an empty vector.)

The same effect can be had by pressing [to push an incomplete vector on the stack, using TAB (`calc-roll-down`) to sneak the incomplete object up past a certain number of elements, and then pressing] to complete the vector.

Negative packing modes create other kinds of composite objects:

-1

Two values are collected to build a complex number. For example, 5 RET 7 C-u -1 v p creates the complex number (5, 7). The result is always a rectangular complex number. The two input values must both be real numbers, i.e., integers, fractions, or floats. If they are not, Calc will instead build a formula like `a + (0, 1) b'. (The other packing modes also create a symbolic answer if the components are not suitable.)

-2

Two values are collected to build a polar complex number. The first is the magnitude; the second is the phase expressed in either degrees or radians according to the current angular mode.

-3

Three values are collected into an HMS form. The first two values (hours and minutes) must be integers or integer-valued floats. The third value may be any real number.

-4

Two values are collected into an error form. The inputs may be real numbers or formulas.

-5

Two values are collected into a modulo form. The inputs must be real numbers.

-6

Two values are collected into the interval `[a .. b]'. The inputs may be real numbers, HMS or date forms, or formulas.

-7

Two values are collected into the interval `[a .. b)'.

-8

Two values are collected into the interval `(a .. b]'.

-9

Two values are collected into the interval `(a .. b)'.

-10

Two integer values are collected into a fraction.

-11

Two values are collected into a floating-point number. The first is the mantissa; the second, which must be an integer, is the exponent. The result is the mantissa times ten to the power of the exponent.

-12

This is treated the same as *-11* by the v p command. When unpacking, *-12* specifies that a floating-point mantissa is desired.

-13

A real number is converted into a date form.

-14

Three numbers (year, month, day) are packed into a pure date form.

-15

Six numbers are packed into a date/time form.

With any of the two-input negative packing modes, either or both of the inputs may be vectors. If both are vectors of the same length, the result is another vector made by packing corresponding elements of the input vectors. If one input is a vector and the other is a plain number, the number is packed along with each vector element to produce a new vector. For example, C-u -4 v p could be used to convert a vector of numbers and a vector of errors into a single vector of error forms; C-u -5 v p could convert a vector of numbers and a single number M into a vector of numbers modulo M.

If you don't give a prefix argument to v p, it takes the packing mode from the top of the stack. The elements to be packed then begin at stack level 2. Thus 1 RET 2 RET 4 n v p is another way to enter the error form `1 +/- 2'.

If the packing mode taken from the stack is a vector, the result is a matrix with the dimensions specified

by the elements of the vector, which must each be integers. For example, if the packing mode is `[2, 3]' then six numbers will be taken from the stack and returned in the form `[[a, b, c], [d, e, f]]'.

If any elements of the vector are negative, other kinds of packing are done at that level as described above. For example, `[2, 3, -4]' takes 12 objects and creates a 2x3 matrix of error forms: `[[a +/- b, c +/- d ...]]'. Also, `[-4, -10]' will convert four integers into an error form consisting of two fractions: `a:b +/- c:d'.

There is an equivalent algebraic function, `pack(mode, items)' where mode is a packing mode (an integer or a vector of integers) and items is a vector of objects to be packed (re-packed, really) according to that mode. For example, `pack([3, -4], [a,b,c,d,e,f])' yields `[a +/- b, c +/- d, e +/- f]'. The function is left in symbolic form if the packing mode is illegal, or if the number of data items does not match the number of items required by the mode.

The v u (`calc-unpack`) command takes the vector, complex number, HMS form, or other composite object on the top of the stack and "unpacks" it, pushing each of its elements onto the stack as separate objects. Thus, it is the "inverse" of v p. If the value at the top of the stack is a formula, v u unpacks it by pushing each of the arguments of the top-level operator onto the stack.

You can optionally give a numeric prefix argument to v u to specify an explicit (un)packing mode. If the packing mode is negative and the input is actually a vector or matrix, the result will be two or more similar vectors or matrices of the elements. For example, given the vector `[a +/- b, c^2, d +/- 7]', the result of C-u -4 v u will be the two vectors `[a, c^2, d]' and `[b, 0, 7]'.

Note that the prefix argument can have an effect even when the input is not a vector. For example, if the input is the number *-5*, then c-u -1 v u yields *-5* and 0 (the components of *-5* when viewed as a rectangular complex number); C-u -2 v u yields 5 and 180 (assuming degrees mode); and C-u -10 v u yields *-5* and 1 (the numerator and denominator of *-5*, viewed as a rational number). Plain v u with this input would complain that the input is not a composite object.

Unpacking mode *-11* converts a float into an integer mantissa and an integer exponent, where the mantissa is not divisible by 10 (except that 0.0 is represented by a mantissa and exponent of 0). Unpacking mode *-12* converts a float into a floating-point mantissa and integer exponent, where the mantissa (for non-zero numbers) is guaranteed to lie in the range [1 .. 10). In both cases, the mantissa is shifted left or right (and the exponent adjusted to compensate) in order to satisfy these constraints.

Positive unpacking modes are treated differently than for v p. A mode of 1 is much like plain v u with no prefix argument, except that in addition to the components of the input object, a suitable packing mode to re-pack the object is also pushed. Thus, C-u 1 v u followed by v p will re-build the original object.

A mode of 2 unpacks two levels of the object; the resulting re-packing mode will be a vector of length 2. This might be used to unpack a matrix, say, or a vector of error forms. Higher unpacking modes unpack the input even more deeply.

There are two algebraic functions analogous to v u. The `unpack(mode, item)' function unpacks the item using the given mode, returning the result as a vector of components. Here the mode must be an integer, not a vector. For example, `unpack(-4, a +/- b)' returns `[a, b]', as does `unpack(1, a +/- b)'.

The `unpackt` function is like `unpack` but instead of returning a simple vector of items, it returns a

vector of two things: The mode, and the vector of items. For example, `unpackt(1, 2:3 +/- 1:4)' returns `[-4, [2:3, 1:4]]', and `unpackt(2, 2:3 +/- 1:4)' returns `[[-4, -10], [2, 3, 1, 4]]'. The identity for re-building the original object is `apply(pack, unpackt(n, x)) = x'. (The `apply` function builds a function call given the function name and a vector of arguments.)

Subscript notation is a useful way to extract a particular part of an object. For example, to get the numerator of a rational number, you can use `unpack(-10, x)_1'.

Building Vectors

Vectors and matrices can be added, subtracted, multiplied, and divided; see section Basic Arithmetic.

The | (`calc-concat`) command "concatenates" two vectors into one. For example, after [1 , 2] [3 , 4] |, the stack will contain the single vector `[1, 2, 3, 4]'. If the arguments are matrices, the rows of the first matrix are concatenated with the rows of the second. (In other words, two matrices are just two vectors of row-vectors as far as | is concerned.)

If either argument to | is a scalar (a non-vector), it is treated like a one-element vector for purposes of concatenation: 1 [2 , 3] | produces the vector `[1, 2, 3]'. Likewise, if one argument is a matrix and the other is a plain vector, the vector is treated as a one-row matrix.

The H | (`calc-append`) [`append`] command concatenates two vectors without any special cases. Both inputs must be vectors. Whether or not they are matrices is not taken into account. If either argument is a scalar, the `append` function is left in symbolic form. See also `cons` and `rcons` below.

The I | and H I | commands are similar, but they use their two stack arguments in the opposite order. Thus I | is equivalent to TAB |, but possibly more convenient and also a bit faster.

The v d (`calc-diag`) [`diag`] function builds a diagonal square matrix. The optional numeric prefix gives the number of rows and columns in the matrix. If the value at the top of the stack is a vector, the elements of the vector are used as the diagonal elements; the prefix, if specified, must match the size of the vector. If the value on the stack is a scalar, it is used for each element on the diagonal, and the prefix argument is required.

To build a constant square matrix, e.g., a @c{3×3} 3x3 matrix filled with ones, use 0 M-3 v d 1 +, i.e., build a zero matrix first and then add a constant value to that matrix. (Another alternative would be to use v b and v a; see below.)

The v i (`calc-ident`) [`idn`] function builds an identity matrix of the specified size. It is a convenient form of v d where the diagonal element is always one. If no prefix argument is given, this command prompts for one.

In algebraic notation, `idn(a,n)' acts much like `diag(a,n)', except that a is required to be a scalar (non-vector) quantity. If n is omitted, `idn(a)' represents a times an identity matrix of unknown size. Calc can operate algebraically on such generic identity matrices, and if one is combined with a matrix whose size is known, it is converted automatically to an identity matrix of a suitable matching size. The v i command with an argument of zero creates a generic identity matrix, `idn(1)'. Note that in dimensioned matrix mode (see section Matrix and Scalar Modes), generic identity matrices are immediately expanded

to the current default dimensions.

The v x (`calc-index`) [`index`] function builds a vector of consecutive integers from 1 to n, where n is the numeric prefix argument. If you do not provide a prefix argument, you will be prompted to enter a suitable number. If n is negative, the result is a vector of negative integers from n to *-1*.

With a prefix argument of just C-u, the v x command takes three values from the stack: n, start, and incr (with incr at top-of-stack). Counting starts at start and increases by incr for successive vector elements. If start or n is in floating-point format, the resulting vector elements will also be floats. Note that start and incr may in fact be any kind of numbers or formulas.

When start and incr are specified, a negative n has a different interpretation: It causes a geometric instead of arithmetic sequence to be generated. For example, `index(-3, a, b)' produces `[a, a b, a b^2]'. If you omit incr in the algebraic form, `index(n, start)', the default value for incr is one for positive n or two for negative n.

The v b (`calc-build-vector`) [`cvec`] function builds a vector of n copies of the value on the top of the stack, where n is the numeric prefix argument. In algebraic formulas, `cvec(x,n,m)' can also be used to build an n-by-m matrix of copies of x. (Interactively, just use v b twice: once to build a row, then again to build a matrix of copies of that row.)

The v h (`calc-head`) [`head`] function returns the first element of a vector. The I v h (`calc-tail`) [`tail`] function returns the vector with its first element removed. In both cases, the argument must be a non-empty vector.

The v k (`calc-cons`) [`cons`] function takes a value h and a vector t from the stack, and produces the vector whose head is h and whose tail is t. This is similar to |, except if h is itself a vector, | will concatenate the two vectors whereas `cons` will insert h at the front of the vector t.

Each of these three functions also accepts the Hyperbolic flag [`rhead`, `rtail`, `rcons`] in which case t instead represents the *last* single element of the vector, with h representing the remainder of the vector. Thus the vector `[a, b, c, d] = cons(a, [b, c, d]) = rcons([a, b, c], d)'. Also, `head([a, b, c, d]) = a', `tail([a, b, c, d]) = [b, c, d]', `rhead([a, b, c, d]) = [a, b, c]', and `rtail([a, b, c, d]) = d'.

Extracting Vector Elements

The v r (`calc-mrow`) [`mrow`] command extracts one row of the matrix on the top of the stack, or one element of the plain vector on the top of the stack. The row or element is specified by the numeric prefix argument; the default is to prompt for the row or element number. The matrix or vector is replaced by the specified row or element in the form of a vector or scalar, respectively.

With a prefix argument of C-u only, v r takes the index of the element or row from the top of the stack, and the vector or matrix from the second-to-top position. If the index is itself a vector of integers, the result is a vector of the corresponding elements of the input vector, or a matrix of the corresponding rows of the input matrix. This command can be used to obtain any permutation of a vector.

With C-u, if the index is an interval form with integer components, it is interpreted as a range of indices and the corresponding subvector or submatrix is returned.

Subscript notation in algebraic formulas (`a_b') stands for the Calc function `subscr`, which is synonymous with `mrow`. Thus, `[x, y, z]_k' produces x, y, or z if k is one, two, or three, respectively. A double subscript (`M_i_j', equivalent to `subscr(subscr(M, i), j)') will access the element at row i, column j of a matrix. The a _ (`calc-subscript`) command creates a subscript formula `a_b' out of two stack entries. (It is on the a "algebra" prefix because subscripted variables are often used purely as an algebraic notation.)

Given a negative prefix argument, v r instead deletes one row or element from the matrix or vector on the top of the stack. Thus C-u 2 v r replaces a matrix with its second row, but C-u -2 v r replaces the matrix with the same matrix with its second row removed. In algebraic form this function is called `mrrow`.

Given a prefix argument of zero, v r extracts the diagonal elements of a square matrix in the form of a vector. In algebraic form this function is called `getdiag`.

The v c (`calc-mcol`) [`mcol` or `mrcol`] command is the analogous operation on columns of a matrix. Given a plain vector it extracts (or removes) one element, just like v r. If the index in C-u v c is an interval or vector and the argument is a matrix, the result is a submatrix with only the specified columns retained (and possibly permuted in the case of a vector index).

To extract a matrix element at a given row and column, use v r to extract the row as a vector, then v c to extract the column element from that vector. In algebraic formulas, it is often more convenient to use subscript notation: `m_i_j' gives row i, column j of matrix m.

The v s (`calc-subvector`) [`subvec`] command extracts a subvector of a vector. The arguments are the vector, the starting index, and the ending index, with the ending index in the top-of-stack position. The starting index indicates the first element of the vector to take. The ending index indicates the first element *past* the range to be taken. Thus, `subvec([a, b, c, d, e], 2, 4)' produces the subvector `[b, c]'. You could get the same result using `mrow([a, b, c, d, e], [2 .. 4))'.

If either the start or the end index is zero or negative, it is interpreted as relative to the end of the vector. Thus `subvec([a, b, c, d, e], 2, -2)' also produces `[b, c]'. In the algebraic form, the end index can be omitted in which case it is taken as zero, i.e., elements from the starting element to the end of the vector are used. The infinity symbol, `inf`, also has this effect when used as the ending index.

With the Inverse flag, I v s [`rsubvec`] removes a subvector from a vector. The arguments are interpreted the same as for the normal v s command. Thus, `rsubvec([a, b, c, d, e], 2, 4)' produces `[a, d, e]'. It is always true that `subvec` and `rsubvec` return complementary parts of the input vector.

See section Selecting Sub-Formulas, for an alternative way to operate on vectors one element at a time.

Manipulating Vectors

The v l (`calc-vlength`) [`vlen`] command computes the length of a vector. The length of a non-vector is considered to be zero. Note that matrices are just vectors of vectors for the purposes of this command.

With the Hyperbolic flag, H v l [`mdims`] computes a vector of the dimensions of a vector, matrix, or higher-order object. For example, `mdims([[a,b,c],[d,e,f]])' returns `[2, 3]' since its argument is a

@c{2×3} 2x3 matrix.

The v f (`calc-vector-find`) [`find`] command searches along a vector for the first element equal to a given target. The target is on the top of the stack; the vector is in the second-to-top position. If a match is found, the result is the index of the matching element. Otherwise, the result is zero. The numeric prefix argument, if given, allows you to select any starting index for the search.

The v a (`calc-arrange-vector`) [`arrange`] command rearranges a vector to have a certain number of columns and rows. The numeric prefix argument specifies the number of columns; if you do not provide an argument, you will be prompted for the number of columns. The vector or matrix on the top of the stack is flattened into a plain vector. If the number of columns is nonzero, this vector is then formed into a matrix by taking successive groups of n elements. If the number of columns does not evenly divide the number of elements in the vector, the last row will be short and the result will not be suitable for use as a matrix. For example, with the matrix `[[1, 2], [3, 4]]' on the stack, v a 4 produces `[[1, 2, 3, 4]]' (a @c{1×4} 1x4 matrix), v a 1 produces `[[1], [2], [3], [4]]' (a @c{$4\times1$} 4x1 matrix), v a 2 produces `[[1, 2], [3, 4]]' (the original @c{2×2} 2x2 matrix), v a 3 produces `[[1, 2, 3], [4]]' (not a matrix), and v a 0 produces the flattened list `[1, 2, 3, 4]'.

The V S (`calc-sort`) [`sort`] command sorts the elements of a vector into increasing order. Real numbers, real infinities, and constant interval forms come first in this ordering; next come other kinds of numbers, then variables (in alphabetical order), then finally come formulas and other kinds of objects; these are sorted according to a kind of lexicographic ordering with the useful property that one vector is less or greater than another if the first corresponding unequal elements are less or greater, respectively. Since quoted strings are stored by Calc internally as vectors of ASCII character codes (see section Strings), this means vectors of strings are also sorted into alphabetical order by this command.

The I V S [`rsort`] command sorts a vector into decreasing order.

The V G (`calc-grade`) [`grade`, `rgrade`] command produces an index table or permutation vector which, if applied to the input vector (as the index of C-u v r, say), would sort the vector. A permutation vector is just a vector of integers from 1 to n, where each integer occurs exactly once. One application of this is to sort a matrix of data rows using one column as the sort key; extract that column, grade it with V G, then use the result to reorder the original matrix with C-u v r. Another interesting property of the `V G` command is that, if the input is itself a permutation vector, the result will be the inverse of the permutation. The inverse of an index table is a rank table, whose kth element says where the kth original vector element will rest when the vector is sorted. To get a rank table, just use V G V G.

With the Inverse flag, I V G produces an index table that would sort the input into decreasing order. Note that V S and V G use a "stable" sorting algorithm, i.e., any two elements which are equal will not be moved out of their original order. Generally there is no way to tell with V S, since two elements which are equal look the same, but with V G this can be an important issue. In the matrix-of-rows example, suppose you have names and telephone numbers as two columns and you wish to sort by phone number primarily, and by name when the numbers are equal. You can sort the data matrix by names first, and then again by phone numbers. Because the sort is stable, any two rows with equal phone numbers will remain sorted by name even after the second sort.

The V H (`calc-histogram`) [`histogram`] command builds a histogram of a vector of numbers. Vector elements are assumed to be integers or real numbers in the range [0..n) for some "number of bins"

n, which is the numeric prefix argument given to the command. The result is a vector of n counts of h many times each value appeared in the original vector. Non-integers in the input are rounded down t integers. Any vector elements outside the specified range are ignored. (You can tell if elements have been ignored by noting that the counts in the result vector don't add up to the length of the input vector.)

With the Hyperbolic flag, H V H pulls two vectors from the stack. The second-to-top vector is the list of numbers as before. The top vector is an equal-sized list of "weights" to attach to the elements of the data vector. For example, if the first data element is 4.2 and the first weight is 10, then 10 will be added to bin 4 of the result vector. Without the hyperbolic flag, every element has a weight of one.

The v t (`calc-transpose`) [`trn`] command computes the transpose of the matrix at the top of the stack. If the argument is a plain vector, it is treated as a row vector and transposed into a one-column matrix.

The v v (`calc-reverse-vector`) [`vec`] command reverses a vector end-for-end. Given a matrix, it reverses the order of the rows. (To reverse the columns instead, just use v t v v v t. The same principle can be used to apply other vector commands to the columns of a matrix.)

The v m (`calc-mask-vector`) [`vmask`] command uses one vector as a mask to extract elements of another vector. The mask is in the second-to-top position; the target vector is on the top of the stack. These vectors must have the same length. The result is the same as the target vector, but with all elements which correspond to zeros in the mask vector deleted. Thus, for example, `vmask([1, 0, 1, 0, 1], [a, b, c, d, e])' produces `[a, c, e]'. See section Logical Operations.

The v e (`calc-expand-vector`) [`vexp`] command expands a vector according to another mask vector. The result is a vector the same length as the mask, but with nonzero elements replaced by successive elements from the target vector. The length of the target vector is normally the number of nonzero elements in the mask. If the target vector is longer, its last few elements are lost. If the target vector is shorter, the last few nonzero mask elements are left unreplaced in the result. Thus `vexp([2, 0, 3, 0, 7], [a, b])' produces `[a, 0, b, 0, 7]'.

With the Hyperbolic flag, H v e takes a filler value from the top of the stack; the mask and target vectors come from the third and second elements of the stack. This filler is used where the mask is zero: `vexp([2, 0, 3, 0, 7], [a, b], z)' produces `[a, z, c, z, 7]'. If the filler value is itself a vector, then successive values are taken from it, so that the effect is to interleave two vectors according to the mask: `vexp([2, 0, 3, 7, 0, 0], [a, b], [x, y])' produces `[a, x, b, 7, y, 0]'.

Another variation on the masking idea is to combine `[a, b, c, d, e]' with the mask `[1, 0, 1, 0, 1]' to produce `[a, 0, c, 0, e]'. You can accomplish this with V M a &, mapping the logical "and" operation across the two vectors. See section Logical Operations. Note that the `? :` operation also discussed there allows other types of masking using vectors.

Vector and Matrix Arithmetic

Basic arithmetic operations like addition and multiplication are defined for vectors and matrices as well as for numbers. Division of matrices, in the sense of multiplying by the inverse, is supported. (Division by a matrix actually uses LU-decomposition for greater accuracy and speed.) See section Basic

Arithmetic.

The following functions are applied element-wise if their arguments are vectors or matrices: `change-sign`, `conj`, `arg`, `re`, `im`, `polar`, `rect`, `clean`, `float`, `frac`. See section Index of Algebraic Functions.

The V J (`calc-conj-transpose`) [`ctrn`] command computes the conjugate transpose of its argument, i.e., `conj(trn(x))'.

The A (`calc-abs`) [`abs`] command computes the Frobenius norm of a vector or matrix argument. This is the square root of the sum of the squares of the absolute values of the elements of the vector or matrix. If the vector is interpreted as a point in two- or three-dimensional space, this is the distance from that point to the origin.

The v n (`calc-rnorm`) [`rnorm`] command computes the row norm, or infinity-norm, of a vector or matrix. For a plain vector, this is the maximum of the absolute values of the elements. For a matrix, this is the maximum of the row-absolute-value-sums, i.e., of the sums of the absolute values of the elements along the various rows.

The V N (`calc-cnorm`) [`cnorm`] command computes the column norm, or one-norm, of a vector or matrix. For a plain vector, this is the sum of the absolute values of the elements. For a matrix, this is the maximum of the column-absolute-value-sums. General k-norms for k other than one or infinity are not provided.

The V C (`calc-cross`) [`cross`] command computes the right-handed cross product of two vectors, each of which must have exactly three elements.

The & (`calc-inv`) [`inv`] command computes the inverse of a square matrix. If the matrix is singular, the inverse operation is left in symbolic form. Matrix inverses are recorded so that once an inverse (or determinant) of a particular matrix has been computed, the inverse and determinant of the matrix can be recomputed quickly in the future.

If the argument to & is a plain number x, this command simply computes 1/x. This is okay, because the `/' operator also does a matrix inversion when dividing one by a matrix.

The V D (`calc-mdet`) [`det`] command computes the determinant of a square matrix.

The V L (`calc-mlud`) [`lud`] command computes the LU decomposition of a matrix. The result is a list of three matrices which, when multiplied together left-to-right, form the original matrix. The first is a permutation matrix that arises from pivoting in the algorithm, the second is lower-triangular with ones on the diagonal, and the third is upper-triangular.

The V T (`calc-mtrace`) [`tr`] command computes the trace of a square matrix. This is defined as the sum of the diagonal elements of the matrix.

Set Operations using Vectors

Calc includes several commands which interpret vectors as sets of objects. A set is a collection of objects; any given object can appear only once in the set. Calc stores sets as vectors of objects in sorted order. Objects in a Calc set can be any of the usual things, such as numbers, variables, or formulas. Two set elements are considered equal if they are identical, except that numerically equal numbers like the integer 4 and the float 4.0 are considered equal even though they are not "identical." Variables are treated like plain symbols without attached values by the set operations; subtracting the set `[b]' from `[a, b]' always yields the set `[a]' even though if the variables `a' and `b' both equalled 17, you might expect the answer `[]'.

If a set contains interval forms, then it is assumed to be a set of real numbers. In this case, all set operations require the elements of the set to be only things that are allowed in intervals: Real numbers, plus and minus infinity, HMS forms, and date forms. If there are variables or other non-real objects present in a real set, all set operations on it will be left in unevaluated form.

If the input to a set operation is a plain number or interval form a, it is treated like the one-element vector `[a]'. The result is always a vector, except that if the set consists of a single interval, the interval itself is returned instead.

See section Logical Operations, for the `in` function which tests if a certain value is a member of a given set. To test if the set A is a subset of the set B, use `vdiff(A, B) = []'.

The V + (`calc-remove-duplicates`) [`rdup`] command converts an arbitrary vector into set notation. It works by sorting the vector as if by V S, then removing duplicates. (For example, [a, 5, 4, a, 4.0] is sorted to `[4, 4.0, 5, a, a]' and then reduced to `[4, 5, a]'). Overlapping intervals are merged as necessary. You rarely need to use V + explicitly, since all the other set-based commands apply V + to their inputs before using them.

The V V (`calc-set-union`) [`vunion`] command computes the union of two sets. An object is in the union of two sets if and only if it is in either (or both) of the input sets. (You could accomplish the same thing by concatenating the sets with |, then using V +.)

The V ^ (`calc-set-intersect`) [`vint`] command computes the intersection of two sets. An object is in the intersection if and only if it is in both of the input sets. Thus if the input sets are disjoint, i.e., if they share no common elements, the result will be the empty vector `[]'. Note that the characters V and ^ were chosen to be close to the conventional mathematical notation for set union@c{ ($A \cup B$)} and intersection@c{ ($A \cap B$)} .

The V - (`calc-set-difference`) [`vdiff`] command computes the difference between two sets. An object is in the difference A - B if and only if it is in A but not in B. Thus subtracting `[y,z]' from a set will remove the elements `y' and `z' if they are present. You can also think of this as a general set complement operator; if A is the set of all possible values, then A - B is the "complement" of B. Obviously this is only practical if the set of all possible values in your problem is small enough to list in a Calc vector (or simple enough to express in a few intervals).

The V X (`calc-set-xor`) [`vxor`] command computes the "exclusive-or," or "symmetric difference" of two sets. An object is in the symmetric difference of two sets if and only if it is in one, but *not* both, of

the sets. Objects that occur in both sets "cancel out."

The V ~ (`calc-set-complement`) [`vcompl`] command computes the complement of a set with respect to the real numbers. Thus `vcompl(x)' is equivalent to `vdiff([-inf .. inf], x)'. For example, `vcompl([2, (3 .. 4]])' evaluates to `[[-inf .. 2), (2 .. 3], (4 .. inf]]'.

The V F (`calc-set-floor`) [`vfloor`] command reinterprets a set as a set of integers. Any non-integer values, and intervals that do not enclose any integers, are removed. Open intervals are converted to equivalent closed intervals. Successive integers are converted into intervals of integers. For example, the complement of the set `[2, 6, 7, 8]' is messy, but if you wanted the complement with respect to the set of integers you could type V ~ V F to get `[[-inf .. 1], [3 .. 5], [9 .. inf]]'.

The V E (`calc-set-enumerate`) [`venum`] command converts a set of integers into an explicit vector. Intervals in the set are expanded out to lists of all integers encompassed by the intervals. This only works for finite sets (i.e., sets which do not involve `-inf' or `inf').

The V : (`calc-set-span`) [`vspan`] command converts any set of reals into an interval form that encompasses all its elements. The lower limit will be the smallest element in the set; the upper limit will be the largest element. For an empty set, `vspan([])' returns the empty interval `[0 .. 0)'.

The V # (`calc-set-cardinality`) [`vcard`] command counts the number of integers in a set. The result is the length of the vector that would be produced by V E, although the computation is much more efficient than actually producing that vector.

Another representation for sets that may be more appropriate in some cases is binary numbers. If you are dealing with sets of integers in the range 0 to 49, you can use a 50-bit binary number where a particular bit is 1 if the corresponding element is in the set. See section Binary Number Functions, for a list of commands that operate on binary numbers. Note that many of the above set operations have direct equivalents in binary arithmetic: b o (`calc-or`), b a (`calc-and`), b d (`calc-diff`), b x (`calc-xor`), and b n (`calc-not`), respectively. You can use whatever representation for sets is most convenient to you.

The b u (`calc-unpack-bits`) [`vunpack`] command converts an integer that represents a set in binary into a set in vector/interval notation. For example, `vunpack(67)' returns `[[0 .. 1], 6]'. If the input is negative, the set it represents is semi-infinite: `vunpack(-4) = [2 .. inf)'. Use V E afterwards to expand intervals to individual values if you wish. Note that this command uses the b (binary) prefix key.

The b p (`calc-pack-bits`) [`vpack`] command converts the other way, from a vector or interval representing a set of nonnegative integers into a binary integer describing the same set. The set may include positive infinity, but must not include any negative numbers. The input is interpreted as a set of integers in the sense of V F (`vfloor`). Beware that a simple input like `[100]' can result in a huge integer representation (@c{2^{100}} 2^100, a 31-digit integer, in this case).

Statistical Operations on Vectors

The commands in this section take vectors as arguments and compute various statistical measures on the data stored in the vectors. The references used in the definitions of these functions are Bevington's *Data Reduction and Error Analysis for the Physical Sciences*, and *Numerical Recipes* by Press, Flannery,

Teukolsky and Vetterling.

The statistical commands use the u prefix key followed by a shifted letter or other character.

See section Manipulating Vectors, for a description of V H (`calc-histogram`).

See section Curve Fitting, for the a F command for doing least-squares fits to statistical data.

See section Probability Distribution Functions, for several common probability distribution functions.

Single-Variable Statistics

These functions do various statistical computations on single vectors. Given a numeric prefix argument, they actually pop n objects from the stack and combine them into a data vector. Each object may be either a number or a vector; if a vector, any sub-vectors inside it are "flattened" as if by v a 0; see section Manipulating Vectors. By default one object is popped, which (in order to be useful) is usually a vector.

If an argument is a variable name, and the value stored in that variable is a vector, then the stored vector is used. This method has the advantage that if your data vector is large, you can avoid the slow process of manipulating it directly on the stack.

These functions are left in symbolic form if any of their arguments are not numbers or vectors, e.g., if an argument is a formula, or a non-vector variable. However, formulas embedded within vector arguments are accepted; the result is a symbolic representation of the computation, based on the assumption that the formula does not itself represent a vector. All varieties of numbers such as error forms and interval forms are acceptable.

Some of the functions in this section also accept a single error form or interval as an argument. They then describe a property of the normal or uniform (respectively) statistical distribution described by the argument. The arguments are interpreted in the same way as the M argument of the random number function k r. In particular, an interval with integer limits is considered an integer distribution, so that `[2 .. 6)' is the same as `[2 .. 5]'. An interval with at least one floating-point limit is a continuous distribution: `[2.0 .. 6.0)' is *not* the same as `[2.0 .. 5.0]'!

The u # (`calc-vector-count`) [`vcount`] command computes the number of data values represented by the inputs. For example, `vcount(1, [2, 3], [[4, 5], [], x, y])' returns 7. If the argument is a single vector with no sub-vectors, this simply computes the length of the vector.

The u + (`calc-vector-sum`) [`vsum`] command computes the sum of the data values. The u * (`calc-vector-prod`) [`vprod`] command computes the product of the data values. If the input is a single flat vector, these are the same as V R + and V R * (see section Reducing and Mapping Vectors).

The u X (`calc-vector-max`) [`vmax`] command computes the maximum of the data values, and the u N (`calc-vector-min`) [`vmin`] command computes the minimum. If the argument is an interval, this finds the minimum or maximum value in the interval. (Note that `vmax([2..6)) = 5' as described above.) If the argument is an error form, this returns plus or minus infinity.

The u M (`calc-vector-mean`) [`vmean`] command computes the average (arithmetic mean) of the data values. If the inputs are error forms @c{x +/- σ} `x +/- s', this is the weighted mean of

the x values with weights @c{$1 / \sigma^2$} 1 / s^2. If the inputs are not error forms, this is simply the sum of the values divided by the count of the values.

Note that a plain number can be considered an error form with error @c{$\sigma = 0$} s = 0. If the input to u M is a mixture of plain numbers and error forms, the result is the mean of the plain numbers, ignoring all values with non-zero errors. (By the above definitions it's clear that a plain number effectively has an infinite weight, next to which an error form with a finite weight is completely negligible.)

This function also works for distributions (error forms or intervals). The mean of an error form `*a* `+/-` *b*' is simply a. The mean of an interval is the mean of the minimum and maximum values of the interval.

The I u M (`calc-vector-mean-error`) [`vmeane`] command computes the mean of the data points expressed as an error form. This includes the estimated error associated with the mean. If the inputs are error forms, the error is the square root of the reciprocal of the sum of the reciprocals of the squares of the input errors. (I.e., the variance is the reciprocal of the sum of the reciprocals of the variances.) If the inputs are plain numbers, the error is equal to the standard deviation of the values divided by the square root of the number of values. (This works out to be equivalent to calculating the standard deviation and then assuming each value's error is equal to this standard deviation.)

The H u M (`calc-vector-median`) [`vmedian`] command computes the median of the data values. The values are first sorted into numerical order; the median is the middle value after sorting. (If the number of data values is even, the median is taken to be the average of the two middle values.) The median function is different from the other functions in this section in that the arguments must all be real numbers; variables are not accepted even when nested inside vectors. (Otherwise it is not possible to sort the data values.) If any of the input values are error forms, their error parts are ignored.

The median function also accepts distributions. For both normal (error form) and uniform (interval) distributions, the median is the same as the mean.

The H I u M (`calc-vector-harmonic-mean`) [`vhmean`] command computes the harmonic mean of the data values. This is defined as the reciprocal of the arithmetic mean of the reciprocals of the values.

The u G (`calc-vector-geometric-mean`) [`vgmean`] command computes the geometric mean of the data values. This is the *N*th root of the product of the values. This is also equal to the `exp` of the arithmetic mean of the logarithms of the data values.

The H u G [`agmean`] command computes the "arithmetic-geometric mean" of two numbers taken from the stack. This is computed by replacing the two numbers with their arithmetic mean and geometric mean, then repeating until the two values converge.

Another commonly used mean, the RMS (root-mean-square), can be computed for a vector of numbers simply by using the A command.

The u S (`calc-vector-sdev`) [`vsdev`] command computes the standard deviation@c{ σ} of the data values. If the values are error forms, the errors are used as weights just as for u M. This is the *sample* standard deviation, whose value is the square root of the sum of the squares of the differences between the values and the mean of the N values, divided by N-1.

This function also applies to distributions. The standard deviation of a single error form is simply the error part. The standard deviation of a continuous interval happens to equal the difference between the limits, divided by @c{$\sqrt{12}$} sqrt(12). The standard deviation of an integer interval is the same as the standard deviation of a vector of those integers.

The I u S (`calc-vector-pop-sdev`) [`vpsdev`] command computes the *population* standard deviation. It is defined by the same formula as above but dividing by N instead of by N-1. The population standard deviation is used when the input represents the entire set of data values in the distribution; the sample standard deviation is used when the input represents a sample of the set of all data values, so that the mean computed from the input is itself only an estimate of the true mean.

For error forms and continuous intervals, `vpsdev` works exactly like `vsdev`. For integer intervals, it computes the population standard deviation of the equivalent vector of integers.

The H u S (`calc-vector-variance`) [`vvar`] and H I u S (`calc-vector-pop-variance`) [`vpvar`] commands compute the variance of the data values. The variance is the square@c{ σ^2} of the standard deviation, i.e., the sum of the squares of the deviations of the data values from the mean. (This definition also applies when the argument is a distribution.)

The `vflat` algebraic function returns a vector of its arguments, interpreted in the same way as the other functions in this section. For example, `vflat(1, [2, [3, 4]], 5)' returns `[1, 2, 3, 4, 5]'.

Paired-Sample Statistics

The functions in this section take two arguments, which must be vectors of equal size. The vectors are each flattened in the same way as by the single-variable statistical functions. Given a numeric prefix argument of 1, these functions instead take one object from the stack, which must be an @c{$N\times2$} Nx2 matrix of data values. Once again, variable names can be used in place of actual vectors and matrices.

The u C (`calc-vector-covariance`) [`vcov`] command computes the sample covariance of two vectors. The covariance of vectors x and y is the sum of the products of the differences between the elements of x and the mean of x times the differences between the corresponding elements of y and the mean of y, all divided by N-1. Note that the variance of a vector is just the covariance of the vector with itself. Once again, if the inputs are error forms the errors are used as weight factors. If both x and y are composed of error forms, the error for a given data point is taken as the square root of the sum of the squares of the two input errors.

The I u C (`calc-vector-pop-covariance`) [`vpcov`] command computes the population covariance, which is the same as the sample covariance computed by u C except dividing by N instead of N-1.

The H u C (`calc-vector-correlation`) [`vcorr`] command computes the linear correlation coefficient of two vectors. This is defined by the covariance of the vectors divided by the product of their standard deviations. (There is no difference between sample or population statistics here.)

Reducing and Mapping Vectors

The commands in this section allow for more general operations on the elements of vectors.

The simplest of these operations is V A (`calc-apply`) [`apply`], which applies a given operator to the elements of a vector. For example, applying the hypothetical function `f` to the vector `[1, 2, 3]' would produce the function call `f(1, 2, 3)'. Applying the + function to the vector `[a, b]' gives `a + b'. Applying + to the vector `[a, b, c]' is an error, since the + function expects exactly two arguments.

While V A is useful in some cases, you will usually find that either V R or V M, described below, is closer to what you want.

Specifying Operators

Commands in this section (like V A) prompt you to press the key corresponding to the desired operator. Press ? for a partial list of the available operators. Generally, an operator is any key or sequence of keys that would normally take one or more arguments from the stack and replace them with a result. For example, V A H C uses the hyperbolic cosine operator, `cosh`. (Since `cosh` expects one argument, V A H C requires a vector with a single element as its argument.)

You can press x at the operator prompt to select any algebraic function by name to use as the operator. This includes functions you have defined yourself using the Z F command. (See section Programming with Formulas.) If you give a name for which no function has been defined, the result is left in symbolic form, as in `f(1, 2, 3)'. Calc will prompt for the number of arguments the function takes if it can't figure it out on its own (say, because you named a function that is currently undefined). It is also possible to type a digit key before the function name to specify the number of arguments, e.g., V M 3 x f RET calls `f` with three arguments even if it looks like it ought to have only two. This technique may be necessary if the function allows a variable number of arguments. For example, the v e [`vexp`] function accepts two or three arguments; if you want to map with the three-argument version, you will have to type V M 3 v e.

It is also possible to apply any formula to a vector by treating that formula as a function. When prompted for the operator to use, press ' (the apostrophe) and type your formula as an algebraic entry. You will then be prompted for the argument list, which defaults to a list of all variables that appear in the formula, sorted into alphabetic order. For example, suppose you enter the formula `x + 2y^x'. The default argument list would be `(x y)', which means that if this function is applied to the arguments `[3, 10]' the result will be `3 + 2*10^3'. (If you plan to use a certain formula in this way often, you might consider defining it as a function with Z F.)

Another way to specify the arguments to the formula you enter is with $, $$, and so on. For example, V A ' $$ + 2$^$$ has the same effect as the previous example. The argument list is automatically taken to be `($$ $)'. (The order of the arguments may seem backwards, but it is analogous to the way normal algebraic entry interacts with the stack.)

If you press $ at the operator prompt, the effect is similar to the apostrophe except that the relevant formula is taken from top-of-stack instead. The actual vector arguments of the V A $ or related command then start at the second-to-top stack position. You will still be prompted for an argument list.

A function can be written without a name using the notation `<#1 - #2>', which means "a function of arguments that computes the first argument minus the second argument." The symbols `#1' and `#2' are placeholders for the arguments. You can use any names for these placeholders if you wish, by including an argument list followed by a colon: `<x, y : x - y>'. When you type V A ' $$ + 2$^$$ RET, Calc builds the nameless function `<#1 + 2 #2^#1>' as the function to map across the vectors. When you type V A ' x + 2y^x RET RET, Calc builds the nameless function `<x, y : x + 2 y^x>'. In both cases, Calc also writes the nameless function to the Trail so that you can get it back later if you wish.

If there is only one argument, you can write `#' in place of `#1'. (Note that `< >' notation is also used for date forms. Calc tells that `<stuff>' is a nameless function by the presence of `#' signs inside stuff, or by the fact that stuff begins with a list of variables followed by a colon.)

You can type a nameless function directly to V A ', or put one on the stack and use it with V A $. Calc will not prompt for an argument list in this case, since the nameless function specifies the argument list as well as the function itself. In V A ', you can omit the `< >' marks if you use `#' notation for the arguments, so that V A ' #1+#2 RET is the same as V A ' <#1+#2> RET, which in turn is the same as V A ' $$+$ RET.

The internal format for `<x, y : x + y>' is `lambda(x, y, x + y)'. (The word `lambda` derives from Lisp notation and the theory of functions.) The internal format for `<#1 + #2>' is `lambda(ArgA, ArgB, ArgA + ArgB)'. Note that there is no actual Calc function called `lambda`; the whole point is that the `lambda` expression is used in its symbolic form, not evaluated for an answer until it is applied to specific arguments by a command like V A or V M.

(Actually, `lambda` does have one special property: Its arguments are never evaluated; for example, putting `<(2/3) #>' on the stack will not simplify the `2/3' until the nameless function is actually called.)

As usual, commands like V A have algebraic function name equivalents. For example, V A k g with an argument of `v' is equivalent to `apply(gcd, v)'. The first argument specifies the operator name, and is either a variable whose name is the same as the function name, or a nameless function like `<#^3+1>'. Operators that are normally written as algebraic symbols have the names `add`, `sub`, `mul`, `div`, `pow`, `neg`, `mod`, and `vconcat`.

The `call` function builds a function call out of several arguments: `call(gcd, x, y)' is the same as `apply(gcd, [x, y])', which in turn is the same as `gcd(x, y)'. The first argument of `call`, like the other functions described here, may be either a variable naming a function, or a nameless function (`call(<#1+2#2>, x, y)' is the same as `x + 2y').

(Experts will notice that it's not quite proper to use a variable to name a function, since the name `gcd` corresponds to the Lisp variable `var-gcd` but to the Lisp function `calcFunc-gcd`. Calc automatically makes this translation, so you don't have to worry about it.)

Mapping

The V M (`calc-map`) [`map`] command applies a given operator elementwise to one or more vectors. For example, mapping `A` [`abs`] produces a vector of the absolute values of the elements in the input vector. Mapping `+` pops two vectors from the stack, which must be of equal length, and produces a vector of the pairwise sums of the elements. If either argument is a non-vector, it is duplicated for each

element of the other vector. For example, [1,2,3] 2 V M ^ squares the elements of the specified vector. With the 2 listed first, it would have computed a vector of powers of two. Mapping a user-defined function pops as many arguments from the stack as the function requires. If you give an undefined name, you will be prompted for the number of arguments to use.

If any argument to V M is a matrix, the operator is normally mapped across all elements of the matrix. For example, given the matrix [[1, -2, 3], [-4, 5, -6]], V M A takes six absolute values to produce another @c{3×2} 3x2 matrix, [[1, 2, 3], [4, 5, 6]].

The command V M _ [`mapr`] (i.e., type an underscore at the operator prompt) maps by rows instead. For example, V M _ A views the above matrix as a vector of two 3-element row vectors. It produces a new vector which contains the absolute values of those row vectors, namely [3.74, 8.77]. (Recall, the absolute value of a vector is defined as the square root of the sum of the squares of the elements.) Some operators accept vectors and return new vectors; for example, v v reverses a vector, so V M _ v v would reverse each row of the matrix to get a new matrix, [[3, -2, 1], [-6, 5, -4]].

Sometimes a vector of vectors (representing, say, strings, sets, or lists) happens to look like a matrix. If so, remember to use V M _ if you want to map a function across the whole strings or sets rather than across their individual elements.

The command V M : [`mapc`] maps by columns. Basically, it transposes the input matrix, maps by rows, and then, if the result is a matrix, transposes again. For example, V M : A takes the absolute values of the three columns of the matrix, treating each as a 2-vector, and V M : v v reverses the columns to get the matrix [[-4, 5, -6], [1, -2, 3]].

(The symbols _ and : were chosen because they had row-like and column-like appearances, and were not already taken by useful operators. Also, they appear shifted on most keyboards so they are easy to type after V M.)

The _ and : modifiers have no effect on arguments that are not matrices (so if none of the arguments are matrices, they have no effect at all). If some of the arguments are matrices and others are plain numbers, the plain numbers are held constant for all rows of the matrix (so that 2 V M _ ^ squares every row of a matrix; squaring a vector takes a dot product of the vector with itself).

If some of the arguments are vectors with the same lengths as the rows (for V M _) or columns (for V M :) of the matrix arguments, those vectors are also held constant for every row or column.

Sometimes it is useful to specify another mapping command as the operator to use with V M. For example, V M _ V A + applies V A + to each row of the input matrix, which in turn adds the two values on that row. If you give another vector-operator command as the operator for V M, it automatically uses map-by-rows mode if you don't specify otherwise; thus V M V A + is equivalent to V M _ V A +. (If you really want to map-by-elements another mapping command, you can use a triple-nested mapping command: V M V M V A + means to map V M V A + over the rows of the matrix; in turn, V A + is mapped over the elements of each row.)

Previous versions of Calc had "map across" and "map down" modes that are now considered obsolete; the old "map across" is now simply V M V A, and "map down" is now V M : V A. The algebraic functions `mapa` and `mapd` are still supported, though. Note also that, while the old mapping modes were persistent (once you set the mode, it would apply to later mapping commands until you reset it), the new

: and _ modifiers apply only to the current mapping command. The default V M always means map-by-elements.

See section Algebraic Manipulation, for the a M command, which is like V M but for equations and inequalities instead of vectors. See section Storing Variables, for the s m command which modifies a variable's stored value using a V M-like operator.

Reducing

The V R (`calc-reduce`) [`reduce`] command applies a given binary operator across all the elements of a vector. A binary operator is a function such as `+` or `max` which takes two arguments. For example, reducing `+` over a vector computes the sum of the elements of the vector. Reducing `-` computes the first element minus each of the remaining elements. Reducing `max` computes the maximum element and so on. In general, reducing `f` over the vector `[a, b, c, d]' produces `f(f(f(a, b), c), d)'.

The I V R [`rreduce`] command is similar to V R except that works from right to left through the vector. For example, plain V R - on the vector `[a, b, c, d]' produces `a - b - c - d' but I V R - on the same vector produces `a - (b - (c - d))', or `a - b + c - d'. This "alternating sum" occurs frequently in power series expansions.

The V U (`calc-accumulate`) [`accum`] command does an accumulation operation. Here Calc does the corresponding reduction operation, but instead of producing only the final result, it produces a vector of all the intermediate results. Accumulating `+` over the vector `[a, b, c, d]' produces the vector `[a, a + b, a + b + c, a + b + c + d]'.

The I V U [`raccum`] command does a right-to-left accumulation. For example, I V U - on the vector `[a, b, c, d]' produces the vector `[a - b + c - d, b - c + d, c - d, d]'.

As for V M, V R normally reduces a matrix elementwise. For example, given the matrix [[a, b, c], [d, e, f]], V R + will compute a + b + c + d + e + f. You can type V R _ or V R : to modify this behavior. The V R _ [`reducea`] command reduces "across" the matrix; it reduces each row of the matrix as a vector, then collects the results. Thus V R _ + of this matrix would produce [a + b + c, d + e + f]. Similarly, V R : [`reduced`] reduces down; V R : + would produce [a + d, b + e, c + f].

There is a third "by rows" mode for reduction that is occasionally useful; V R = [`reducer`] simply reduces the operator over the rows of the matrix themselves. Thus V R = + on the above matrix would get the same result as V R : +, since adding two row vectors is equivalent to adding their elements. But V R = * would multiply the two rows (to get a single number, their dot product), while V R : * would produce a vector of the products of the columns.

These three matrix reduction modes work with V R and I V R, but they are not currently supported with V U or I V U.

The obsolete reduce-by-columns function, `reducec`, is still supported but there is no way to get it through the V R command.

The commands M-# : and M-# _ are equivalent to typing M-# r to grab a rectangle of data into Calc, and then typing V R : + or V R _ +, respectively, to sum the columns or rows of the matrix. See section

Grabbing from Other Buffers.

Nesting and Fixed Points

The H V R [`nest`] command applies a function to a given argument repeatedly. It takes two values, `a' and `n', from the stack, where `n' must be an integer. It then applies the function nested `n' times; if the function is `f' and `n' is 3, the result is `f(f(f(a)))'. The number `n' may be negative if Calc knows an inverse for the function `f'; for example, `nest(sin, a, -2)' returns `arcsin(arcsin(a))'.

The H V U [`anest`] command is an accumulating version of `nest`: It returns a vector of `n+1' values, e.g., `[a, f(a), f(f(a)), f(f(f(a)))]'. If `n' is negative and `F' is the inverse of `f', then the result is of the form `[a, F(a), F(F(a)), F(F(F(a)))]'.

The H I V R [`fixp`] command is like H V R, except that it takes only an `a' value from the stack; the function is applied until it reaches a "fixed point," i.e., until the result no longer changes.

The H I V U [`afixp`] command is an accumulating `fixp`. The first element of the return vector will be the initial value `a'; the last element will be the final result that would have been returned by `fixp`.

For example, 0.739085 is a fixed point of the cosine function (in radians): `cos(0.739085) = 0.739085'. You can find this value by putting, say, 1.0 on the stack and typing H I V U C. (We use the accumulating version so we can see the intermediate results: `[1, 0.540302, 0.857553, 0.65329, ...]'. With a precision of six, this command will take 36 steps to converge to 0.739085.)

Newton's method for finding roots is a classic example of iteration to a fixed point. To find the square root of five starting with an initial guess, Newton's method would look for a fixed point of the function `(x + 5/x) / 2'. Putting a guess of 1 on the stack and typing H I V R ' ($ + 5/$)/2 RET quickly yields the result 2.23607. This is equivalent to using the a R (`calc-find-root`) command to find a root of the equation `x^2 = 5'.

These examples used numbers for `a' values. Calc keeps applying the function until two successive results are equal to within the current precision. For complex numbers, both the real parts and the imaginary parts must be equal to within the current precision. If `a' is a formula (say, a variable name), then the function is applied until two successive results are exactly the same formula. It is up to you to ensure that the function will eventually converge; if it doesn't, you may have to press C-g to stop the Calculator.

The algebraic `fixp` function takes two optional arguments, `n' and `tol'. The first is the maximum number of steps to be allowed, and must be either an integer or the symbol `inf' (infinity, the default). The second is a convergence tolerance. If a tolerance is specified, all results during the calculation must be numbers, not formulas, and the iteration stops when the magnitude of the difference between two successive results is less than or equal to the tolerance. (This implies that a tolerance of zero iterates until the results are exactly equal.)

Putting it all together, `fixp(<(# + A/#)/2>, B, 20, 1e-10)' computes the square root of `A' given the initial guess `B', stopping when the result is correct within the specified tolerance, or when 20 steps have been taken, whichever is sooner.

Generalized Products

The V O (`calc-outer-product`) [`outer`] command applies a given binary operator to all possible pairs of elements from two vectors, to produce a matrix. For example, V O * with `[a, b]' and `[x, y, z]' on the stack produces a multiplication table: `[[a x, a y, a z], [b x, b y, b z]]'. Element r,c of the result matrix is obtained by applying the operator to element r of the lefthand vector and element c of the righthand vector.

The V I (`calc-inner-product`) [`inner`] command computes the generalized inner product of two vectors or matrices, given a "multiplicative" operator and an "additive" operator. These can each actually be any binary operators; if they are `*' and `+', respectively, the result is a standard matrix multiplication. Element r,c of the result matrix is obtained by mapping the multiplicative operator across row r of the lefthand matrix and column c of the righthand matrix, and then reducing with the additive operator. Just as for the standard * command, this can also do a vector-matrix or matrix-vector inner product, or a vector-vector generalized dot product.

Since V I requires two operators, it prompts twice. In each case, you can use any of the usual methods for entering the operator. If you use $ twice to take both operator formulas from the stack, the first (multiplicative) operator is taken from the top of the stack and the second (additive) operator is taken from second-to-top.

Vector and Matrix Display Formats

Commands for controlling vector and matrix display use the v prefix instead of the usual d prefix. But they are display modes; in particular, they are influenced by the I and H prefix keys in the same way (see section Display Modes). Matrix display is also influenced by the d O (`calc-flat-language`) mode; see section Normal Language Modes.

The commands v < (`calc-matrix-left-justify`), v > (`calc-matrix-right-justify`), and v = (`calc-matrix-center-justify`) control whether matrix elements are justified to the left, right, or center of their columns.

The v [(`calc-vector-brackets`) command turns the square brackets that surround vectors and matrices displayed in the stack on and off. The v { (`calc-vector-braces`) and v ((`calc-vector-parens`) commands use curly braces or parentheses, respectively, instead of square brackets. For example, v { might be used in preparation for yanking a matrix into a buffer running Mathematica. (In fact, the Mathematica language mode uses this mode; see section Mathematica Language Mode.) Note that, regardless of the display mode, either brackets or braces may be used to enter vectors, and parentheses may never be used for this purpose.

The v] (`calc-matrix-brackets`) command controls the "big" style display of matrices. It prompts for a string of code letters; currently implemented letters are `R`, which enables brackets on each row of the matrix; `O`, which enables outer brackets in opposite corners of the matrix; and `C`, which enables commas or semicolons at the ends of all rows but the last. The default format is `RO'. (Before Calc 2.00, the format was fixed at `ROC'.) Here are some example matrices:


```
[ [ 123,  0,   0  ]       [ [ 123,  0,   0  ],
  [  0,  123,  0  ]         [  0,  123,  0  ],
  [  0,   0,  123 ] ]       [  0,   0,  123 ] ]

        RO                        ROC


  [ 123,  0,   0            [ 123,  0,   0 ;
    0,  123,  0               0,  123,  0 ;
    0,   0,  123 ]            0,   0,  123 ]

        O                         OC


  [ 123,  0,   0  ]           123,  0,   0
  [  0,  123,  0  ]            0,  123,  0
  [  0,   0,  123 ]            0,   0,  123

        R                        blank
```

Note that of the formats shown here, `RO', `ROC', and `OC' are all recognized as matrices during reading, while the others are useful for display only.

The v , (`calc-vector-commas`) command turns commas on and off in vector and matrix display.

In vectors of length one, and in all vectors when commas have been turned off, Calc adds extra parentheses around formulas that might otherwise be ambiguous. For example, `[a b]' could be a vector of the one formula `a b', or it could be a vector of two variables with commas turned off. Calc will display the former case as `[(a b)]'. You can disable these extra parentheses (to make the output less cluttered at the expense of allowing some ambiguity) by adding the letter `P` to the control string you give to v] (as described above).

The v . (`calc-full-vectors`) command turns abbreviated display of long vectors on and off. In this mode, vectors of six or more elements, or matrices of six or more rows or columns, will be displayed in an abbreviated form that displays only the first three elements and the last element: `[a, b, c, ..., z]'. When very large vectors are involved this will substantially improve Calc's display speed.

The t . (`calc-full-trail-vectors`) command controls a similar mode for recording vectors in the Trail. If you turn on this mode, vectors of six or more elements and matrices of six or more rows or columns will be abbreviated when they are put in the Trail. The t y (`calc-trail-yank`) command will be unable to recover those vectors. If you are working with very large vectors, this mode will improve the speed of all operations that involve the trail.

The v / (`calc-break-vectors`) command turns multi-line vector display on and off. Normally, matrices are displayed with one row per line but all other types of vectors are displayed in a single line. This mode causes all vectors, whether matrices or not, to be displayed with a single element per line. Sub-vectors within the vectors will still use the normal linear form.

Go to the previous, next section.

Go to the previous, next section.

Algebra

This section covers the Calc features that help you work with algebraic formulas. First, the general sub-formula selection mechanism is described; this works in conjunction with any Calc commands. Then, commands for specific algebraic operations are described. Finally, the flexible rewrite rule mechanism is discussed.

The algebraic commands use the a key prefix; selection commands use the j (for "just a letter that wasn't used for anything else") prefix.

See section Editing Stack Entries, to see how to manipulate formulas using regular Emacs editing commands.

When doing algebraic work, you may find several of the Calculator's modes to be helpful, including algebraic-simplification mode (m A) or no-simplification mode (m O), algebraic-entry mode (m a), fraction mode (m f), and symbolic mode (m s). See section Mode Settings, for discussions of these modes. You may also wish to select "big" display mode (d B). See section Normal Language Modes.

Selecting Sub-Formulas

When working with an algebraic formula it is often necessary to manipulate a portion of the formula rather than the formula as a whole. Calc allows you to "select" a portion of any formula on the stack. Commands which would normally operate on that stack entry will now operate only on the sub-formula, leaving the surrounding part of the stack entry alone.

One common non-algebraic use for selection involves vectors. To work on one element of a vector in-place, simply select that element as a "sub-formula" of the vector.

Making Selections

To select a sub-formula, move the Emacs cursor to any character in that sub-formula, and press j s (`calc-select-here`). Calc will highlight the smallest portion of the formula that contains that character. By default the sub-formula is highlighted by blanking out all of the rest of the formula with dots. Selection works in any display mode but is perhaps easiest in "big" (d B) mode. Suppose you enter the following formula:

```
                3    ___
    (a + b)    + V c
1:  ---------------
        2 x + 1
```

(by typing ' ((a+b)^3 + sqrt(c)) / (2x+1)). If you move the cursor to the letter `b' and press j s, the display changes to


```
          .    ...
    .. . b.  . . .
1*  ..............
        . . . .
```

Every character not part of the sub-formula `b' has been changed to a dot. The `*' next to the line number is to remind you that the formula has a portion of it selected. (In this case, it's very obvious, but it might not always be. If Embedded Mode is enabled, the word `Sel' also appears in the mode line because the stack may not be visible. see section Embedded Mode.)

If you had instead placed the cursor on the parenthesis immediately to the right of the `b', the selection would have been:

```
          .    ...
    (a + b)  . . .
1*  ..............
        . . . .
```

The portion selected is always large enough to be considered a complete formula all by itself, so selecting the parenthesis selects the whole formula that it encloses. Putting the cursor on the the `+' sign would have had the same effect.

(Strictly speaking, the Emacs cursor is really the manifestation of the Emacs "point," which is a position *between* two characters in the buffer. So purists would say that Calc selects the smallest sub-formula which contains the character to the right of "point.")

If you supply a numeric prefix argument n, the selection is expanded to the nth enclosing sub-formula. Thus, positioning the cursor on the `b' and typing C-u 1 j s will select `a + b'; typing C-u 2 j s will select `(a + b)^3', and so on.

If the cursor is not on any part of the formula, or if you give a numeric prefix that is too large, the entire formula is selected.

If the cursor is on the `.' line that marks the top of the stack (i.e., its normal "rest position"), this command selects the entire formula at stack level 1. Most selection commands similarly operate on the formula at the top of the stack if you haven't positioned the cursor on any stack entry.

The j a (`calc-select-additional`) command enlarges the current selection to encompass the cursor. To select the smallest sub-formula defined by two different points, move to the first and press j s, then move to the other and press j a. This is roughly analogous to using C-@ (`set-mark-command`) to select the two ends of a region of text during normal Emacs editing.

The j o (`calc-select-once`) command selects a formula in exactly the same way as j s, except that the selection will last only as long as the next command that uses it. For example, j o 1 + is a handy way to add one to the sub-formula indicated by the cursor.

(A somewhat more precise definition: The j o command sets a flag such that the next command involving selected stack entries will clear the selections on those stack entries afterwards. All other selection commands except j a and j O clear this flag.)

The j S (`calc-select-here-maybe`) and j O (`calc-select-once-maybe`) commands are equivalent to j s and j o, respectively, except that if the formula already has a selection they have no effect. This is analogous to the behavior of some commands such as j r (`calc-rewrite-selection`; see section Selections with Rewrite Rules) and is mainly intended to be used in keyboard macros that implement your own selection-oriented commands.

Selection of sub-formulas normally treats associative terms like `a + b - c + d' and `x * y * z' as single levels of the formula. If you place the cursor anywhere inside `a + b - c + d' except on one of the variable names and use j s, you will select the entire four-term sum.

The j b (`calc-break-selections`) command controls a mode in which the "deep structure" of these associative formulas shows through. Calc actually stores the above formulas as `((a + b) - c) + d' and `x * (y * z)'. (Note that for certain obscure reasons, Calc treats multiplication as right-associative.) Once you have enabled j b mode, selecting with the cursor on the `-' sign would only select the `a + b - c' portion, which makes sense when the deep structure of the sum is considered. There is no way to select the `b - c + d' portion; although this might initially look like just as legitimate a sub-formula as `a + b - c', the deep structure shows that it isn't. The d U command can be used to view the deep structure of any formula (see section Normal Language Modes).

When j b mode has not been enabled, the deep structure is generally hidden by the selection commands--what you see is what you get.

The j u (`calc-unselect`) command unselects the formula that the cursor is on. If there was no selection in the formula, this command has no effect. With a numeric prefix argument, it unselects the nth stack element rather than using the cursor position.

The j c (`calc-clear-selections`) command unselects all stack elements.

Changing Selections

Once you have selected a sub-formula, you can expand it using the j m (`calc-select-more`) command. If `a + b' is selected, pressing j m repeatedly works as follows:

```
           3    ...                3   ___                3   ___
    (a + b)   . . .         (a + b)  + V c         (a + b)  + V c
1*  ...............     1*  ...............     1*  ---------------
        . . . .                 . . . .                 2 x + 1
```

In the last example, the entire formula is selected. This is roughly the same as having no selection at all, but because there are subtle differences the `*' character is still there on the line number.

With a numeric prefix argument n, j m expands n times (or until the entire formula is selected). Note that j s with argument n is equivalent to plain j s followed by j m with argument n. If j m is used when there is no current selection, it is equivalent to j s.

Even though j m does not explicitly use the location of the cursor within the formula, it nevertheless uses the cursor to determine which stack element to operate on. As usual, j m when the cursor is not on any stack element operates on the top stack element.

The j l (`calc-select-less`) command reduces the current selection around the cursor position. That is, it selects the immediate sub-formula of the current selection which contains the cursor, the opposite of m. If the cursor is not inside the current selection, the command de-selects the formula.

The j 1 through j 9 (`calc-select-part`) commands select the nth sub-formula of the current selection. They are like j l (`calc-select-less`) except they use counting rather than the cursor position to decide which sub-formula to select. For example, if the current selection is a + b + c or f(a, b, c) or [a, b, c], then j 1 selects `a', j 2 selects `b', and j 3 selects `c'; in each of these cases, j 4 through j 9 would be errors.

If there is no current selection, j 1 through j 9 select the nth top-level sub-formula. (In other words, they act as if the entire stack entry were selected first.) To select the nth sub-formula where n is greater than nine, you must instead invoke j 1 with n as a numeric prefix argument.

The j n (`calc-select-next`) and j p (`calc-select-previous`) commands change the current selection to the next or previous sub-formula at the same level. For example, if `b' is selected in `2 + a*b*c + x', then j n selects `c'. Further j n commands would be in error because, even though there is something to the right of `c' (namely, `x'), it is not at the same level; in this case, it is not a term of the same product as `b' and `c'. However, j m (to select the whole product `a*b*c' as a term of the sum) followed by j n would successfully select the `x'.

Similarly, j p moves the selection from the `b' in this sample formula to the `a'. Both commands accept numeric prefix arguments to move several steps at a time.

It is interesting to compare Calc's selection commands with the Emacs Info system's commands for navigating through hierarchically organized documentation. Calc's j n command is completely analogous to Info's n command. Likewise, j p maps to p, j 2 maps to 2, and Info's u is like j m. (Note that j u stands for `calc-unselect`, not "up".) The Info m command is somewhat similar to Calc's j s and j l; in each case, you can jump directly to a sub-component of the hierarchy simply by pointing to it with the cursor.

Displaying Selections

The j d (`calc-show-selections`) command controls how selected sub-formulas are displayed. One of the alternatives is illustrated in the above examples; if we press j d we switch to the other style in which the selected portion itself is obscured by `#' signs:

```
          3    ...                    #    ___
    (a + b)  . . .              ## # ##  + V c
1*  ..............          1*  ---------------
       . . . .                      2 x + 1
```

Operating on Selections

Once a selection is made, all Calc commands that manipulate items on the stack will operate on the selected portions of the items instead. (Note that several stack elements may have selections at once, though there can be only one selection at a time in any given stack element.)

The j e (`calc-enable-selections`) command disables the effect that selections have on Calc commands. The current selections still exist, but Calc commands operate on whole stack elements anyway.

This mode can be identified by the fact that the `*' markers on the line numbers are gone, even though selections are visible. To reactivate the selections, press j e again.

To extract a sub-formula as a new formula, simply select the sub-formula and press RET. This normally duplicates the top stack element; here it duplicates only the selected portion of that element.

To replace a sub-formula with something different, you can enter the new value onto the stack and press TAB. This normally exchanges the top two stack elements; here it swaps the value you entered into the selected portion of the formula, returning the old selected portion to the top of the stack.

```
             3    ...                      ...                          ___
    (a + b)  . . .           17 x y . . .          17 x y + V c
2*  ..............      2*   ............      2:  -------------
       . . . .                  . . . .                2 x + 1

                                    3                        3
1:  17 x y                  1:  (a + b)              1:  (a + b)
```

In this example we select a sub-formula of our original example, enter a new formula, TAB it into place, then deselect to see the complete, edited formula.

If you want to swap whole formulas around even though they contain selections, just use j e before and after.

The j ' (`calc-enter-selection`) command is another way to replace a selected sub-formula. This command does an algebraic entry just like the regular ' key. When you press RET, the formula you type replaces the original selection. You can use the `$' symbol in the formula to refer to the original selection. If there is no selection in the formula under the cursor, the cursor is used to make a temporary selection for the purposes of the command. Thus, to change a term of a formula, all you have to do is move the Emacs cursor to that term and press j '.

The j ` (`calc-edit-selection`) command is a similar analogue of the ` (`calc-edit`) command. It edits the selected sub-formula in a separate buffer. If there is no selection, it edits the sub-formula indicated by the cursor.

To delete a sub-formula, press DEL. This generally replaces the sub-formula with the constant zero, but in a few suitable contexts it uses the constant one instead. The DEL key automatically deselects and re-simplifies the entire formula afterwards. Thus:

```
               ###
    17 x y + # #              17 x y              17 # y              17 y
1*  -------------      1:   -------      1*    -------      1:    -------
       2 x + 1               2 x + 1             2 x + 1             2 x + 1
```

In this example, we first delete the `sqrt(c)' term; Calc accomplishes this by replacing `sqrt(c)' with zero and resimplifying. We then delete the x in the numerator; since this is part of a product, Calc replaces it with `1' and resimplifies.

If you select an element of a vector and press DEL, that element is deleted from the vector. If you delete

one side of an equation or inequality, only the opposite side remains.

The j DEL (`calc-del-selection`) command is like DEL but with the auto-selecting behavior of j ' and j `. It deletes the selected portion of the formula indicated by the cursor, or, in the absence of a selection, it deletes the sub-formula indicated by the cursor position.

(There is also an auto-selecting j RET (`calc-copy-selection`) command.)

Normal arithmetic operations also apply to sub-formulas. Here we select the denominator, press 5 - to subtract five from the denominator, press n to negate the denominator, then press Q to take the square root.

```
     .. .          .. .          .. .          .. .
1*  .......   1*  .......   1*  .......   1*  .........
    2 x + 1       2 x - 4       4 - 2 x      _________
                                             V 4 - 2 x
```

Certain types of operations on selections are not allowed. For example, for an arithmetic function like - no more than one of the arguments may be a selected sub-formula. (As the above example shows, the result of the subtraction is spliced back into the argument which had the selection; if there were more than one selection involved, this would not be well-defined.) If you try to subtract two selections, the command will abort with an error message.

Operations on sub-formulas sometimes leave the formula as a whole in an "un-natural" state. Consider negating the `2 x' term of our sample formula by selecting it and pressing n (`calc-change-sign`).

```
      .. .            .. .
1*  .........   1*  ..........
     .......         .........
    . . . 2 x       . . . -2 x
```

Unselecting the sub-formula reveals that the minus sign, which would normally have cancelled out with the subtraction automatically, has not been able to do so because the subtraction was not part of the selected portion. Pressing = (`calc-evaluate`) or doing any other mathematical operation on the whole formula will cause it to be simplified.

```
       17 y            17 y
1:  ----------  1:  ----------
     _________       _________
    V 4 - -2 x      V 4 + 2 x
```

Rearranging Formulas using Selections

The j R (`calc-commute-right`) command moves the selected sub-formula to the right in its surrounding formula. Generally the selection is one term of a sum or product; the sum or product is rearranged according to the commutative laws of algebra.

As with j ' and j DEL, the term under the cursor is used if there is no selection in the current formula. All commands described in this section share this property. In this example, we place the cursor on the `a' and

type j R, then repeat.

```
1:  a + b - c          1:  b + a - c          1:  b - c + a
```

Note that in the final step above, the `a' is switched with the `c' but the signs are adjusted accordingly. When moving terms of sums and products, j R will never change the mathematical meaning of the formula.

The selected term may also be an element of a vector or an argument of a function. The term is exchanged with the one to its right. In this case, the "meaning" of the vector or function may of course be drastically changed.

```
1:  [a, b, c]          1:  [b, a, c]          1:  [b, c, a]

1:  f(a, b, c)         1:  f(b, a, c)         1:  f(b, c, a)
```

The j L (`calc-commute-left`) command is like j R except that it swaps the selected term with the one to its left.

With numeric prefix arguments, these commands move the selected term several steps at a time. It is an error to try to move a term left or right past the end of its enclosing formula. With numeric prefix arguments of zero, these commands move the selected term as far as possible in the given direction.

The j D (`calc-sel-distribute`) command mixes the selected sum or product into the surrounding formula using the distributive law. For example, in `a * (b - c)' with the `b - c' selected, the result is `a b - a c'. This also distributes products or quotients into surrounding powers, and can also do transformations like `exp(a + b)' to `exp(a) exp(b)', where `a + b' is the selected term, and `ln(a ^ b)' to `ln(a) b', where `a ^ b' is the selected term.

For multiple-term sums or products, j D takes off one term at a time: `a * (b + c - d)' goes to `a * (c - d) + a b' with the `c - d' selected so that you can type j D repeatedly to expand completely. The j D command allows a numeric prefix argument which specifies the maximum number of times to expand at once; the default is one time only.

The j D command is implemented using rewrite rules. See section Selections with Rewrite Rules. The rules are stored in the Calc variable `DistribRules`. A convenient way to view these rules is to use s e (`calc-edit-variable`) which displays and edits the stored value of a variable. Press M-# M-# to return from editing mode; be careful not to make any actual changes or else you will affect the behavior of future j D commands!

To extend j D to handle new cases, just edit `DistribRules` as described above. You can then use the s p command to save this variable's value permanently for future Calc sessions. See section Other Operations on Variables.

The j M (`calc-sel-merge`) command is the complement of j D; given `a b - a c' with either `a b' or `a c' selected, the result is `a * (b - c)'. Once again, j M can also merge calls to functions like `exp` and `ln`; examine the variable `MergeRules` to see all the relevant rules.

The j C (`calc-sel-commute`) command swaps the arguments of the selected sum, product, or equation. It always behaves as if j b mode were in effect, i.e., the sum `a + b + c' is treated as the nested sums `(a + b)

+ c' by this command. If you put the cursor on the first `+', the result is `(b + a) + c'; if you put the cursor on the second `+', the result is `c + (a + b)' (which the default simplifications will rearrange to `(c + a) + b'). The relevant rules are stored in the variable `CommuteRules`.

You may need to turn default simplifications off (with the m O command) in order to get the full benefit of j C. For example, commuting `a - b' produces `-b + a', but the default simplifications will "simplify" this right back to `a - b' if you don't turn them off. The same is true of some of the other manipulations described in this section.

The j N (`calc-sel-negate`) command replaces the selected term with the negative of that term, then adjusts the surrounding formula in order to preserve the meaning. For example, given `exp(a - b)' where `a - b' is selected, the result is `1 / exp(b - a)'. By contrast, selecting a term and using the regular n (`calc-change-sign`) command negates the term without adjusting the surroundings, thus changing the meaning of the formula as a whole. The rules variable is `NegateRules`.

The j & (`calc-sel-invert`) command is similar to j N except it takes the reciprocal of the selected term. For example, given `a - ln(b)' with `b' selected, the result is `a + ln(1/b)'. The rules variable is `InvertRules`.

The j E (`calc-sel-jump-equals`) command moves the selected term from one side of an equation to the other. Given `a + b = c + d' with `c' selected, the result is `a + b - c = d'. This command also works if the selected term is part of a `*', `/', or `^' formula. The relevant rules variable is `JumpRules`.

The j I (`calc-sel-isolate`) command isolates the selected term on its side of an equation. It uses the a S (`calc-solve-for`) command to solve the equation, and the Hyperbolic flag affects it in the same way. See section Solving Equations. When it applies, j I is often easier to use than j E. It understands more rules of algebra, and works for inequalities as well as equations.

The j * (`calc-sel-mult-both-sides`) command prompts for a formula using algebraic entry, then multiplies both sides of the selected quotient or equation by that formula. It simplifies each side with a s (`calc-simplify`) before re-forming the quotient or equation. You can suppress this simplification by providing any numeric prefix argument. There is also a j / (`calc-sel-div-both-sides`) which is similar to j * but dividing instead of multiplying by the factor you enter.

As a special feature, if the numerator of the quotient is 1, then the denominator is expanded at the top level using the distributive law (i.e., using the C-u -1 a x command). Suppose the formula on the stack is `1 / (sqrt(a) + 1)', and you wish to eliminate the square root in the denominator by multiplying both sides by `sqrt(a) - 1'. Calc's default simplifications would change the result `(sqrt(a) - 1) / (sqrt(a) - 1) (sqrt(a) + 1)' right back to the original form by cancellation; Calc expands the denominator to `sqrt(a) (sqrt(a) - 1) + sqrt(a) - 1' to prevent this. (You would now want to use an a x command to expand the rest of the way, whereupon the denominator would cancel out to the desired form, `a - 1'.) When the numerator is not 1, this initial expansion is not necessary because Calc's default simplifications will not notice the potential cancellation.

If the selection is an inequality, j * and j / will accept any factor, but will warn unless they can prove the factor is either positive or negative. (In the latter case the direction of the inequality will be switched appropriately.) See section Declarations, for ways to inform Calc that a given variable is positive or negative. If Calc can't tell for sure what the sign of the factor will be, it will assume it is positive and display a warning message.

For selections that are not quotients, equations, or inequalities, these commands pull out a multiplicative factor: They divide (or multiply) by the entered formula, simplify, then multiply (or divide) back by the formula.

The j + (`calc-sel-add-both-sides`) and j - (`calc-sel-sub-both-sides`) commands analogously add to or subtract from both sides of an equation or inequality. For other types of selections, they extract an additive factor. A numeric prefix argument suppresses simplification of the intermediate results.

The j U (`calc-sel-unpack`) command replaces the selected function call with its argument. For example, given `a + sin(x^2)' with `sin(x^2)' selected, the result is `a + x^2'. (The `x^2' will remain selected; if you wanted to change the `sin` to `cos`, just press C now to take the cosine of the selected part.)

The j v (`calc-sel-evaluate`) command performs the normal default simplifications on the selected sub-formula. These are the simplifications that are normally done automatically on all results, but which may have been partially inhibited by previous selection-related operations, or turned off altogether by the m O command. This command is just an auto-selecting version of the a v command (see section Algebraic Manipulation).

With a numeric prefix argument of 2, C-u 2 j v applies the a s (`calc-simplify`) command to the selected sub-formula. With a prefix argument of 3 or more, e.g., C-u j v applies the a e (`calc-simplify-extended`) command. See section Simplifying Formulas. With a negative prefix argument it simplifies at the top level only, just as with a v. Here the "top" level refers to the top level of the selected sub-formula.

The j " (`calc-sel-expand-formula`) command is to a " (see section Algebraic Manipulation) what j v is to a v.

You can use the j r (`calc-rewrite-selection`) command to define other algebraic operations on sub-formulas. See section Rewrite Rules.

Algebraic Manipulation

The commands in this section perform general-purpose algebraic manipulations. They work on the whole formula at the top of the stack (unless, of course, you have made a selection in that formula).

Many algebra commands prompt for a variable name or formula. If you answer the prompt with a blank line, the variable or formula is taken from top-of-stack, and the normal argument for the command is taken from the second-to-top stack level.

The a v (`calc-alg-evaluate`) command performs the normal default simplifications on a formula; for example, `a - -b' is changed to `a + b'. These simplifications are normally done automatically on all Calc results, so this command is useful only if you have turned default simplifications off with an m O command. See section Simplification Modes.

It is often more convenient to type =, which is like a v but which also substitutes stored values for variables in the formula. Use a v if you want the variables to ignore their stored values.

If you give a numeric prefix argument of 2 to a v, it simplifies as if in algebraic simplification mode. This is

equivalent to typing a s; see section Simplifying Formulas. If you give a numeric prefix of 3 or more, it uses extended simplification mode (a e).

If you give a negative prefix argument *-1*, *-2*, or *-3*, it simplifies in the corresponding mode but only works on the top-level function call of the formula. For example, `(2 + 3) * (2 + 3)' will simplify to `(2 + 3)^2', without simplifying the sub-formulas `2 + 3'. As another example, typing V R + to sum the vector `[1, 2, 3, 4]' produces the formula `reduce(add, [1, 2, 3, 4])' in no-simplify mode. Using a v will evaluate this all the way to 10; using C-u - a v will evaluate it only to `1 + 2 + 3 + 4'. (See section Reducing and Mapping Vectors.)

The = command corresponds to the `evalv` function, and the related N command, which is like = but temporarily disables symbolic (m s) mode during the evaluation, corresponds to the `evalvn` function. (These commands interpret their prefix arguments differently than a v; = treats the prefix as the number of stack elements to evaluate at once, and N treats it as a temporary different working precision.)

The `evalvn` function can take an alternate working precision as an optional second argument. This argument can be either an integer, to set the precision absolutely, or a vector containing a single integer, to adjust the precision relative to the current precision. Note that `evalvn` with a larger than current precision will do the calculation at this higher precision, but the result will as usual be rounded back down to the current precision afterward. For example, `evalvn(pi - 3.1415)' at a precision of 12 will return `9.265359e-5'; `evalvn(pi - 3.1415, 30)' will return `9.26535897932e-5' (computing a 25-digit result which is then rounded down to 12); and `evalvn(pi - 3.1415, [-2])' will return `9.2654e-5'.

The a " (`calc-expand-formula`) command expands functions into their defining formulas wherever possible. For example, `deg(x^2)' is changed to `180 x^2 / pi'. Most functions, like `sin` and `gcd`, are not defined by simple formulas and so are unaffected by this command. One important class of functions which *can* be expanded is the user-defined functions created by the Z F command. See section Programming with Formulas. Other functions which a " can expand include the probability distribution functions, most of the financial functions, and the hyperbolic and inverse hyperbolic functions. A numeric prefix argument affects a " in the same way as it does a v: A positive argument expands all functions in the formula and then simplifies in various ways; a negative argument expands and simplifies only the top-level function call.

The a M (`calc-map-equation`) [`mapeq`] command applies a given function or operator to one or more equations. It is analogous to V M, which operates on vectors instead of equations. see section Reducing and Mapping Vectors. For example, a M S changes `x = y+1' to `sin(x) = sin(y+1)', and a M + with `x = y+1' and 6 on the stack produces `x+6 = y+7'. With two equations on the stack, a M + would add the lefthand sides together and the righthand sides together to get the two respective sides of a new equation.

Mapping also works on inequalities. Mapping two similar inequalities produces another inequality of the same type. Mapping an inequality with an equation produces an inequality of the same type. Mapping a `<=' with a `<' or `!=' (not-equal) produces a `<'. If inequalities with opposite direction (e.g., `<' and `>') are mapped, the direction of the second inequality is reversed to match the first: Using a M + on `a < b' and `a > 2' reverses the latter to get `2 < a', which then allows the combination `a + 2 < b + a', which the a s command can then simplify to get `2 < b'.

Using a M *, a M /, a M n, or a M & to negate or invert an inequality will reverse the direction of the inequality. Other adjustments to inequalities are *not* done automatically; a M S will change `x < y' to `sin(x)

< sin(y)' even though this is not true for all values of the variables.

With the Hyperbolic flag, H a M [`mapeqp`] does a plain mapping operation without reversing the direction of any inequalities. Thus, H a M & would change x > 2 to 1/x > 0.5. (This change is mathematically incorrect, but perhaps you were fixing an inequality which was already incorrect.)

With the Inverse flag, I a M [`mapeqr`] always reverses the direction of the inequality. You might use I a M C to change `x < y' to `cos(x) > cos(y)' if you know you are working with small positive angles.

The a b (`calc-substitute`) [`subst`] command substitutes all occurrences of some variable or sub-expression of an expression with a new sub-expression. For example, substituting `sin(x)' with `cos(y)' in `2 sin(x)^2 + x sin(x) + sin(2 x)' produces `2 cos(y)^2 + x cos(y) + sin(2 x)'. Note that this is a purely structural substitution; the lone `x' and the `sin(2 x)' stayed the same because they did not look like `sin(x)'. See section Rewrite Rules, for a more general method for doing substitutions.

The a b command normally prompts for two formulas, the old one and the new one. If you enter a blank line for the first prompt, all three arguments are taken from the stack (new, then old, then target expression). If you type an old formula but then enter a blank line for the new one, the new formula is taken from top-of-stack and the target from second-to-top. If you answer both prompts, the target is taken from top-of-stack as usual.

Note that a b has no understanding of commutativity or associativity. The pattern `x+y' will not match the formula `y+x'. Also, `y+z' will not match inside the formula `x+y+z' because the `+' operator is left-associative, so the "deep structure" of that formula is `(x+y) + z'. Use d U (`calc-unformatted-language`) mode to see the true structure of a formula. The rewrite rule mechanism, discussed later, does not have these limitations.

As an algebraic function, `subst` takes three arguments: Target expression, old, new. Note that `subst` is always evaluated immediately, even if its arguments are variables, so if you wish to put a call to `subst` onto the stack you must turn the default simplifications off first (with m O).

Simplifying Formulas

The a s (`calc-simplify`) [`simplify`] command applies various algebraic rules to simplify a formula. This includes rules which are not part of the default simplifications because they may be too slow to apply all the time, or may not be desirable all of the time. For example, non-adjacent terms of sums are combined, as in `a + b + 2 a' to `b + 3 a', and some formulas like `sin(arcsin(x))' are simplified to `x'.

The sections below describe all the various kinds of algebraic simplifications Calc provides in full detail. None of Calc's simplification commands are designed to pull rabbits out of hats; they simply apply certain specific rules to put formulas into less redundant or more pleasing forms. Serious algebra in Calc must be done manually, usually with a combination of selections and rewrite rules. See section Rearranging Formulas using Selections. See section Rewrite Rules.

See section Simplification Modes, for commands to control what level of simplification occurs automatically. Normally only the "default simplifications" occur.

Default Simplifications

This section describes the "default simplifications," those which are normally applied to all results. For example, if you enter the variable x on the stack twice and push +, Calc's default simplifications automatically change x + x to 2 x.

The m O command turns off the default simplifications, so that x + x will remain in this form unless you give an explicit "simplify" command like = or a v. See section Algebraic Manipulation. The m D command turns the default simplifications back on.

The most basic default simplification is the evaluation of functions. For example, 2 + 3 is evaluated to 5, and `sqrt`(9) is evaluated to 3. Evaluation does not occur if the arguments to a function are somehow of the wrong type (`tan`([2,3,4]), range (`tan`(90)), or number (`tan`(3,5)), or if the function name is not recognized (`f`(5)), or if "symbolic" mode (see section Symbolic Mode) prevents evaluation (`sqrt`(2)).

Calc simplifies (evaluates) the arguments to a function before it simplifies the function itself. Thus `sqrt`(5+4) is simplified to `sqrt`(9) before the `sqrt` function itself is applied. There are very few exceptions to this rule: `quote`, `lambda`, and `condition` (the `::` operator) do not evaluate their arguments, `if` (the `? :` operator) does not evaluate all of its arguments, and `evalto` does not evaluate its lefthand argument.

Most commands apply the default simplifications to all arguments they take from the stack, perform a particular operation, then simplify the result before pushing it back on the stack. In the common special case of regular arithmetic commands like + and Q `[sqrt]`, the arguments are simply popped from the stack and collected into a suitable function call, which is then simplified (the arguments being simplified first as part of the process, as described above).

The default simplifications are too numerous to describe completely here, but this section will describe the ones that apply to the major arithmetic operators. This list will be rather technical in nature, and will probably be interesting to you only if you are a serious user of Calc's algebra facilities.

As well as the simplifications described here, if you have stored any rewrite rules in the variable `EvalRules` then these rules will also be applied before any built-in default simplifications. See section Automatic Rewrites, for details.

And now, on with the default simplifications:

Arithmetic operators like + and * always take two arguments in Calc's internal form. Sums and products of three or more terms are arranged by the associative law of algebra into a left-associative form for sums, ((a + b) + c) + d, and a right-associative form for products, a * (b * (c * d)). Formulas like (a + b) + (c + d) are rearranged to left-associative form, though this rarely matters since Calc's algebra commands are designed to hide the inner structure of sums and products as much as possible. Sums and products in their proper associative form will be written without parentheses in the examples below.

Sums and products are *not* rearranged according to the commutative law (a + b to b + a) except in a few special cases described below. Some algebra programs always rearrange terms into a canonical order, which enables them to see that a b + b a can be simplified to 2 a b. Calc assumes you have put the terms into the order you want and generally leaves that order alone, with the consequence that formulas like the above will only be simplified if you explicitly give the a s command. See section Algebraic Simplifications.

Differences a - b are treated like sums a + (-b) for purposes of simplification; one of the default simplifications is to rewrite a + (-b) or (-b) + a, where -b represents a "negative-looking" term, into a - b form. "Negative-looking" means negative numbers, negated formulas like -x, and products or quotients in which either term is negative-looking.

Other simplifications involving negation are -(-x) to x; -(a b) or -(a/b) where either a or b is negative-looking, simplified by negating that term, or else where a or b is any number, by negating that number; -(a + b) to -a - b, and -(b - a) to a - b. (This, and rewriting (-b) + a to a - b, are the only cases where the order of terms in a sum is changed by the default simplifications.)

The distributive law is used to simplify sums in some cases: a x + b x to (a + b) x, where a represents a number or an implicit 1 or *-1* (as in x or -x) and similarly for b. Use the a c, a f, or j M commands to merge sums with non-numeric coefficients using the distributive law.

The distributive law is only used for sums of two terms, or for adjacent terms in a larger sum. Thus a + b + b + c is simplified to a + 2 b + c, but a + b + c + b is not simplified. The reason is that comparing all terms of a sum with one another would require time proportional to the square of the number of terms; Calc relegates potentially slow operations like this to commands that have to be invoked explicitly, like a s.

Finally, a + 0 and 0 + a are simplified to a. A consequence of the above rules is that 0 - a is simplified to -a.

The products 1 a and a 1 are simplified to a; (-1) a and a (-1) are simplified to -a; 0 a and a 0 are simplified to 0, except that in matrix mode where a is not provably scalar the result is the generic zero matrix `idn(0)', and that if a is infinite the result is `nan'.

Also, (-a) b and a (-b) are simplified to -(a b), where this occurs for negated formulas but not for regular negative numbers.

Products are commuted only to move numbers to the front: a b 2 is commuted to 2 a b.

The product a (b + c) is distributed over the sum only if a and at least one of b and c are numbers: 2 (x + 3) goes to 2 x + 6. The formula (-a) (b - c), where -a is a negative number, is rewritten to a (c - b).

The distributive law of products and powers is used for adjacent terms of the product: x^a x^b goes to @c{x^{a+b}} x^(a+b) where a is a number, or an implicit 1 (as in x), or the implicit one-half of `sqrt`(x), and similarly for b. The result is written using `sqrt' or `1/sqrt' if the sum of the powers is 1/2 or -1/2, respectively. If the sum of the powers is zero, the product is simplified to 1 or to `idn(1)' if matrix mode is enabled.

The product of a negative power times anything but another negative power is changed to use division: @c{$x^{-2} y$} x^(-2) y goes to y / x^2 unless matrix mode is in effect and neither x nor y are scalar (in which case it is considered unsafe to rearrange the order of the terms).

Finally, a (b/c) is rewritten to (a b)/c, and also (a/b) c is changed to (a c)/b unless in matrix mode.

Simplifications for quotients are analogous to those for products. The quotient 0 / x is simplified to 0, with the same exceptions that were noted for 0 x. Likewise, x / 1 and x / (-1) are simplified to x and -x, respectively.

The quotient x / 0 is left unsimplified or changed to an infinite quantity, as directed by the current infinite mode. See section Infinite Mode.

The expression @c{a / b^{-c}} a / b^(-c) is changed to a b^c, where -c is any negative-looking pow
Also, 1 / b^c is changed to @c{b^{-c}} b^(-c) for any power c.

Also, (-a) / b and a / (-b) go to -(a/b); (a/b) / c goes to a / (b c); and a / (b/c) goes to (a c) / b unless matrix mode prevents this rearrangement. Similarly, a / (b:c) is simplified to (c:b) a for any fraction b:c.

The distributive law is applied to (a + b) / c only if c and at least one of a and b are numbers. Quotients of powers and square roots are distributed just as described for multiplication.

Quotients of products cancel only in the leading terms of the numerator and denominator. In other words, a x b / a y b is cancelled to x b / y b but not to x / y. Once again this is because full cancellation can be slow; use a s to cancel all terms of the quotient.

Quotients of negative-looking values are simplified according to (-a) / (-b) to a / b, (-a) / (b - c) to a / (c - b), and (a - b) / (-c) to (b - a) / c.

The formula x^0 is simplified to 1, or to `idn(1)' in matrix mode. The formula 0^x is simplified to 0 unless x is a negative number or complex number, in which case the result is an infinity or an unsimplified formula according to the current infinite mode. Note that 0^0 is an indeterminate form, as evidenced by the fact that the simplifications for x^0 and 0^x conflict when x=0.

Powers of products or quotients (a b)^c, (a/b)^c are distributed to a^c b^c, a^c / b^c only if c is an integer, or if either a or b are nonnegative real numbers. Powers of powers (a^b)^c are simplified to a^(b c) only when c is an integer and b c also evaluates to an integer. Without these restrictions these simplifications would not be safe because of problems with principal values. (In other words, @c{$((-3)^{1/2})^2$} ((-3)^1:2)^2 is safe to simplify, but ((-3)^2)^1:2 is not.) See section Declarations, for ways to inform Calc that your variables satisfy these requirements.

As a special case of this rule, `sqrt`(x)^n is simplified to x^(n/2) only for even integers n.

If a is known to be real, b is an even integer, and c is a half- or quarter-integer, then (a^b)^c is simplified to @c{$`abs`(a^{b c})$} `abs`(a^(b c)).

Also, (-a)^b is simplified to a^b if b is an even integer, or to -(a^b) if b is an odd integer, for any negative-looking expression -a.

Square roots `sqrt`(x) generally act like one-half powers x^1:2 for the purposes of the above-listed simplifications.

Also, note that @c{$1 / x^{1:2}$} 1 / x^1:2 is changed to @c{$x^{-1:2}$} x^(-1:2), but 1 / `sqrt`(x) is left alone.

Generic identity matrices (see section Matrix and Scalar Modes) are simplified by the following rules: `idn`(a) + b to a + b if b is provably scalar, or expanded out if b is a matrix; `idn`(a) + `idn`(b) to `idn`(a + b); -`idn`(a) to `idn`(-a); a `idn`(b) to `idn`(a b) if a is provably scalar, or to a b if a is provably non-scalar; `idn`(a) `idn`(b) to `idn`(a b); analogous simplifications for quotients involving `idn`; and `idn`(a)^n to `idn`(a^n) where n is an integer.

The `floor` function and other integer truncation functions vanish if the argument is provably integer-valued, so that `floor`(`round`(x)) simplifies to `round`(x). Also, combinations of `float`, `floor` and its friends, and `ffloor` and its friends, are simplified in appropriate ways. See section Integer

Truncation.

The expression `abs`(-x) changes to `abs`(x). The expression `abs`(`abs`(x)) changes to `abs`(x); in fact, `abs`(x) changes to x or -x if x is provably nonnegative or nonpositive (see section Declarations).

While most functions do not recognize the variable `i` as an imaginary number, the `arg` function does handle the two cases `arg`(`i`) and `arg`(-`i`) just for convenience.

The expression `conj`(`conj`(x)) simplifies to x. Various other expressions involving `conj`, `re`, and `im` are simplified, especially if some of the arguments are provably real or involve the constant `i`. For example, `conj`(a + b i) is changed to `conj`(a) - `conj`(b) i, or to a - b i if a and b are known to be real.

Functions like `sin` and `arctan` generally don't have any default simplifications beyond simply evaluating the functions for suitable numeric arguments and infinity. The a s command described in the next section does provide some simplifications for these functions, though.

One important simplification that does occur is that `ln`(`e`) is simplified to 1, and `ln`(`e`^x) is simplified to x for any x. This occurs even if you have stored a different value in the Calc variable `e'; but this would be a bad idea in any case if you were also using natural logarithms!

Among the logical functions, `!`*(a <= b)* changes to a > b and so on. Equations and inequalities where both sides are either negative-looking or zero are simplified by negating both sides and reversing the inequality. While it might seem reasonable to simplify !!x to x, this would not be valid in general because !!2 is 1, not 2.

Most other Calc functions have few if any default simplifications defined, aside of course from evaluation when the arguments are suitable numbers.

Algebraic Simplifications

The a s command makes simplifications that may be too slow to do all the time, or that may not be desirable all of the time. If you find these simplifications are worthwhile, you can type m A to have Calc apply them automatically.

This section describes all simplifications that are performed by the a s command. Note that these occur in addition to the default simplifications; even if the default simplifications have been turned off by an m O command, a s will turn them back on temporarily while it simplifies the formula.

There is a variable, `AlgSimpRules`, in which you can put rewrites to be applied by a s. Its use is analogous to `EvalRules`, but without the special restrictions. Basically, the simplifier does `a r AlgSimpRules' with an infinite repeat count on the whole expression being simplified, then it traverses the expression applying the built-in rules described below. If the result is different from the original expression, the process repeats with the default simplifications (including `EvalRules`), then `AlgSimpRules`, then the built-in simplifications, and so on.

Sums are simplified in two ways. Constant terms are commuted to the end of the sum, so that a + 2 + b changes to a + b + 2. The only exception is that a constant will not be commuted away from the first position of a difference, i.e., 2 - x is not commuted to -x + 2.

Also, terms of sums are combined by the distributive law, as in x + y + 2 x to y + 3 x. This always occurs

for adjacent terms, but a s compares all pairs of terms including non-adjacent ones.

Products are sorted into a canonical order using the commutative law. For example, b c a is commuted to a b c. This allows easier comparison of products; for example, the default simplifications will not change x y + y x to 2 x y, but a s will; it first rewrites the sum to x y + x y, and then the default simplifications are able to recognize a sum of identical terms.

The canonical ordering used to sort terms of products has the property that real-valued numbers, interval forms and infinities come first, and are sorted into increasing order. The V S command uses the same ordering when sorting a vector.

Sorting of terms of products is inhibited when matrix mode is turned on; in this case, Calc will never exchange the order of two terms unless it knows at least one of the terms is a scalar.

Products of powers are distributed by comparing all pairs of terms, using the same method that the default simplifications use for adjacent terms of products.

Even though sums are not sorted, the commutative law is still taken into account when terms of a product are being compared. Thus (x + y) (y + x) will be simplified to (x + y)^2. A subtle point is that (x - y) (y - x) will *not* be simplified to -(x - y)^2; Calc does not notice that one term can be written as a constant times the other, even if that constant is *-1*.

A fraction times any expression, (a:b) x, is changed to a quotient involving integers: a x / b. This is not done for floating-point numbers like 0.5, however. This is one reason why you may find it convenient to turn Fraction mode on while doing algebra; see section Fraction Mode.

Quotients are simplified by comparing all terms in the numerator with all terms in the denominator for possible cancellation using the distributive law. For example, a x^2 b / c x^3 d will cancel x^2 from both sides to get a b / c x d. (The terms in the denominator will then be rearranged to c d x as described above.) If there is any common integer or fractional factor in the numerator and denominator, it is cancelled out; for example, (4 x + 6) / 8 x simplifies to (2 x + 3) / 4 x.

Non-constant common factors are not found even by a s. To cancel the factor a in (a x + a) / a^2 you could first use j M on the product a x to Merge the numerator to a (1+x), which can then be simplified successfully.

Integer powers of the variable `i` are simplified according to the identity i^2 = -1. If you store a new value other than the complex number (0,1) in `i`, this simplification will no longer occur. This is done by a s instead of by default in case someone (unwisely) uses the name `i` for a variable unrelated to complex numbers; it would be unfortunate if Calc quietly and automatically changed this formula for reasons the user might not have been thinking of.

Square roots of integer or rational arguments are simplified in several ways. (Note that these will be left unevaluated only in Symbolic mode.) First, square integer or rational factors are pulled out so that `sqrt`(8) is rewritten as 2 sqrt(2). Conceptually speaking this implies factoring the argument into primes and moving pairs of primes out of the square root, but for reasons of efficiency Calc only looks for primes up to 29.

Square roots in the denominator of a quotient are moved to the numerator: 1 / `sqrt`(3) changes to `sqrt`(3) / 3. The same effect occurs for the square root of a fraction: `sqrt`(2:3) changes to `sqrt`(6) / 3.

The % (modulo) operator is simplified in several ways when the modulus M is a positive real number. First,

if the argument is of the form x + n for some real number n, then n is itself reduced modulo M. For example, `(x - 23) % 10' is simplified to `(x + 7) % 10'.

If the argument is multiplied by a constant, and this constant has a common integer divisor with the modulus, then this factor is cancelled out. For example, `12 x % 15' is changed to `3 (4 x % 5)' by factoring out 3. Also, `(12 x + 1) % 15' is changed to `3 ((4 x + 1:3) % 5)'. While these forms may not seem "simpler," they allow Calc to discover useful information about modulo forms in the presence of declarations.

If the modulus is 1, then Calc can use `int` declarations to evaluate the expression. For example, the idiom `x % 2' is often used to check whether a number is odd or even. As described above, `2 n % 2' and `(2 n + 1) % 2' are simplified to `2 (n % 1)' and `2 ((n + 1:2) % 1)', respectively; Calc can simplify these to 0 and 1 (respectively) if `n` has been declared to be an integer.

Trigonometric functions are simplified in several ways. First, `sin`(`arcsin`(x)) is simplified to x, and similarly for `cos` and `tan`. If the argument to `sin` is negative-looking, it is simplified to `-sin`(x), and similarly for `cos` and `tan`. Finally, certain special values of the argument are recognized; see section Trigonometric/Hyperbolic Functions.

Trigonometric functions of inverses of different trigonometric functions can also be simplified, as in `sin`(`arccos`(x)) to `sqrt`(1 - x^2).

Hyperbolic functions of their inverses and of negative-looking arguments are also handled, as are exponentials of inverse hyperbolic functions.

No simplifications for inverse trigonometric and hyperbolic functions are known, except for negative arguments of `arcsin`, `arctan`, `arcsinh`, and `arctanh`. Note that `arcsin`(`sin`(x)) can *not* safely change to x, since this only correct within an integer multiple of @c{2π} 2 pi radians or 360 degrees. However, `arcsinh`(`sinh`(x)) is simplified to x if x is known to be real.

Several simplifications that apply to logarithms and exponentials are that `exp`(`ln`(x)), @c{$e^{\ln(x)}$} e^`ln`(x), and 10^`log10`(x) all reduce to x. Also, `ln`(`exp`(x)), etc., can reduce to x if x is provably real. The form `exp`(x)^y is simplified to `exp`(x y). If x is a suitable multiple of @c{πi} pi i (as described above for the trigonometric functions), then `exp`(x) or e^x will be expanded. Finally, `ln`(x) is simplified to a form involving `pi` and `i` where x is provably negative, positive imaginary, or negative imaginary.

The error functions `erf` and `erfc` are simplified when their arguments are negative-looking or are calls to the `conj` function.

Equations and inequalities are simplified by cancelling factors of products, quotients, or sums on both sides. Inequalities change sign if a negative multiplicative factor is cancelled. Non-constant multiplicative factors as in a b = a c are cancelled from equations only if they are provably nonzero (generally because they were declared so; see section Declarations). Factors are cancelled from inequalities only if they are nonzero and their sign is known.

Simplification also replaces an equation or inequality with 1 or 0 ("true" or "false") if it can through the use of declarations. If x is declared to be an integer greater than 5, then x < 3, x = 3, and x = 7.5 are all simplified to 0, but x > 3 is simplified to 1. By a similar analysis, abs(x) >= 0 is simplified to 1, as is x^2 >= 0 if x is known to be real.

"Unsafe" Simplifications

The a e (`calc-simplify-extended`) [`esimplify`] command is like a s except that it applies some additional simplifications which are not "safe" in all cases. Use this only if you know the values in your formula lie in the restricted ranges for which these simplifications are valid. The symbolic integrator uses a e; one effect of this is that the integrator's results must be used with caution. Where an integral table will often attach conditions like "for positive a only," Calc (like most other symbolic integration programs) will simply produce an unqualified result.

Because a e's simplifications are unsafe, it is sometimes better to type C-u -3 a v, which does extended simplification only on the top level of the formula without affecting the sub-formulas. In fact, C-u -3 j v allows you to target extended simplification to any specific part of a formula.

The variable `ExtSimpRules` contains rewrites to be applied by the a e command. These are applied in addition to `EvalRules` and `AlgSimpRules`. (The a r AlgSimpRules step described above is simply followed by an a r ExtSimpRules step.)

Following is a complete list of "unsafe" simplifications performed by a e.

Inverse trigonometric or hyperbolic functions, called with their corresponding non-inverse functions as arguments, are simplified by a e. For example, `arcsin`(`sin`(x)) changes to x. Also, `arcsin`(`cos`(x)) and `arccos`(`sin`(x)) both change to `pi`/2 - x. These simplifications are unsafe because they are valid only for values of x in a certain range; outside that range, values are folded down to the 360-degree range that the inverse trigonometric functions always produce.

Powers of powers (x^a)^b are simplified to @c{$x^{a b}$} x^(a b) for all a and b. These results will be valid only in a restricted range of x; for example, in @c{$(x^2)^{1:2}$} (x^2)^1:2 the powers cancel to get x, which is valid for positive values of x but not for negative or complex values.

Similarly, `sqrt`(x^a) and `sqrt`(x)^a are both simplified (possibly unsafely) to @c{$x^{a/2}$} x^(a/2).

Forms like `sqrt`(1 - `sin`(x)^2) are simplified to, e.g., `cos`(x). Calc has identities of this sort for `sin`, `cos`, `tan`, `sinh`, and `cosh`.

Arguments of square roots are partially factored to look for squared terms that can be extracted. For example, `sqrt`(a^2 b^3 + a^3 b^2) simplifies to a b `sqrt`(a+b).

The simplifications of `ln`(`exp`(x)), `ln`(`e`^x), and `log10`(10^x) to x are also unsafe because of problems with principal values (although these simplifications are safe if x is known to be real).

Common factors are cancelled from products on both sides of an equation, even if those factors may be zero: a x / b x to a / b. Such factors are never cancelled from inequalities: Even a e is not bold enough to reduce a x < b x to a < b (or a > b, depending on whether you believe x is positive or negative). The a M / command can be used to divide a factor out of both sides of an inequality.

Simplification of Units

The simplifications described in this section are applied by the u s (`calc-simplify-units`) command. These are in addition to the regular a s (but not a e) simplifications described earlier. See section Basic Operations on Units.

The variable `UnitSimpRules` contains rewrites to be applied by the u s command. These are applied in addition to `EvalRules` and `AlgSimpRules`.

Scalar mode is automatically put into effect when simplifying units. See section Matrix and Scalar Modes.

Sums a + b involving units are simplified by extracting the units of a as if by the u x command (call the result u_a), then simplifying the expression b / u_a using u b and u s. If the result has units then the sum is inconsistent and is left alone. Otherwise, it is rewritten in terms of the units u_a.

If units auto-ranging mode is enabled, products or quotients in which the first argument is a number which is out of range for the leading unit are modified accordingly.

When cancelling and combining units in products and quotients, Calc accounts for unit names that differ only in the prefix letter. For example, `2 km m' is simplified to `2000 m^2'. However, compatible but different units like `ft` and `in` are not combined in this way.

Quotients a / b are simplified in three additional ways. First, if b is a number or a product beginning with a number, Calc computes the reciprocal of this number and moves it to the numerator.

Second, for each pair of unit names from the numerator and denominator of a quotient, if the units are compatible (e.g., they are both units of area) then they are replaced by the ratio between those units. For example, in `3 s in N / kg cm' the units `in / cm' will be replaced by 2.54.

Third, if the units in the quotient exactly cancel out, so that a u b command on the quotient would produce a dimensionless number for an answer, then the quotient simplifies to that number.

For powers and square roots, the "unsafe" simplifications (a b)^c to a^c b^c, (a/b)^c to a^c / b^c, and (a^b)^c to @c{$a^{b c}$} a^(b c) are done if the powers are real numbers. (These are safe in the context of units because all numbers involved can reasonably be assumed to be real.)

Also, if a unit name is raised to a fractional power, and the base units in that unit name all occur to powers which are a multiple of the denominator of the power, then the unit name is expanded out into its base units, which can then be simplified according to the previous paragraph. For example, `acre^1.5' is simplified by noting that 1.5 = 3:2, that `acre' is defined in terms of `m^2', and that the 2 in the power of `m` is a multiple of 2 in 3:2. Thus, `acre^1.5` is replaced by approximately @c{$(4046 m^2)^{1.5}$} (4046 m^2)^1.5, which is then changed to @c{$4046^{1.5} \, (m^2)^{1.5}$} 4046^1.5 (m^2)^1.5, then to 257440 m^3.

The functions `float`, `frac`, `clean`, `abs`, as well as `floor` and the other integer truncation functions, applied to unit names or products or quotients involving units, are simplified. For example, `round(1.6 in)' is changed to `round(1.6) round(in)'; the lefthand term evaluates to 2, and the righthand term simplifies to `in`.

The functions `sin`, `cos`, and `tan` with arguments that have angular units like `rad` or `arcmin` are simplified by converting to base units (radians), then evaluating with the angular mode temporarily set to radians.

Polynomials

A polynomial is a sum of terms which are coefficients times various powers of a "base" variable. For example, 2 x^2 + 3 x - 4 is a polynomial in x. Some formulas can be considered polynomials in several different variables: 1 + 2 x + 3 y + 4 x y^2 is a polynomial in both x and y. Polynomial coefficients are often numbers, but they may in general be any formulas not involving the base variable.

The a f (`calc-factor`) [`factor`] command factors a polynomial into a product of terms. For example, the polynomial x^3 + 2 x^2 + x is factored into `x*(x+1)^2'. As another example, a c + b d + b c + a d is factored into the product (a + b) (c + d).

Calc currently has three algorithms for factoring. Formulas which are linear in several variables, such as the second example above, are merged according to the distributive law. Formulas which are polynomials in a single variable, with constant integer or fractional coefficients, are factored into irreducible linear and/or quadratic terms. The first example above factors into three linear terms (x, x+1, and x+1 again). Finally, formulas which do not fit the above criteria are handled by the algebraic rewrite mechanism.

Calc's polynomial factorization algorithm works by using the general root-finding command (a P) to solve for the roots of the polynomial. It then looks for roots which are rational numbers or complex-conjugate pairs, and converts these into linear and quadratic terms, respectively. Because it uses floating-point arithmetic, it may be unable to find terms that involve large integers (whose number of digits approaches the current precision). Also, irreducible factors of degree higher than quadratic are not found, and polynomials in more than one variable are not treated. (A more robust factorization algorithm may be included in a future version of Calc.)

The rewrite-based factorization method uses rules stored in the variable `FactorRules`. See section Rewrite Rules, for a discussion of the operation of rewrite rules. The default `FactorRules` are able to factor quadratic forms symbolically into two linear terms, (a x + b) (c x + d). You can edit these rules to include other cases if you wish. To use the rules, Calc builds the formula `thecoefs(x, [a, b, c, ...])' where `x` is the polynomial base variable and `a`, `b`, etc., are polynomial coefficients (which may be numbers or formulas). The constant term is written first, i.e., in the `a` position. When the rules complete, they should have changed the formula into the form `thefactors(x, [f1, f2, f3, ...])' where each `fi` should be a factored term, e.g., `x - ai'. Calc then multiplies these terms together to get the complete factored form of the polynomial. If the rules do not change the `thecoefs` call to a `thefactors` call, a f leaves the polynomial alone on the assumption that it is unfactorable. (Note that the function names `thecoefs` and `thefactors` are used only as placeholders; there are no actual Calc functions by those names.)

The H a f [`factors`] command also factors a polynomial, but it returns a list of factors instead of an expression which is the product of the factors. Each factor is represented by a sub-vector of the factor, and the power with which it appears. For example, x^5 + x^4 - 33 x^3 + 63 x^2 factors to (x + 7) x^2 (x - 3)^2 in a f, or to [[x, 2], [x+7, 1], [x-3, 2]] in H a f. If there is an overall numeric factor, it always comes first in the list. The functions `factor` and `factors` allow a second argument when written in algebraic form; `factor(x,v)' factors x with respect to the specific variable v. The default is to factor with respect to all the variables that appear in x.

The a c (`calc-collect`) [`collect`] command rearranges a formula as a polynomial in a given variable, ordered in decreasing powers of that variable. For example, given 1 + 2 x + 3 y + 4 x y^2 on the stack, a c x would produce (2 + 4 y^2) x + (1 + 3 y), and a c y would produce (4 x) y^2 + 3 y + (1 + 2 x).

The polynomial will be expanded out using the distributive law as necessary: Collecting x in (x - 1)^3 produces x^3 - 3 x^2 + 3 x - 1. Terms not involving x will not be expanded.

The "variable" you specify at the prompt can actually be any expression: a c ln(x+1) will collect together all terms multiplied by `ln(x+1)' or integer powers thereof. If `x' also appears in the formula in a context other than `ln(x+1)', a c will treat those occurrences as unrelated to `ln(x+1)', i.e., as constants.

The a x (`calc-expand') [`expand'] command expands an expression by applying the distributive law everywhere. It applies to products, quotients, and powers involving sums. By default, it fully distributes all parts of the expression. With a numeric prefix argument, the distributive law is applied only the specified number of times, then the partially expanded expression is left on the stack.

The a x and j D commands are somewhat redundant. Use a x if you want to expand all products of sums in your formula. Use j D if you want to expand a particular specified term of the formula. There is an exactly analogous correspondence between a f and j M. (The j D and j M commands also know many other kinds of expansions, such as `exp(a + b) = exp(a) exp(b)', which a x and a f do not do.)

Calc's automatic simplifications will sometimes reverse a partial expansion. For example, the first step in expanding (x+1)^3 is to write (x+1) (x+1)^2. If a x stops there and tries to put this formula onto the stack, though, Calc will automatically simplify it back to (x+1)^3 form. The solution is to turn simplification off first (see section Simplification Modes), or to run a x without a numeric prefix argument so that it expands all the way in one step.

The a a (`calc-apart') [`apart'] command expands a rational function by partial fractions. A rational function is the quotient of two polynomials; `apart' pulls this apart into a sum of rational functions with simple denominators. In algebraic notation, the `apart' function allows a second argument that specifies which variable to use as the "base"; by default, Calc chooses the base variable automatically.

The a n (`calc-normalize-rat') [`nrat'] command attempts to arrange a formula into a quotient of two polynomials. For example, given 1 + (a + b/c) / d, the result would be (b + a c + c d) / c d. The quotient is reduced, so that a n will simplify (x^2 + 2x + 1) / (x^2 - 1) by dividing out the common factor x + 1, yielding (x + 1) / (x - 1).

The a \ (`calc-poly-div') [`pdiv'] command divides two polynomials u and v, yielding a new polynomial q. If several variables occur in the inputs, the inputs are considered multivariate polynomials. (Calc divides by the variable with the largest power in u first, or, in the case of equal powers, chooses the variables in alphabetical order.) For example, dividing x^2 + 3 x + 2 by x + 2 yields x + 1. The remainder from the division, if any, is reported at the bottom of the screen and is also placed in the Trail along with the quotient.

Using `pdiv' in algebraic notation, you can specify the particular variable to be used as the base: `pdiv(*a*,*b*,*x*)'. If `pdiv' is given only two arguments (as is always the case with the a \ command), then it does a multivariate division as outlined above.

The a % (`calc-poly-rem') [`prem'] command divides two polynomials and keeps the remainder r. The quotient q is discarded. For any formulas a and b, the results of a \ and a % satisfy a = q b + r. (This is analogous to plain \ and %, which compute the integer quotient and remainder from dividing two numbers.)

The a / (`calc-poly-div-rem') [`pdivrem'] command divides two polynomials and reports both the quotient and the remainder as a vector [q, r]. The H a / [`pdivide'] command divides two polynomials and

constructs the formula q + r/b on the stack. (Naturally if the remainder is zero, this will immediately simplify to q.)

The a g (`calc-poly-gcd`) [`pgcd`] command computes the greatest common divisor of two polynomials. (The GCD actually is unique only to within a constant multiplier; Calc attempts to choose a GCD which will be unsurprising.) For example, the a n command uses a g to take the GCD of the numerator and denominator of a quotient, then divides each by the result using a \. (The definition of GCD ensures that this division can take place without leaving a remainder.)

While the polynomials used in operations like a / and a g often have integer coefficients, this is not required. Calc can also deal with polynomials over the rationals or floating-point reals. Polynomials with modulo-form coefficients are also useful in many applications; if you enter `(x^2 + 3 x - 1) mod 5', Calc automatically transforms this into a polynomial over the field of integers mod 5: `(1 mod 5) x^2 + (3 mod 5) x + (4 mod 5)'.

Congratulations and thanks go to Ove Ewerlid (`ewerlid@mizar.DoCS.UU.SE`), who contributed many of the polynomial routines used in the above commands.

See section Decomposing Polynomials, for several useful functions for extracting the individual coefficients of a polynomial.

Calculus

The following calculus commands do not automatically simplify their inputs or outputs using `calc-simplify`. You may find it helps to do this by hand by typing a s or a e. It may also help to use a x and/or a c to arrange a result in the most readable way.

Differentiation

The a d (`calc-derivative`) [`deriv`] command computes the derivative of the expression on the top of the stack with respect to some variable, which it will prompt you to enter. Normally, variables in the formula other than the specified differentiation variable are considered constant, i.e., `deriv(y,x)' is reduced to zero. With the Hyperbolic flag, the `tderiv` (total derivative) operation is used instead, in which derivatives of variables are not reduced to zero unless those variables are known to be "constant," i.e., independent of any other variables. (The built-in special variables like `pi` are considered constant, as are variables that have been declared `const`; see section Declarations.)

With a numeric prefix argument n, this command computes the nth derivative.

When working with trigonometric functions, it is best to switch to radians mode first (with m r). The derivative of `sin(x)' in degrees is `(pi/180) cos(x)', probably not the expected answer!

If you use the `deriv` function directly in an algebraic formula, you can write `deriv(f,x,x0)' which represents the derivative of f with respect to x, evaluated at the point @c{$x=x_0$} x=x0.

If the formula being differentiated contains functions which Calc does not know, the derivatives of those functions are produced by adding primes (apostrophe characters). For example, `deriv(f(2x), x)' produces `2 f'(2 x)', where the function `f'` represents the derivative of `f`.

For functions you have defined with the Z F command, Calc expands the functions according to their defining formulas unless you have also defined `f'` suitably. For example, suppose we define `sinc(x) = sin(x)/x' using Z F. If we then differentiate the formula `sinc(2 x)', the formula will be expanded to `sin(2 x) / (2 x)' and differentiated. However, if we also define `sinc'(x) = dsinc(x)', say, then Calc will write the result as `2 dsinc(2 x)'. See section Programming with Formulas.

For multi-argument functions `f(x,y,z)', the derivative with respect to the first argument is written `f'(x,y,z)'; derivatives with respect to the other arguments are `f'2(x,y,z)' and `f'3(x,y,z)'. Various higher-order derivatives can be formed in the obvious way, e.g., `f'''(x) (the second derivative of `f`) or `f'''2'3(x,y,z) (`f` differentiated with respect to each argument once).

Integration

The a i (`calc-integral`) [`integ`] command computes the indefinite integral of the expression on the top of the stack with respect to a variable. The integrator is not guaranteed to work for all integrable functions, but it is able to integrate several large classes of formulas. In particular, any polynomial or rational function (a polynomial divided by a polynomial) is acceptable. (Rational functions don't have to be in explicit quotient form, however; @c{$x/(1+x^{-2})$} x/(1+x^-2) is not strictly a quotient of polynomials, but it is equivalent to x^3/(x^2+1), which is.) Also, square roots of terms involving x and x^2 may appear in rational functions being integrated. Finally, rational functions involving trigonometric or hyperbolic functions can be integrated.

Please note that the current implementation of Calc's integrator sometimes produces results that are significantly more complex than they need to be. For example, the integral Calc finds for @c{$1/(x+\sqrt{x^2+1})$} 1/(x+sqrt(x^2+1)) is several times more complicated than the answer Mathematica returns for the same input, although the two forms are numerically equivalent. Also, any indefinite integral should be considered to have an arbitrary constant of integration added to it, although Calc does not write an explicit constant of integration in its result. For example, Calc's solution for @c{$1/(1+\tan x)$} 1/(1+tan(x)) differs from the solution given in the *CRC Math Tables* by a constant factor of @c{$\pi i / 2$} pi i / 2, due to a different choice of constant of integration.

The Calculator remembers all the integrals it has done. If conditions change in a way that would invalidate the old integrals, say, a switch from degrees to radians mode, then they will be thrown out. If you suspect this is not happening when it should, use the `calc-flush-caches` command; see section Caches.

Calc normally will pursue integration by substitution or integration by parts up to 3 nested times before abandoning an approach as fruitless. If the integrator is taking too long, you can lower this limit by storing a number (like 2) in the variable `IntegLimit`. (The s I command is a convenient way to edit `IntegLimit`.) If this variable has no stored value or does not contain a nonnegative integer, a limit of 3 is used. The lower this limit is, the greater the chance that Calc will be unable to integrate a function it could otherwise handle. Raising this limit allows the Calculator to solve more integrals, though the time it takes may grow exponentially. You can monitor the integrator's actions by creating an Emacs buffer called `*Trace*`. If such a buffer exists, the a i command will write a log of its actions there.

If you want to manipulate integrals in a purely symbolic way, you can set the integration nesting limit to 0 to prevent all but fast table-lookup solutions of integrals. You might then wish to define rewrite rules for integration by parts, various kinds of substitutions, and so on. See section Rewrite Rules.

Customizing the Integrator

Calc has two built-in rewrite rules called `IntegRules` and `IntegAfterRules` which you can edit to define new integration methods. See section Rewrite Rules. At each step of the integration process, Calc wraps the current integrand in a call to the fictitious function `integtry(expr,var)', where expr is the integrand and var is the integration variable. If your rules rewrite this to be a plain formula (not a call to `integtry`), then Calc will use this formula as the integral of expr. For example, the rule `integtry(mysin(x),x) := -mycos(x)' would define a rule to integrate a function `mysin` that acts like the sine function. Then, putting `4 mysin(2y+1)' on the stack and typing a i y will produce the integral `-2 mycos(2y+1)'. Note that Calc has automatically made various transformations on the integral to allow it to use your rule; integral tables generally give rules for `mysin(a x + b)', but you don't need to use this much generality in your `IntegRules`.

As a more serious example, the expression `exp(x)/x' cannot be integrated in terms of the standard functions, so the "exponential integral" function @c{${\rm Ei}(x)$} Ei(x) was invented to describe it. We can get Calc to do this integral in terms of a made-up `Ei` function by adding the rule `[integtry(exp(x)/x, x) := Ei(x)]' to `IntegRules`. Now entering `exp(2x)/x' on the stack and typing a i x yields `Ei(2 x)'. This new rule will work with Calc's various built-in integration methods (such as integration by substitution) to solve a variety of other problems involving `Ei`: For example, now Calc will also be able to integrate `exp(exp(x))' and `ln(ln(x))' (to get `Ei(exp(x))' and `x ln(ln(x)) - Ei(ln(x))', respectively).

Your rule may do further integration by calling `integ`. For example, `integtry(twice(u),x) := twice(integ(u))' allows Calc to integrate `twice(sin(x))' to get `twice(-cos(x))'. Note that `integ` was called with only one argument. This notation is allowed only within `IntegRules`; it means "integrate this with respect to the same integration variable." If Calc is unable to integrate `u`, the integration that invoked `IntegRules` also fails. Thus integrating `twice(f(x))' fails, returning the unevaluated integral `integ(twice(f(x)), x)'. It is still legal to call `integ` with two or more arguments, however; in this case, if `u` is not integrable, `twice` itself will still be integrated: If the above rule is changed to `... := twice(integ(u,x))', then integrating `twice(f(x))' will yield `twice(integ(f(x),x))'.

If a rule instead produces the formula `integsubst(sexpr, svar)', either replacing the top-level `integtry` call or nested anywhere inside the expression, then Calc will apply the substitution `u = sexpr(svar)' to try to integrate the original expr. For example, the rule `sqrt(a) := integsubst(sqrt(x),x)' says that if Calc ever finds a square root in the integrand, it should attempt the substitution `u = sqrt(x)'. (This particular rule is unnecessary because Calc always tries "obvious" substitutions where sexpr actually appears in the integrand.) The variable svar may be the same as the var that appeared in the call to `integtry`, but it need not be.

When integrating according to an `integsubst`, Calc uses the equation solver to find the inverse of sexpr (if the integrand refers to var anywhere except in subexpressions that exactly match sexpr). It uses the differentiator to find the derivative of sexpr and/or its inverse (it has two methods that use one derivative or the other). You can also specify these items by adding extra arguments to the `integsubst` your rules construct; the general form is `integsubst(sexpr, svar, sinv, sprime)', where sinv is the inverse of sexpr (still written as a function of svar), and sprime is the derivative of sexpr with respect to svar. If you don't specify these things, and Calc is not able to work them out on its own with the information it knows, then your substitution rule will work only in very specific, simple cases.

Calc applies `IntegRules` as if by C-u 1 a r IntegRules; in other words, Calc stops rewriting as soon as

any rule in your rule set succeeds. (If it weren't for this, the `integsubst(sqrt(x),x)' example above would keep on adding layers of `integsubst` calls forever!)

Another set of rules, stored in `IntegSimpRules`, are applied every time the integrator uses a s to simplify an intermediate result. For example, putting the rule `twice(x) := 2 x' into `IntegSimpRules` would tell Calc to convert the `twice` function into a form it knows whenever integration is attempted.

One more way to influence the integrator is to define a function with the Z F command (see section Programming with Formulas). Calc's integrator automatically expands such functions according to their defining formulas, even if you originally asked for the function to be left unevaluated for symbolic arguments. (Certain other Calc systems, such as the differentiator and the equation solver, also do this.)

Sometimes Calc is able to find a solution to your integral, but it expresses the result in a way that is unnecessarily complicated. If this happens, you can either use `integsubst` as described above to try to hint at a more direct path to the desired result, or you can use `IntegAfterRules`. This is an extra rule set that runs after the main integrator returns its result; basically, Calc does an a r IntegAfterRules on the result before showing it to you. (It also does an a s, without `IntegSimpRules`, after that to further simplify the result.) For example, Calc's integrator sometimes produces expressions of the form `ln(1+x) - ln(1-x)'; the default `IntegAfterRules` rewrite this into the more readable form `2 arctanh(x)'. Note that, unlike `IntegRules`, `IntegSimpRules` and `IntegAfterRules` are applied any number of times until no further changes are possible. Rewriting by `IntegAfterRules` occurs only after the main integrator has finished, not at every step as for `IntegRules` and `IntegSimpRules`.

Numerical Integration

If you want a purely numerical answer to an integration problem, you can use the a I (`calc-num-integral`) [`ninteg`] command. This command prompts for an integration variable, a lower limit, and an upper limit. Except for the integration variable, all other variables that appear in the integrand formula must have stored values. (A stored value, if any, for the integration variable itself is ignored.)

Numerical integration works by evaluating your formula at many points in the specified interval. Calc uses an "open Romberg" method; this means that it does not evaluate the formula actually at the endpoints (so that it is safe to integrate `sin(x)/x' from zero, for example). Also, the Romberg method works especially well when the function being integrated is fairly smooth. If the function is not smooth, Calc will have to evaluate it at quite a few points before it can accurately determine the value of the integral.

Integration is much faster when the current precision is small. It is best to set the precision to the smallest acceptable number of digits before you use a I. If Calc appears to be taking too long, press C-g to halt it and try a lower precision. If Calc still appears to need hundreds of evaluations, check to make sure your function is well-behaved in the specified interval.

It is possible for the lower integration limit to be `-inf' (minus infinity). Likewise, the upper limit may be plus infinity. Calc internally transforms the integral into an equivalent one with finite limits. However, integration to or across singularities is not supported: The integral of `1/sqrt(x)' from 0 to 1 exists (it can be found by Calc's symbolic integrator, for example), but a I will fail because the integrand goes to infinity at one of the endpoints.

Taylor Series

The a t (`calc-taylor`) [`taylor`] command computes a power series expansion or Taylor series of a function. You specify the variable and the desired number of terms. You may give an expression of the form `var = a' or `var - a' instead of just a variable to produce a Taylor expansion about the point a. You may specify the number of terms with a numeric prefix argument; otherwise the command will prompt you for the number of terms. Note that many series expansions have coefficients of zero for some terms, so you may appear to get fewer terms than you asked for.

If the a i command is unable to find a symbolic integral for a function, you can get an approximation by integrating the function's Taylor series.

Solving Equations

The a S (`calc-solve-for`) [`solve`] command rearranges an equation to solve for a specific variable. An equation is an expression of the form L = R. For example, the command a S x will rearrange y = 3x + 6 to the form, x = y/3 - 2. If the input is not an equation, it is treated like an equation of the form X = 0.

This command also works for inequalities, as in y < 3x + 6. Some inequalities cannot be solved where the analogous equation could be; for example, solving @c{$a < b \, c$} a < b c for b is impossible without knowing the sign of c. In this case, a S will produce the result @c{$b \mathbin{\hbox{\code{!=}}} a/c$} b != a/c (using the not-equal-to operator) to signify that the direction of the inequality is now unknown. The inequality @c{$a \le b \, c$} a <= b c is not even partially solved. See section Declarations, for a way to tell Calc that the signs of the variables in a formula are in fact known.

Two useful commands for working with the result of a S are a . (see section Logical Operations), which converts x = y/3 - 2 to y/3 - 2, and s l (see section The Let Command) which evaluates another formula with x set equal to y/3 - 2.

Multiple Solutions

Some equations have more than one solution. The Hyperbolic flag (`H a S`) [`fsolve`] tells the solver to report the fully general family of solutions. It will invent variables `n1`, `n2`, ..., which represent independent arbitrary integers, and `s1`, `s2`, ..., which represent independent arbitrary signs (either *+1* or *-1*). If you don't use the Hyperbolic flag, Calc will use zero in place of all arbitrary integers, and plus one in place of all arbitrary signs. Note that variables like `n1` and `s1` are not given any special interpretation in Calc except by the equation solver itself. As usual, you can use the s l (`calc-let`) command to obtain solutions for various actual values of these variables.

For example, ' x^2 = y RET H a S x RET solves to get `x = s1 sqrt(y)', indicating that the two solutions to the equation are `sqrt(y)' and `-sqrt(y)'. Another way to think about it is that the square-root operation is really a two-valued function; since every Calc function must return a single result, `sqrt` chooses to return the positive result. Then H a S doctors this result using `s1` to indicate the full set of possible values of the mathematical square-root.

There is a similar phenomenon going the other direction: Suppose we solve `sqrt(y) = x' for `y`. Calc squares both sides to get `y = x^2'. This is correct, except that it introduces some dubious solutions. Consider

solving `sqrt(y) = -3': Calc will report y = 9 as a valid solution, which is true in the mathematical sense square-root, but false (there is no solution) for the actual Calc positive-valued `sqrt`. This happens for a S and H a S.

If you store a positive integer in the Calc variable `GenCount`, then Calc will generate formulas of the form `as(n)' for arbitrary signs, and `an(n)' for arbitrary integers, where n represents successive values taken by incrementing `GenCount` by one. While the normal arbitrary sign and integer symbols start over at `s1` and `n1` with each new Calc command, the `GenCount` approach will give each arbitrary value a name that is unique throughout the entire Calc session. Also, the arbitrary values are function calls instead of variables, which is advantageous in some cases. For example, you can make a rewrite rule that recognizes all arbitrary signs using a pattern like `as(n)'. The s l command only works on variables, but you can use the a b (`calc-substitute`) command to substitute actual values for function calls like `as(3)'.

The s G (`calc-edit-GenCount`) command is a convenient way to create or edit this variable. Press M-# M-# to finish.

If you have not stored a value in `GenCount`, or if the value in that variable is not a positive integer, the regular `s1`/`n1` notation is used.

With the Inverse flag, I a S [`finv`] treats the expression on top of the stack as a function of the specified variable and solves to find the inverse function, written in terms of the same variable. For example, I a S x inverts 2x + 6 to x/2 - 3. You can use both Inverse and Hyperbolic [`ffinv`] to obtain a fully general inverse, as described above.

Some equations, specifically polynomials, have a known, finite number of solutions. The a P (`calc-poly-roots`) [`roots`] command uses H a S to solve an equation in general form, then, for all arbitrary-sign variables like `s1`, and all arbitrary-integer variables like `n1` for which `n1` only usefully varies over a finite range, it expands these variables out to all their possible values. The results are collected into a vector, which is returned. For example, `roots(x^4 = 1, x)' returns the four solutions `[1, -1, (0, 1), (0, -1)]'. Generally an nth degree polynomial will always have n roots on the complex plane. (If you have given a `real` declaration for the solution variable, then only the real-valued solutions, if any, will be reported; see section Declarations.)

Note that because a P uses H a S, it is able to deliver symbolic solutions if the polynomial has symbolic coefficients. Also note that Calc's solver is not able to get exact symbolic solutions to all polynomials. Polynomials containing powers up to x^4 can always be solved exactly; polynomials of higher degree sometimes can be: x^6 + x^3 + 1 is converted to (x^3)^2 + (x^3) + 1, which can be solved for x^3 using the quadratic equation, and then for x by taking cube roots. But in many cases, like x^6 + x + 1, Calc does not know how to rewrite the polynomial into a form it can solve. The a P command can still deliver a list of numerical roots, however, provided that symbolic mode (m s) is not turned on. (If you work with symbolic mode on, recall that the N (`calc-eval-num`) key is a handy way to reevaluate the formula on the stack with symbolic mode temporarily off.) Naturally, a P can only provide numerical roots if the polynomial coefficents are all numbers (real or complex).

Solving Systems of Equations

You can also use the commands described above to solve systems of simultaneous equations. Just create a vector of equations, then specify a vector of variables for which to solve. (You can omit the surrounding

brackets when entering the vector of variables at the prompt.)

For example, putting `[x + y = a, x - y = b]' on the stack and typing a S x,y RET produces the vector of solutions `[x = a - (a-b)/2, y = (a-b)/2]'. The result vector will have the same length as the variables vector, and the variables will be listed in the same order there. Note that the solutions are not always simplified as far as possible; the solution for x here could be improved by an application of the a n command.

Calc's algorithm works by trying to eliminate one variable at a time by solving one of the equations for that variable and then substituting into the other equations. Calc will try all the possibilities, but you can speed things up by noting that Calc first tries to eliminate the first variable with the first equation, then the second variable with the second equation, and so on. It also helps to put the simpler (e.g., more linear) equations toward the front of the list. Calc's algorithm will solve any system of linear equations, and also many kinds of nonlinear systems.

Normally there will be as many variables as equations. If you give fewer variables than equations (an "over-determined" system of equations), Calc will find a partial solution. For example, typing a S y RET with the above system of equations would produce `[y = a - x]'. There are now several ways to express this solution in terms of the original variables; Calc uses the first one that it finds. You can control the choice by adding variable specifiers of the form `elim(v)' to the variables list. This says that v should be eliminated from the equations; the variable will not appear at all in the solution. For example, typing a S y,elim(x) would yield `[y = a - (b+a)/2]'.

If the variables list contains only `elim` specifiers, Calc simply eliminates those variables from the equations and then returns the resulting set of equations. For example, a S elim(x) produces `[a - 2 y = b]'. Every variable eliminated will reduce the number of equations in the system by one.

Again, a S gives you one solution to the system of equations. If there are several solutions, you can use H a S to get a general family of solutions, or, if there is a finite number of solutions, you can use a P to get a list. (In the latter case, the result will take the form of a matrix where the rows are different solutions and the columns correspond to the variables you requested.)

Another way to deal with certain kinds of overdetermined systems of equations is the a F command, which does least-squares fitting to satisfy the equations. See section Curve Fitting.

Decomposing Polynomials

The `poly` function takes a polynomial and a variable as arguments, and returns a vector of polynomial coefficients (constant coefficient first). For example, `poly(x^3 + 2 x, x)' returns [0, 2, 0, 1]. If the input is not a polynomial in x, the call to `poly` is left in symbolic form. If the input does not involve the variable x, the input is returned in a list of length one, representing a polynomial with only a constant coefficient. The call `poly(x, x)' returns the vector [0, 1]. The last element of the returned vector is guaranteed to be nonzero; note that `poly(0, x)' returns the empty vector []. Note also that x may actually be any formula; for example, `poly(sin(x)^2 - sin(x) + 3, sin(x))' returns [3, -1, 1].

To get the x^k coefficient of polynomial p, use `poly(p, x)_(k+1)'. To get the degree of polynomial p, use `vlen(poly(p, x)) - 1'. For example, `poly((x+1)^4, x)' returns `[1, 4, 6, 4, 1]', so `poly((x+1)^4, x)_(2+1)' gives the x^2 coefficient of this polynomial, 6.

One important feature of the solver is its ability to recognize formulas which are "essentially" polynomials.

This ability is made available to the user through the `gpoly` function, which is used just like `poly`: `gpoly(expr, var)'. If expr is a polynomial in some term which includes var, then this function will return a vector `[x, c, a]' where x is the term that depends on var, c is a vector of polynomial coefficients (like the one returned by `poly`), and a is a multiplier which is usually 1. Basically, `expr = a*(c_1 + c_2 x + c_3 x^2 + ...)'. The last element of c is guaranteed to be non-zero, and c will not equal `[1]' (i.e., the trivial decomposition expr = x is not considered a polynomial). One side effect is that `gpoly(x, x)' and `gpoly(6, x)', both of which might be expected to recognize their arguments as polynomials, will not because the decomposition is considered trivial.

For example, `gpoly((x-2)^2, x)' returns `[x, [4, -4, 1], 1]', since the expanded form of this polynomial is 4 - 4 x + x^2.

The term x may itself be a polynomial in var. This is done to reduce the size of the c vector. For example, `gpoly(x^4 + x^2 - 1, x)' returns `[x^2, [-1, 1, 1], 1]', since a quadratic polynomial in x^2 is easier to solve than a quartic polynomial in x.

A few more examples of the kinds of polynomials `gpoly` can discover:

```
sin(x) - 1                [sin(x), [-1, 1], 1]
x + 1/x - 1               [x, [1, -1, 1], 1/x]
x + 1/x                   [x^2, [1, 1], 1/x]
x^3 + 2 x                 [x^2, [2, 1], x]
x + x^2:3 + sqrt(x)       [x^1:6, [1, 1, 0, 1], x^1:2]
x^(2a) + 2 x^a + 5        [x^a, [5, 2, 1], 1]
(exp(-x) + exp(x)) / 2    [e^(2 x), [0.5, 0.5], e^-x]
```

The `poly` and `gpoly` functions accept a third integer argument which specifies the largest degree of polynomial that is acceptable. If this is n, then only c vectors of length n+1 or less will be returned. Otherwise, the `poly` or `gpoly` call will remain in symbolic form. For example, the equation solver can handle quartics and smaller polynomials, so it calls `gpoly(expr, var, 4)' to discover whether expr can be treated by its linear, quadratic, cubic, or quartic formulas.

The `pdeg` function computes the degree of a polynomial; `pdeg(p,x)' is the highest power of `x` that appears in `p`. This is the same as `vlen(poly(p,x))-1', but is much more efficient. If `p` is constant with respect to `x`, then `pdeg(p,x) = 0'. If `p` is not a polynomial in `x` (e.g., `pdeg(2 cos(x), x)', the function remains unevaluated. It is possible to omit the second argument `x`, in which case `pdeg(p)' returns the highest total degree of any term of the polynomial, counting all variables that appear in `p`. Note that `pdeg(c) = pdeg(c,x) = 0` for any nonzero constant `c`; the degree of the constant zero is considered to be `-inf` (minus infinity).

The `plead` function finds the leading term of a polynomial. Thus `plead(p,x)' is equivalent to `poly(p,x)_vlen(poly(p,x))', though again more efficient. In particular, `plead((2x+1)^10, x)' returns 1024 without expanding out the list of coefficients. The value of `plead(p,x)` will be zero only if p = 0.

The `pcont` function finds the content of a polynomial. This is the greatest common divisor of all the coefficients of the polynomial. With two arguments, `pcont(p,x)` effectively uses `poly(p,x)' to get a list of coefficients, then uses `pgcd` (the polynomial GCD function) to combine these into an answer. For example, `pcont(4 x y^2 + 6 x^2 y, x)' is `2 y'. The content is basically the "biggest" polynomial that can be divided into `p` exactly. The sign of the content is the same as the sign of the leading coefficient.

With only one argument, `pcont(p)' computes the numerical content of the polynomial, i.e., the `gcd` of all numerical coefficients of all the terms in the formula. Note that `gcd` is defined on rational numbers as well as integers; it computes the `gcd` of the numerators and the `lcm` of the denominators. Thus `pcont(4:3 x y^2 + 6 x^2 y)' returns 2:3. Dividing the polynomial by this number will clear all the denominators, as well as dividing by any common content in the numerators. The numerical content of a polynomial is negative only if all the coefficients in the polynomial are negative.

The `pprim` function finds the primitive part of a polynomial, which is simply the polynomial divided (using `pdiv` if necessary) by its content. If the input polynomial has rational coefficients, the result will have integer coefficients in simplest terms.

Numerical Solutions

Not all equations can be solved symbolically. The commands in this section use numerical algorithms that can find a solution to a specific instance of an equation to any desired accuracy. Note that the numerical commands are slower than their algebraic cousins; it is a good idea to try a S before resorting to these commands.

(See section Curve Fitting, for some other, more specialized, operations on numerical data.)

Root Finding

The a R (`calc-find-root`) [`root`] command finds a numerical solution (or root) of an equation. (This command treats inequalities the same as equations. If the input is any other kind of formula, it is interpreted as an equation of the form X = 0.)

The a R command requires an initial guess on the top of the stack, and a formula in the second-to-top position. It prompts for a solution variable, which must appear in the formula. All other variables that appear in the formula must have assigned values, i.e., when a value is assigned to the solution variable and the formula is evaluated with =, it should evaluate to a number. Any assigned value for the solution variable itself is ignored and unaffected by this command.

When the command completes, the initial guess is replaced on the stack by a vector of two numbers: The value of the solution variable that solves the equation, and the difference between the lefthand and righthand sides of the equation at that value. Ordinarily, the second number will be zero or very nearly zero. (Note that Calc uses a slightly higher precision while finding the root, and thus the second number may be slightly different from the value you would compute from the equation yourself.)

The v h (`calc-head`) command is a handy way to extract the first element of the result vector, discarding the error term.

The initial guess can be a real number, in which case Calc searches for a real solution near that number, or a complex number, in which case Calc searches the whole complex plane near that number for a solution, or it can be an interval form which restricts the search to real numbers inside that interval.

Calc tries to use a d to take the derivative of the equation. If this succeeds, it uses Newton's method. If the equation is not differentiable Calc uses a bisection method. (If Newton's method appears to be going astray, Calc switches over to bisection if it can, or otherwise gives up. In this case it may help to try again with a

slightly different initial guess.) If the initial guess is a complex number, the function must be differenti

If the formula (or the difference between the sides of an equation) is negative at one end of the interval you specify and positive at the other end, the root finder is guaranteed to find a root. Otherwise, Calc subdivides the interval into small parts looking for positive and negative values to bracket the root. When your guess is an interval, Calc will not look outside that interval for a root.

The H a R [`wroot`] command is similar to a R, except that if the initial guess is an interval for which the function has the same sign at both ends, then rather than subdividing the interval Calc attempts to widen it to enclose a root. Use this mode if you are not sure if the function has a root in your interval.

If the function is not differentiable, and you give a simple number instead of an interval as your initial guess, Calc uses this widening process even if you did not type the Hyperbolic flag. (If the function *is* differentiable, Calc uses Newton's method which does not require a bounding interval in order to work.)

If Calc leaves the `root` or `wroot` function in symbolic form on the stack, it will normally display an explanation for why no root was found. If you miss this explanation, press w (`calc-why`) to get it back.

Minimization

The a N (`calc-find-minimum`) [`minimize`] command finds a minimum value for a formula. It is very similar in operation to a R (`calc-find-root`): You give the formula and an initial guess on the stack, and are prompted for the name of a variable. The guess may be either a number near the desired minimum, or an interval enclosing the desired minimum. The function returns a vector containing the value of the the variable which minimizes the formula's value, along with the minimum value itself.

Note that this command looks for a *local* minimum. Many functions have more than one minimum; some, like @c{$x \sin x$} x sin(x), have infinitely many. In fact, there is no easy way to define the "global" minimum of @c{$x \sin x$} x sin(x) but Calc can still locate any particular local minimum for you. Calc basically goes downhill from the initial guess until it finds a point at which the function's value is greater both to the left and to the right. Calc does not use derivatives when minimizing a function.

If your initial guess is an interval and it looks like the minimum occurs at one or the other endpoint of the interval, Calc will return that endpoint only if that endpoint is closed; thus, minimizing 17 x over [2..3] will return [2, 38], but minimizing over (2..3] would report no minimum found. In general, you should use closed intervals to find literally the minimum value in that range of x, or open intervals to find the local minimum, if any, that happens to lie in that range.

Most functions are smooth and flat near their minimum values. Because of this flatness, if the current precision is, say, 12 digits, the variable can only be determined meaningfully to about six digits. Thus you should set the precision to twice as many digits as you need in your answer.

The H a N [`wminimize`] command, analogously to H a R, expands the guess interval to enclose a minimum rather than requiring that the minimum lie inside the interval you supply.

The a X (`calc-find-maximum`) [`maximize`] and H a X [`wmaximize`] commands effectively minimize the negative of the formula you supply.

The formula must evaluate to a real number at all points inside the interval (or near the initial guess if the guess is a number). If the initial guess is a complex number the variable will be minimized over the

complex numbers; if it is real or an interval it will be minimized over the reals.

Systems of Equations

The a R command can also solve systems of equations. In this case, the equation should instead be a vector of equations, the guess should instead be a vector of numbers (intervals are not supported), and the variable should be a vector of variables. You can omit the brackets while entering the list of variables. Each equation must be differentiable by each variable for this mode to work. The result will be a vector of two vectors: The variable values that solved the system of equations, and the differences between the sides of the equations with those variable values. There must be the same number of equations as variables. Since only plain numbers are allowed as guesses, the Hyperbolic flag has no effect when solving a system of equations.

It is also possible to minimize over many variables with a N (or maximize with a X). Once again the variable name should be replaced by a vector of variables, and the initial guess should be an equal-sized vector of initial guesses. But, unlike the case of multidimensional a R, the formula being minimized should still be a single formula, *not* a vector. Beware that multidimensional minimization is currently *very* slow.

Curve Fitting

The a F command fits a set of data to a model formula, such as y = m x + b where m and b are parameters to be determined. For a typical set of measured data there will be no single m and b that exactly fit the data; in this case, Calc chooses values of the parameters that provide the closest possible fit.

Linear Fits

The a F (`calc-curve-fit`) [`fit`] command attempts to fit a set of data (x and y vectors of numbers) to a straight line, polynomial, or other function of x. For the moment we will consider only the case of fitting to a line, and we will ignore the issue of whether or not the model was in fact a good fit for the data.

In a standard linear least-squares fit, we have a set of (x,y) data points that we wish to fit to the model y = m x + b by adjusting the parameters m and b to make the y values calculated from the formula be as close as possible to the actual y values in the data set. (In a polynomial fit, the model is instead, say, y = a x^3 + b x^2 + c x + d. In a multilinear fit, we have data points of the form (x_1,x_2,x_3,y) and our model is y = a x_1 + b x_2 + c x_3 + d. These will be discussed later.)

In the model formula, variables like x and x_2 are called the independent variables, and y is the dependent variable. Variables like m, a, and b are called the parameters of the model.

The a F command takes the data set to be fitted from the stack. By default, it expects the data in the form of a matrix. For example, for a linear or polynomial fit, this would be a @c{$2\times N$} 2xN matrix where the first row is a list of x values and the second row has the corresponding y values. For the multilinear fit shown above, the matrix would have four rows (x_1, x_2, x_3, and y, respectively).

If you happen to have an @c{$N\times2$} Nx2 matrix instead of a @c{$2\times N$} 2xN matrix, just press v t first to transpose the matrix.

After you type a F, Calc prompts you to select a model. For a linear fit, press the digit 1.

Calc then prompts for you to name the variables. By default it chooses high letters like x and y for independent variables and low letters like a and b for parameters. (The dependent variable doesn't need a name.) The two kinds of variables are separated by a semicolon. Since you generally care more about the names of the independent variables than of the parameters, Calc also allows you to name only those and let the parameters use default names.

For example, suppose the data matrix

is on the stack and we wish to do a simple linear fit. Type a F, then 1 for the model, then RET to use the default names. The result will be the formula 3 + 2 x on the stack. Calc has created the model expression a + b x, then found the optimal values of a and b to fit the data. (In this case, it was able to find an exact fit.) Calc then substituted those values for a and b in the model formula.

The a F command puts two entries in the trail. One is, as always, a copy of the result that went to the stack; the other is a vector of the actual parameter values, written as equations: [a = 3, b = 2], in case you'd rather read them in a list than pick them out of the formula. (You can type t y to move this vector to the stack; see section Trail Commands.)

Specifying a different independent variable name will affect the resulting formula: a F 1 k RET produces 3 + 2 k. Changing the parameter names (say, a F 1 k;b,m RET) will affect the equations that go into the trail.

To see what happens when the fit is not exact, we could change the number 13 in the data matrix to 14 and try the fit again. The result is:

```
2.6 + 2.2 x
```

Evaluating this formula, say with v x 5 RET TAB V M $ RET, shows a reasonably close match to the y-values in the data.

```
[4.8, 7., 9.2, 11.4, 13.6]
```

Since there is no line which passes through all the *N* data points, Calc has chosen a line that best approximates the data points using the method of least squares. The idea is to define the chi-square error measure

which is clearly zero if a + b x exactly fits all data points, and increases as various a + b x_i values fail to match the corresponding y_i values. There are several reasons why the summand is squared, one of them being to ensure that @c{$\chi^2 \ge 0$} chi^2 >= 0. Least-squares fitting simply chooses the values of a and b for which the error @c{χ^2} chi^2 is as small as possible.

Other kinds of models do the same thing but with a different model formula in place of a + b x_i.

A numeric prefix argument causes the a F command to take the data in some other form than one big matrix. A positive argument *N* will take *N* items from the stack, corresponding to the *N* rows of a data matrix. In the linear case, *N* must be 2 since there is always one independent variable and one dependent variable.

A prefix of zero or plain C-u is a compromise; Calc takes two items from the stack, an *N*-row matrix of x values, and a vector of y values. If there is only one independent variable, the x values can be either a

one-row matrix or a plain vector, in which case the C-u prefix is the same as a C-u 2 prefix.

Polynomial and Multilinear Fits

To fit the data to higher-order polynomials, just type one of the digits 2 through 9 when prompted for a model. For example, we could fit the original data matrix from the previous section (with 13, not 14) to a parabola instead of a line by typing a F 2 RET.

```
2.00000000001 x - 1.5e-12 x^2 + 2.99999999999
```

Note that since the constant and linear terms are enough to fit the data exactly, it's no surprise that Calc chose a tiny contribution for x^2. (The fact that it's not exactly zero is due only to roundoff error. Since our data are exact integers, we could get an exact answer by typing m f first to get fraction mode. Then the x^2 term would vanish altogether. Usually, though, the data being fitted will be approximate floats so fraction mode won't help.)

Doing the a F 2 fit on the data set with 14 instead of 13 gives a much larger x^2 contribution, as Calc bends the line slightly to improve the fit.

```
0.142857142855 x^2 + 1.34285714287 x + 3.59999999998
```

An important result from the theory of polynomial fitting is that it is always possible to fit *N* data points exactly using a polynomial of degree *N-1*, sometimes called an interpolating polynomial. Using the modified (14) data matrix, a model number of 4 gives a polynomial that exactly matches all five data points:

```
0.04167 x^4 - 0.4167 x^3 + 1.458 x^2 - 0.08333 x + 4.
```

The actual coefficients we get with a precision of 12, like 0.0416666663588, clearly suffer from loss of precision. It is a good idea to increase the working precision to several digits beyond what you need when you do a fitting operation. Or, if your data are exact, use fraction mode to get exact results.

You can type i instead of a digit at the model prompt to fit the data exactly to a polynomial. This just counts the number of columns of the data matrix to choose the degree of the polynomial automatically.

Fitting data "exactly" to high-degree polynomials is not always a good idea, though. High-degree polynomials have a tendency to wiggle uncontrollably in between the fitting data points. Also, if the exact-fit polynomial is going to be used to interpolate or extrapolate the data, it is numerically better to use the a p command described below. See section Polynomial Interpolation.

Another generalization of the linear model is to assume the y values are a sum of linear contributions from several x values. This is a multilinear fit, and it is also selected by the 1 digit key. (Calc decides whether the fit is linear or multilinear by counting the rows in the data matrix.)

Given the data matrix,

```
[ [  1,   2,   3,    4,   5  ]
  [  7,   2,   3,    5,   2  ]
```


```
  [ 14.5, 15, 18.5, 22.5, 24 ] ]
```

the command a F 1 RET will call the first row x and the second row y, and will fit the values in the third row to the model a + b x + c y.

```
8. + 3. x + 0.5 y
```

Calc can do multilinear fits with any number of independent variables (i.e., with any number of data rows).

Yet another variation is homogeneous linear models, in which the constant term is known to be zero. In the linear case, this means the model formula is simply a x; in the multilinear case, the model might be a x + b y + c z; and in the polynomial case, the model could be a x + b x^2 + c x^3. You can get a homogeneous linear or multilinear model by pressing the letter h followed by a regular model key, like 1 or 2.

It is certainly possible to have other constrained linear models, like 2.3 + a x or a - 4 x. While there is no single key to select models like these, a later section shows how to enter any desired model by hand. In the first case, for example, you would enter a F ' 2.3 + a x.

Another class of models that will work but must be entered by hand are multinomial fits, e.g., a + b x + c y + d x^2 + e y^2 + f x y.

Error Estimates for Fits

With the Hyperbolic flag, H a F `efit` performs the same fitting operation as a F, but reports the coefficients as error forms instead of plain numbers. Fitting our two data matrices (first with 13, then with 14) to a line with H a F gives the results,

```
3. + 2. x
2.6 +/- 0.382970843103 + 2.2 +/- 0.115470053838 x
```

In the first case the estimated errors are zero because the linear fit is perfect. In the second case, the errors are nonzero but moderately small, because the data are still very close to linear.

It is also possible for the *input* to a fitting operation to contain error forms. The data values must either all include errors or all be plain numbers. Error forms can go anywhere but generally go on the numbers in the last row of the data matrix. If the last row contains error forms `*y_i* +/- @c{σ_i} *sigma_i*', then the @c{χ^2} chi^2 statistic is now,

so that data points with larger error estimates contribute less to the fitting operation.

If there are error forms on other rows of the data matrix, all the errors for a given data point are combined; the square root of the sum of the squares of the errors forms the @c{σ_i} sigma_i used for the data point.

Both a F and H a F can accept error forms in the input matrix, although if you are concerned about error analysis you will probably use H a F so that the output also contains error estimates.

If the input contains error forms but all the @c{σ_i} sigma_i values are the same, it is easy to see that the resulting fitted model will be the same as if the input did not have error forms at all (@c{χ^2} chi^2 is simply scaled uniformly by @c{$1 / \sigma^2$} 1 / sigma^2, which doesn't affect where it has a

minimum). But there *will* be a difference in the estimated errors of the coefficients reported by H a F.

Consult any text on statistical modelling of data for a discussion of where these error estimates come from and how they should be interpreted.

With the Inverse flag, I a F [`xfit`] produces even more information. The result is a vector of six items:

1. The model formula with error forms for its coefficients or parameters. This is the result that H a F would have produced.
2. A vector of "raw" parameter values for the model. These are the polynomial coefficients or other parameters as plain numbers, in the same order as the parameters appeared in the final prompt of the I a F command. For polynomials of degree d, this vector will have length M = d+1 with the constant term first.
3. The covariance matrix C computed from the fit. This is an *M*x*M* symmetric matrix; the diagonal elements C_j_j are the variances @c{σ_j^2} sigma_j^2 of the parameters. The other elements are covariances @c{σ_{ij}^2} sigma_i_j^2 that describe the correlation between pairs of parameters. (A related set of numbers, the linear correlation coefficients @c{r_{ij}} r_i_j, are defined as @c{$\sigma_{ij}^2 / \sigma_i \, \sigma_j$} sigma_i_j^2 / sigma_i sigma_j.)
4. A vector of M "parameter filter" functions whose meanings are described below. If no filters are necessary this will instead be an empty vector; this is always the case for the polynomial and multilinear fits described so far.
5. The value of @c{χ^2} chi^2 for the fit, calculated by the formulas shown above. This gives a measure of the quality of the fit; statisticians consider @c{$\chi^2 \approx N - M$} chi^2 = N - M to indicate a moderately good fit (where again N is the number of data points and M is the number of parameters).
6. A measure of goodness of fit expressed as a probability Q. This is computed from the `utpc` probability distribution function using @c{χ^2} chi^2 with N - M degrees of freedom. A value of 0.5 implies a good fit; some texts recommend that often Q = 0.1 or even 0.001 can signify an acceptable fit. In particular, @c{χ^2} chi^2 statistics assume the errors in your inputs follow a normal (Gaussian) distribution; if they don't, you may have to accept smaller values of Q.

   The Q value is computed only if the input included error estimates. Otherwise, Calc will report the symbol `nan` for Q. The reason is that in this case the @c{χ^2} chi^2 value has effectively been used to estimate the original errors in the input, and thus there is no redundant information left over to use for a confidence test.

Standard Nonlinear Models

The a F command also accepts other kinds of models besides lines and polynomials. Some common models have quick single-key abbreviations; others must be entered by hand as algebraic formulas.

Here is a complete list of the standard models recognized by a F:

1

Linear or multilinear. *a + b x + c y + d z*.

2-9

Polynomials. *a + b x + c x^2 + d x^3*.

e

Exponential. *a* `exp`*(b x)* `exp`*(c y)*.

E

Base-10 exponential. *a* `10^`*(b x)* `10^`*(c y)*.

x

Exponential (alternate notation). `exp`*(a + b x + c y)*.

X

Base-10 exponential (alternate). `10^`*(a + b x + c y)*.

l

Logarithmic. *a + b* `ln`*(x) + c* `ln`*(y)*.

L

Base-10 logarithmic. *a + b* `log10`*(x) + c* `log10`*(y)*.

^

General exponential. *a b^x c^y*.

p

Power law. *a x^b y^c*.

q

Quadratic. *a + b (x-c)^2 + d (x-e)^2*.

g

Gaussian. @c{${a \over b \sqrt{2 \pi}} \exp\left(-{1 \over 2} \left(x - c \over b \right)^2 \right)$} *(a / b sqrt(2 pi)) exp(-0.5*((x-c)/b)^2)*.

All of these models are used in the usual way; just press the appropriate letter at the model prompt, and choose variable names if you wish. The result will be a formula as shown in the above table, with the best-fit values of the parameters substituted. (You may find it easier to read the parameter values from the vector that is placed in the trail.)

All models except Gaussian and polynomials can generalize as shown to any number of independent variables. Also, all the built-in models have an additive or multiplicative parameter shown as a in the above table which can be replaced by zero or one, as appropriate, by typing h before the model key.

Note that many of these models are essentially equivalent, but express the parameters slightly differently. For example, a b^x and the other two exponential models are all algebraic rearrangements of each other. Also, the "quadratic" model is just a degree-2 polynomial with the parameters expressed differently. Use whichever form best matches the problem.

The HP-28/48 calculators support four different models for curve fitting, called `LIN`, `LOG`, `EXP`, and `PWR`. These correspond to Calc models `a + b x', `a + b ln(x)', `a exp(b x)', and `a x^b', respectively. In each case, a is what the HP-48 identifies as the "intercept," and b is what it calls the "slope."

If the model you want doesn't appear on this list, press ' (the apostrophe key) at the model prompt to enter any algebraic formula, such as m x - b, as the model. (Not all models will work, though--see the next section for details.)

The model can also be an equation like y = m x + b. In this case, Calc thinks of all the rows of the data matrix on equal terms; this model effectively has two parameters (m and b) and two independent variables (x and y), with no "dependent" variables. Model equations do not need to take this y = form. For example, the implicit line equation a x + b y = 1 works fine as a model.

When you enter a model, Calc makes an alphabetical list of all the variables that appear in the model. These are used for the default parameters, independent variables, and dependent variable (in that order). If you enter a plain formula (not an equation), Calc assumes the dependent variable does not appear in the formula and thus does not need a name.

For example, if the model formula has the variables a,mu,sigma,t,x, and the data matrix has three rows (meaning two independent variables), Calc will use a,mu,sigma as the default parameters, and the data rows will be named t and x, respectively. If you enter an equation instead of a plain formula, Calc will use a,mu as the parameters, and sigma,t,x as the three independent variables.

You can, of course, override these choices by entering something different at the prompt. If you leave some variables out of the list, those variables must have stored values and those stored values will be used as constants in the model. (Stored values for the parameters and independent variables are ignored by the a F command.) If you list only independent variables, all the remaining variables in the model formula will become parameters.

If there are $ signs in the model you type, they will stand for parameters and all other variables (in alphabetical order) will be independent. Use $ for one parameter, $$ for another, and so on. Thus $ x + $$ is another way to describe a linear model.

If you type a $ instead of ' at the model prompt itself, Calc will take the model formula from the stack. (The data must then appear at the second stack level.) The same conventions are used to choose which variables in the formula are independent by default and which are parameters.

Models taken from the stack can also be expressed as vectors of two or three elements, [model, vars] or [model, vars, params]. Each of vars and params may be either a variable or a vector of variables. (If params is omitted, all variables in model except those listed as vars are parameters.)

When you enter a model manually with ', Calc puts a 3-vector describing the model in the trail so you can get it back if you wish.

Finally, you can store a model in one of the Calc variables `Model1` or `Model2`, then use this model by typing a F u or a F U (respectively). The value stored in the variable can be any of the formats that a F $ would accept for a model on the stack.

Calc uses the principal values of inverse functions like `ln` and `arcsin` when doing fits. For example, when you enter the model `y = sin(a t + b)' Calc actually uses the easier form `arcsin(y) = a t + b'. The `arcsin` function always returns results in the range from -*90* to 90 degrees (or the equivalent range in radians). Suppose you had data that you believed to represent roughly three oscillations of a sine wave, so that the argument of the sine might go from zero to @c{3×360} *3*360* degrees. The above model would appear to be a good way to determine the true frequency and phase of the sine wave, but in practice it would fail utterly. The righthand side of the actual model `arcsin(y) = a t + b' will grow smoothly with t, but the lefthand side will bounce back and forth between -*90* and 90. No values of a and b can make the two sides match, even approximately.

There is no good solution to this problem at present. You could restrict your data to small enough rang that the above problem doesn't occur (i.e., not straddling any peaks in the sine wave). Or, in this case, y could use a totally different method such as Fourier analysis, which is beyond the scope of the a F command. (Unfortunately, Calc does not currently have any facilities for taking Fourier and related transforms.)

Curve Fitting Details

Calc's internal least-squares fitter can only handle multilinear models. More precisely, it can handle any model of the form a f(x,y,z) + b g(x,y,z) + c h(x,y,z), where a,b,c are the parameters and x,y,z are the independent variables (of course there can be any number of each, not just three).

In a simple multilinear or polynomial fit, it is easy to see how to convert the model into this form. For example, if the model is a + b x + c x^2, then f(x) = 1, g(x) = x, and h(x) = x^2 are suitable functions.

For other models, Calc uses a variety of algebraic manipulations to try to put the problem into the form

```
Y(x,y,z) = A(a,b,c) F(x,y,z) + B(a,b,c) G(x,y,z) + C(a,b,c) H(x,y,z)
```

where Y,A,B,C,F,G,H are arbitrary functions. It computes Y, F, G, and H for all the data points, does a standard linear fit to find the values of A, B, and C, then uses the equation solver to solve for a,b,c in terms of A,B,C.

A remarkable number of models can be cast into this general form. We'll look at two examples here to see how it works. The power-law model y = a x^b with two independent variables and two parameters can be rewritten as follows:

```
y = a x^b
y = a exp(b ln(x))
y = exp(ln(a) + b ln(x))
ln(y) = ln(a) + b ln(x)
```

which matches the desired form with @c{$Y = \ln(y)$} Y = ln(y), @c{$A = \ln(a)$} A = ln(a), F = 1, B = b, and @c{$G = \ln(x)$} G = ln(x). Calc thus computes the logarithms of your y and x values, does a linear fit for A and B, then solves to get @c{$a = \exp(A)$} a = exp(A) and b = B.

Another interesting example is the "quadratic" model, which can be handled by expanding according to the distributive law.

```
y = a + b*(x - c)^2
y = a + b c^2 - 2 b c x + b x^2
```

which matches with Y = y, A = a + b c^2, F = 1, B = -2 b c, G = x (the -*2* factor could just as easily have been put into G instead of B), C = b, and H = x^2.

The Gaussian model looks quite complicated, but a closer examination shows that it's actually similar to the quadratic model but with an exponential that can be brought to the top and moved into Y.

An example of a model that cannot be put into general linear form is a Gaussian with a constant

background added on, i.e., d + the regular Gaussian formula. If you have a model like this, your best bet is to replace enough of your parameters with constants to make the model linearizable, then adjust the constants manually by doing a series of fits. You can compare the fits by graphing them, by examining the goodness-of-fit measures returned by I a F, or by some other method suitable to your application. Note that some models can be linearized in several ways. The Gaussian-plus-*d* model can be linearized by setting d (the background) to a constant, or by setting b (the standard deviation) and c (the mean) to constants.

To fit a model with constants substituted for some parameters, just store suitable values in those parameter variables, then omit them from the list of parameters when you answer the variables prompt.

A last desperate step would be to use the general-purpose `minimize` function rather than `fit`. After all, both functions solve the problem of minimizing an expression (the @c{χ^2} chi^2 sum) by adjusting certain parameters in the expression. The a F command is able to use a vastly more efficient algorithm due to its special knowledge about linear chi-square sums, but the a N command can do the same thing by brute force.

A compromise would be to pick out a few parameters without which the fit is linearizable, and use `minimize` on a call to `fit` which efficiently takes care of the rest of the parameters. The thing to be minimized would be the value of @c{χ^2} chi^2 returned as the fifth result of the `xfit` function:

```
minimize(xfit(gaus(a,b,c,d,x), x, [a,b,c], data)_5, d, guess)
```

where `gaus` represents the Gaussian model with background, `data` represents the data matrix, and `guess` represents the initial guess for d that `minimize` requires. This operation will only be, shall we say, extraordinarily slow rather than astronomically slow (as would be the case if `minimize` were used by itself to solve the problem).

The I a F [`xfit`] command is somewhat trickier when nonlinear models are used. The second item in the result is the vector of "raw" parameters A, B, C. The covariance matrix is written in terms of those raw parameters. The fifth item is a vector of filter expressions. This is the empty vector `[]' if the raw parameters were the same as the requested parameters, i.e., if A = a, B = b, and so on (which is always true if the model is already linear in the parameters as written, e.g., for polynomial fits). If the parameters had to be rearranged, the fifth item is instead a vector of one formula per parameter in the original model. The raw parameters are expressed in these "filter" formulas as `fitdummy(1)' for A, `fitdummy(2)' for B, and so on.

When Calc needs to modify the model to return the result, it replaces `fitdummy(1)' in all the filters with the first item in the raw parameters list, and so on for the other raw parameters, then evaluates the resulting filter formulas to get the actual parameter values to be substituted into the original model. In the case of H a F and I a F where the parameters must be error forms, Calc uses the square roots of the diagonal entries of the covariance matrix as error values for the raw parameters, then lets Calc's standard error-form arithmetic take it from there.

If you use I a F with a nonlinear model, be sure to remember that the covariance matrix is in terms of the raw parameters, *not* the actual requested parameters. It's up to you to figure out how to interpret the covariances in the presence of nontrivial filter functions.

Things are also complicated when the input contains error forms. Suppose there are three independent and dependent variables, x, y, and z, one or more of which are error forms in the data. Calc combines all the error values by taking the square root of the sum of the squares of the errors. It then changes x and y to be

plain numbers, and makes z into an error form with this combined error. The Y(x,y,z) part of the linear model is evaluated, and the result should be an error form. The error part of that result is used for @c{σ_i} sigma_i for the data point. If for some reason Y(x,y,z) does not return an error form, the combined error from z is used directly for @c{σ_i} sigma_i. Finally, z is also stripped of its error for use in computing F(x,y,z), G(x,y,z) and so on; the righthand side of the linearized model is computed in regular arithmetic with no error forms.

(While these rules may seem complicated, they are designed to do the most reasonable thing in the typical case that Y(x,y,z) depends only on the dependent variable z, and in fact is often simply equal to z. For common cases like polynomials and multilinear models, the combined error is simply used as the sigma for the data point with no further ado.)

It may be the case that the model you wish to use is linearizable, but Calc's built-in rules are unable to figure it out. Calc uses its algebraic rewrite mechanism to linearize a model. The rewrite rules are kept in the variable `FitRules`. You can edit this variable using the s e FitRules command; in fact, there is a special s F command just for editing `FitRules`. See section Other Operations on Variables.

See section Rewrite Rules, for a discussion of rewrite rules.

Calc uses `FitRules` as follows. First, it converts the model to an equation if necessary and encloses the model equation in a call to the function `fitmodel` (which is not actually a defined function in Calc; it is only used as a placeholder by the rewrite rules). Parameter variables are renamed to function calls `fitparam(1)', `fitparam(2)', and so on, and independent variables are renamed to `fitvar(1)', `fitvar(2)', etc. The dependent variable is the highest-numbered `fitvar`. For example, the power law model a x^b is converted to y = a x^b, then to

```
fitmodel(fitvar(2) = fitparam(1) fitvar(1)^fitparam(2))
```

Calc then applies the rewrites as if by `C-u 0 a r FitRules'. (The zero prefix means that rewriting should continue until no further changes are possible.)

When rewriting is complete, the `fitmodel` call should have been replaced by a `fitsystem` call that looks like this:

```
fitsystem(Y, FGH, abc)
```

where Y is a formula that describes the function Y(x,y,z), FGH is the vector of formulas [F(x,y,z), G(x,y,z), H(x,y,z)], and abc is the vector of parameter filters which refer to the raw parameters as `fitdummy(1)' for A, `fitdummy(2)' for B, etc. While the number of raw parameters (the length of the FGH vector) is usually the same as the number of original parameters (the length of the abc vector), this is not required.

The power law model eventually boils down to

```
fitsystem(ln(fitvar(2)),
          [1, ln(fitvar(1))],
          [exp(fitdummy(1)), fitdummy(2)])
```

The actual implementation of `FitRules` is complicated; it proceeds in four phases. First, common rearrangements are done to try to bring linear terms together and to isolate functions like `exp` and `ln` either

all the way "out" (so that they can be put into Y) or all the way "in" (so that they can be put into abc or FGH). In particular, all non-constant powers are converted to logs-and-exponentials form, and the distributive law is used to expand products of sums. Quotients are rewritten to use the `fitinv' function, where `fitinv(x)' represents 1/x while the `FitRules` are operating. (The use of `fitinv` makes recognition of linear-looking forms easier.) If you modify `FitRules`, you will probably only need to modify the rules for this phase.

Phase two, whose rules can actually also apply during phases one and three, first rewrites `fitmodel` to a two-argument form `fitmodel(Y, model)', where Y is initially zero and model has been changed from a=b to a-b form. It then tries to peel off invertible functions from the outside of model and put them into Y instead, calling the equation solver to invert the functions. Finally, when this is no longer possible, the `fitmodel` is changed to a four-argument `fitsystem`, where the fourth argument is model and the FGH and abc vectors are initially empty. (The last vector is really ABC, corresponding to raw parameters, for now.)

Phase three converts a sum of items in the model to a sum of `fitpart(a, b, c)' terms which represent terms `a*b*c' of the sum, where a is all factors that do not involve any variables, b is all factors that involve only parameters, and c is the factors that involve only independent variables. (If this decomposition is not possible, the rule set will not complete and Calc will complain that the model is too complex.) Then `fitparts` with equal b or c components are merged back together using the distributive law in order to minimize the number of raw parameters needed.

Phase four moves the `fitpart` terms into the FGH and ABC vectors. Also, some of the algebraic expansions that were done in phase 1 are undone now to make the formulas more computationally efficient. Finally, it calls the solver one more time to convert the ABC vector to an abc vector, and removes the fourth model argument (which by now will be zero) to obtain the three-argument `fitsystem` that the linear least-squares solver wants to see.

Two functions which are useful in connection with `FitRules` are `hasfitparams(x)' and `hasfitvars(x)', which check whether x refers to any parameters or independent variables, respectively. Specifically, these functions return "true" if the argument contains any `fitparam` (or `fitvar`) function calls, and "false" otherwise. (Recall that "true" means a nonzero number, and "false" means zero. The actual nonzero number returned is the largest n from all the `fitparam(n)'s or `fitvar(n)'s, respectively, that appear in the formula.)

The `fit` function in algebraic notation normally takes four arguments, `fit(model, vars, params, data)', where model is the model formula as it would be typed after a F ', vars is the independent variable or a vector of independent variables, params likewise gives the parameter(s), and data is the data matrix. Note that the length of vars must be equal to the number of rows in data if model is an equation, or one less than the number of rows if model is a plain formula. (Actually, a name for the dependent variable is allowed but will be ignored in the plain-formula case.)

If params is omitted, the parameters are all variables in model except those that appear in vars. If vars is also omitted, Calc sorts all the variables that appear in model alphabetically and uses the higher ones for vars and the lower ones for params.

Alternatively, `fit(modelvec, data)' is allowed where modelvec is a 2- or 3-vector describing the model and variables, as discussed previously.

If Calc is unable to do the fit, the `fit` function is left in symbolic form, ordinarily with an explanatory

message. The message will be "Model expression is too complex" if the linearizer was unable to put th model into the required form.

The `efit` (corresponding to H a F) and `xfit` (for I a F) functions are completely analogous.

Polynomial Interpolation

The a p (`calc-poly-interp`) [`polint`] command does a polynomial interpolation at a particular x value. It takes two arguments from the stack: A data matrix of the sort used by a F, and a single number which represents the desired x value. Calc effectively does an exact polynomial fit as if by a F i, then substitutes the x value into the result in order to get an approximate y value based on the fit. (Calc does not actually use a F i, however; it uses a direct method which is both more efficient and more numerically stable.)

The result of a p is actually a vector of two values: The y value approximation, and an error measure dy that reflects Calc's estimation of the probable error of the approximation at that value of x. If the input x is equal to any of the x values in the data matrix, the output y will be the corresponding y value from the matrix, and the output dy will be exactly zero.

A prefix argument of 2 causes a p to take separate x- and y-vectors from the stack instead of one data matrix.

If x is a vector of numbers, a p will return a matrix of interpolated results for each of those x values. (The matrix will have two columns, the y values and the dy values.) If x is a formula instead of a number, the `polint` function remains in symbolic form; use the a " command to expand it out to a formula that describes the fit in symbolic terms.

In all cases, the a p command leaves the data vectors or matrix on the stack. Only the x value is replaced by the result.

The H a p [`ratint`] command does a rational function interpolation. It is used exactly like a p, except that it uses as its model the quotient of two polynomials. If there are N data points, the numerator and denominator polynomials will each have degree N/2 (if N is odd, the denominator will have degree one higher than the numerator).

Rational approximations have the advantage that they can accurately describe functions that have poles (points at which the function's value goes to infinity, so that the denominator polynomial of the approximation goes to zero). If x corresponds to a pole of the fitted rational function, then the result will be a division by zero. If Infinite mode is enabled, the result will be `[uinf, uinf]'.

There is no way to get the actual coefficients of the rational function used by H a p. (The algorithm never generates these coefficients explicitly, and quotients of polynomials are beyond a F's capabilities to fit.)

Summations

The a + (`calc-summation`) [`sum`] command computes the sum of a formula over a certain range of index values. The formula is taken from the top of the stack; the command prompts for the name of the summation index variable, the lower limit of the sum (any formula), and the upper limit of the sum. If you enter a blank line at any of these prompts, that prompt and any later ones are answered by reading

additional elements from the stack. Thus, ' k^2 RET ' k RET 1 RET 5 RET a + RET produces the resul

The choice of index variable is arbitrary, but it's best not to use a variable with a stored value. In particular, while `i` is often a favorite index variable, it should be avoided in Calc because `i` has the imaginary constant (0, 1) as a value. If you pressed = on a sum over `i`, it would be changed to a nonsensical sum over the "variable" (0, 1)! If you really want to use `i` as an index variable, use s u i RET first to "unstore" this variable. (See section Storing Variables.)

A numeric prefix argument steps the index by that amount rather than by one. Thus ' a_k RET C-u -2 a + k RET 10 RET 0 RET yields `a_10 + a_8 + a_6 + a_4 + a_2 + a_0'. A prefix argument of plain C-u causes a + to prompt for the step value, in which case you can enter any formula or enter a blank line to take the step value from the stack. With the C-u prefix, a + can take up to five arguments from the stack: The formula, the variable, the lower limit, the upper limit, and (at the top of the stack), the step value.

Calc knows how to do certain sums in closed form. For example, `sum(6 k^2, k, 1, n) = 2 n^3 + 3 n^2 + n'. In particular, this is possible if the formula being summed is polynomial or exponential in the index variable. Sums of logarithms are transformed into logarithms of products. Sums of trigonometric and hyperbolic functions are transformed to sums of exponentials and then done in closed form. Also, of course, sums in which the lower and upper limits are both numbers can always be evaluated just by grinding them out, although Calc will use closed forms whenever it can for the sake of efficiency.

The notation for sums in algebraic formulas is `sum(expr, var, low, high, step)'. If step is omitted, it defaults to one. If high is omitted, low is actually the upper limit and the lower limit is one. If low is also omitted, the limits are `-inf' and `inf', respectively.

Infinite sums can sometimes be evaluated: `sum(.5^k, k, 1, inf)' returns 1. This is done by evaluating the sum in closed form (to `1. - 0.5^n' in this case), then evaluating this formula with `n` set to `inf`. Calc's usual rules for "infinite" arithmetic can find the answer from there. If infinite arithmetic yields a `nan', or if the sum cannot be solved in closed form, Calc leaves the `sum` function in symbolic form. See section Infinities.

As a special feature, if the limits are infinite (or omitted, as described above) but the formula includes vectors subscripted by expressions that involve the iteration variable, Calc narrows the limits to include only the range of integers which result in legal subscripts for the vector. For example, the sum `sum(k [a,b,c,d,e,f,g]_(2k),k)' evaluates to `b + 2 d + 3 f'.

The limits of a sum do not need to be integers. For example, `sum(a_k, k, 0, 2 n, n)' produces `a_0 + a_n + a_(2 n)'. Calc computes the number of iterations using the formula `1 + (high - low) / step', which must, after simplification as if by a s, evaluate to an integer.

If the number of iterations according to the above formula does not come out to an integer, the sum is illegal and will be left in symbolic form. However, closed forms are still supplied, and you are on your honor not to misuse the resulting formulas by substituting mismatched bounds into them. For example, `sum(k, k, 1, 10, 2)' is invalid, but Calc will go ahead and evaluate the closed form solution for the limits 1 and 10 to get the rather dubious answer, 29.25.

If the lower limit is greater than the upper limit (assuming a positive step size), the result is generally zero. However, Calc only guarantees a zero result when the upper limit is exactly one step less than the lower limit, i.e., if the number of iterations is *-1*. Thus `sum(f(k), k, n, n-1)' is zero but the sum from `n' to `n-2' may report a nonzero value if Calc used a closed form solution.

Calc's logical predicates like a < b return 1 for "true" and 0 for "false." See section Logical Operations. can be used to advantage for building conditional sums. For example, `sum(prime(k)*k^2, k, 1, 20)' is the sum of the squares of all prime numbers from 1 to 20; the `prime` predicate returns 1 if its argument is prime and 0 otherwise. You can read this expression as "the sum of k^2, where k is prime." Indeed, `sum(prime(k)*k^2, k)' would represent the sum of *all* primes squared, since the limits default to plus and minus infinity, but there are no such sums that Calc's built-in rules can do in closed form.

As another example, `sum((k != k_0) * f(k), k, 1, n)' is the sum of f(k) for all k from 1 to n, excluding one value k_0. Slightly more tricky is the summand `(k != k_0) / (k - k_0)', which is an attempt to describe the sum of all 1/(k-k_0) except at k = k_0, where this would be a division by zero. But at k = k_0, this formula works out to the indeterminate form 0 / 0, which Calc will not assume is zero. Better would be to use `(k != k_0) ? 1/(k-k_0) : 0'; the `? :' operator does an "if-then-else" test: This expression says, "if @c{$k \ne k_0$} k != k_0, then 1/(k-k_0), else zero." Now the formula 1/(k-k_0) will not even be evaluated by Calc when k = k_0.

The a - (`calc-alt-summation`) [`asum`] command computes an alternating sum. Successive terms of the sequence are given alternating signs, with the first term (corresponding to the lower index value) being positive. Alternating sums are converted to normal sums with an extra term of the form `(-1)^(k-low)'. This formula is adjusted appropriately if the step value is other than one. For example, the Taylor series for the sine function is `asum(x^k / k!, k, 1, inf, 2)'. (Calc cannot evaluate this infinite series, but it can approximate it if you replace `inf` with any particular odd number.) Calc converts this series to a regular sum with a step of one, namely `sum((-1)^k x^(2k+1) / (2k+1)!, k, 0, inf)'.

The a * (`calc-product`) [`prod`] command is the analogous way to take a product of many terms. Calc also knows some closed forms for products, such as `prod(k, k, 1, n) = n!'. Conditional products can be written `prod(k^prime(k), k, 1, n)' or `prod(prime(k) ? k : 1, k, 1, n)'.

The a T (`calc-tabulate`) [`table`] command evaluates a formula at a series of iterated index values, just like `sum` and `prod`, but its result is simply a vector of the results. For example, `table(a_i, i, 1, 7, 2)' produces `[a_1, a_3, a_5, a_7]'.

Logical Operations

The following commands and algebraic functions return true/false values, where 1 represents "true" and 0 represents "false." In cases where a truth value is required (such as for the condition part of a rewrite rule, or as the condition for a Z [Z] control structure), any nonzero value is accepted to mean "true." (Specifically, anything for which `dnonzero` returns 1 is "true," and anything for which `dnonzero` returns 0 or cannot decide is assumed "false." Note that this means that Z [Z] will execute the "then" portion if its condition is provably true, but it will execute the "else" portion for any condition like a = b that is not provably true, even if it might be true. Algebraic functions that have conditions as arguments, like `? :` and &&, remain unevaluated if the condition is neither provably true nor provably false. See section Declarations.)

The a = (`calc-equal-to`) command, or `eq(a,b)' function (which can also be written `a = b' or `a == b' in an algebraic formula) is true if a and b are equal, either because they are identical expressions, or because they are numbers which are numerically equal. (Thus the integer 1 is considered equal to the float 1.0.) If the equality of a and b cannot be determined, the comparison is left in symbolic form. Note that as a

command, this operation pops two values from the stack and pushes back either a 1 or a 0, or a formula b' if the values' equality cannot be determined.

Many Calc commands use `=' formulas to represent equations. For example, the a S (`calc-solve-for`) command rearranges an equation to solve for a given variable. The a M (`calc-map-equation`) command can be used to apply any function to both sides of an equation; for example, 2 a M * multiplies both sides of the equation by two. Note that just 2 * would not do the same thing; it would produce the formula `2 (a = b)' which represents 2 if the equality is true or zero if not.

The `eq` function with more than two arguments (e.g., C-u 3 a = or `a = b = c') tests if all of its arguments are equal. In algebraic notation, the `=' operator is unusual in that it is neither left- nor right-associative: `a = b = c' is not the same as `(a = b) = c' or `a = (b = c)' (which each compare one variable with the 1 or 0 that results from comparing two other variables).

The a # (`calc-not-equal-to`) command, or `neq(a,b)' or `a != b' function, is true if a and b are not equal. This also works with more than two arguments; `a != b != c != d' tests that all four of a, b, c, and d are distinct numbers.

The a < (`calc-less-than`) [`lt(a,b)' or `a < b'] operation is true if a is less than b. Similar functions are a > (`calc-greater-than`) [`gt(a,b)' or `a > b'], a [(`calc-less-equal`) [`leq(a,b)' or `a <= b'], and a] (`calc-greater-equal`) [`geq(a,b)' or `a >= b'].

While the inequality functions like `lt` do not accept more than two arguments, the syntax `a <= b < c' is translated to an equivalent expression involving intervals: `b in [a .. c)'. (See the description of `in` below.) All four combinations of `<' and `<=' are allowed, or any of the four combinations of `>' and `>='. Four-argument constructions like `a < b < c < d', and mixtures like `a < b = c' that involve both equalities and inequalities, are not allowed.

The a . (`calc-remove-equal`) [`rmeq`] command extracts the righthand side of the equation or inequality on the top of the stack. It also works elementwise on vectors. For example, if `[x = 2.34, y = z / 2]' is on the stack, then a . produces `[2.34, z / 2]'. As a special case, if the righthand side is a variable and the lefthand side is a number (as in `2.34 = x'), then Calc keeps the lefthand side instead. Finally, this command works with assignments `x := 2.34' as well as equations, always taking the the righthand side, and for `=>' (evaluates-to) operators, always taking the lefthand side.

The a & (`calc-logical-and`) [`land(a,b)' or `a && b'] function is true if both of its arguments are true, i.e., are non-zero numbers. In this case, the result will be either a or b, chosen arbitrarily. If either argument is zero, the result is zero. Otherwise, the formula is left in symbolic form.

The a | (`calc-logical-or`) [`lor(a,b)' or `a || b'] function is true if either or both of its arguments are true (nonzero). The result is whichever argument was nonzero, choosing arbitrarily if both are nonzero. If both a and b are zero, the result is zero.

The a ! (`calc-logical-not`) [`lnot(a)' or `! a'] function is true if a is false (zero), or false if a is true (nonzero). It is left in symbolic form if a is not a number.

The a : (`calc-logical-if`) [`if(a,b,c)' or `a ? b : c'] function is equal to either b or c if a is a nonzero number or zero, respectively. If a is not a number, the test is left in symbolic form and neither b nor c is evaluated in any way. In algebraic formulas, this is one of the few Calc functions whose arguments are not automatically evaluated when the function itself is evaluated. The others are `lambda`, `quote`, and

`condition`.

One minor surprise to watch out for is that the formula `a?3:4' will not work because the `3:4' is parsed as a fraction instead of as three separate symbols. Type something like `a ? 3 : 4' or `a?(3):4' instead.

As a special case, if a evaluates to a vector, then both b and c are evaluated; the result is a vector of the same length as a whose elements are chosen from corresponding elements of b and c according to whether each element of a is zero or nonzero. Each of b and c must be either a vector of the same length as a, or a non-vector which is matched with all elements of a.

The a { (`calc-in-set`) [`in(a,b)'] function is true if the number a is in the set of numbers represented by b. If b is an interval form, a must be one of the values encompassed by the interval. If b is a vector, a must be equal to one of the elements of the vector. (If any vector elements are intervals, a must be in any of the intervals.) If b is a plain number, a must be numerically equal to b. See section Set Operations using Vectors, for a group of commands that manipulate sets of this sort.

The `typeof(a)' function produces an integer or variable which characterizes a. If a is a number, vector, or variable, the result will be one of the following numbers:

```
 1   Integer
 2   Fraction
 3   Floating-point number
 4   HMS form
 5   Rectangular complex number
 6   Polar complex number
 7   Error form
 8   Interval form
 9   Modulo form
10   Date-only form
11   Date/time form
12   Infinity (inf, uinf, or nan)
100  Variable
101  Vector (but not a matrix)
102  Matrix
```

Otherwise, a is a formula, and the result is a variable which represents the name of the top-level function call.

The `integer(a)' function returns true if a is an integer. The `real(a)' function is true if a is a real number, either integer, fraction, or float. The `constant(a)' function returns true if a is any of the objects for which `typeof` would produce an integer code result except for variables, and provided that the components of an object like a vector or error form are themselves constant. Note that infinities do not satisfy any of these tests, nor do special constants like `pi` and `e`.

See section Declarations, for a set of similar functions that recognize formulas as well as actual numbers. For example, `dint(floor(x))' is true because `floor(x)' is provably integer-valued, but `integer(floor(x))' does not because `floor(x)' is not literally an integer constant.

The `refers(a,b)' function is true if the variable (or sub-expression) b appears in a, or false otherwise. Unlike

the other tests described here, this function returns a definite "no" answer even if its arguments are still in symbolic form. The only case where `refers` will be left unevaluated is if a is a plain variable (different from b).

The `negative(a)' function returns true if a "looks" negative, because it is a negative number, because it is of the form -x, or because it is a product or quotient with a term that looks negative. This is most useful in rewrite rules. Beware that `negative(a)' evaluates to 1 or 0 for *any* argument a, so it can only be stored in a formula if the default simplifications are turned off first with m O (or if it appears in an unevaluated context such as a rewrite rule condition).

The `variable(a)' function is true if a is a variable, or false if not. If a is a function call, this test is left in symbolic form. Built-in variables like `pi` and `inf` are considered variables like any others by this test.

The `nonvar(a)' function is true if a is a non-variable. If its argument is a variable it is left unsimplified; it never actually returns zero. However, since Calc's condition-testing commands consider "false" anything not provably true, this is often good enough.

The functions `lin`, `linnt`, `islin`, and `islinnt` check if an expression is "linear," i.e., can be written in the form a + b x for some constants a and b, and some variable or subformula x. The function `islin(f,x)' checks if formula f is linear in x, returning 1 if so. For example, `islin(x,x)', `islin(-x,x)', `islin(3,x)', and `islin(x y / 3 - 2, x)' all return 1. The `lin(f,x)' function is similar, except that instead of returning 1 it returns the vector [a, b, x]. For the above examples, this vector would be [0, 1, x], [0, -1, x], [3, 0, x], and [-2, y/3, x], respectively. Both `lin` and `islin` generally remain unevaluated for expressions which are not linear, e.g., `lin(2 x^2, x)' and `lin(sin(x), x)'. The second argument can also be a formula; `islin(2 + 3 sin(x), sin(x))' returns true.

The `linnt` and `islinnt` functions perform a similar check, but require a "non-trivial" linear form, which means that the b coefficient must be non-zero. For example, `lin(2,x)' returns [2, 0, x] and `lin(y,x)' returns [y, 0, x], but `linnt(2,x)' and `linnt(y,x)' are left unevaluated (in other words, these formulas are considered to be only "trivially" linear in x).

All four linearity-testing functions allow you to omit the second argument, in which case the input may be linear in any non-constant formula. Here, the a=0, b=1 case is also considered trivial, and only constant values for a and b are recognized. Thus, `lin(2 x y)' returns [0, 2, x y], `lin(2 - x y)' returns [2, -1, x y], and `lin(x y)' returns [0, 1, x y]. The `linnt` function would allow the first two cases but not the third. Also, neither `lin` nor `linnt` accept plain constants as linear in the one-argument case: `islin(2,x)' is true, but `islin(2)' is false.

The `istrue(a)' function returns 1 if a is a nonzero number or provably nonzero formula, or 0 if a is anything else. Calls to `istrue` can only be manipulated if m O mode is used to make sure they are not evaluated prematurely. (Note that declarations are used when deciding whether a formula is true; `istrue` returns 1 when `dnonzero` would return 1, and it returns 0 when `dnonzero` would return 0 or leave itself in symbolic form.)

Rewrite Rules

The a r (`calc-rewrite`) [`rewrite`] command makes substitutions in a formula according to a specified pattern or patterns known as rewrite rules. Whereas a b (`calc-substitute`) matches literally,

so that substituting `sin(x)' with `cos(x)' matches only the `sin` function applied to the variable `x`, rewri rules match general kinds of formulas; rewriting using the rule `sin(x) := cos(x)' matches `sin` of any argument and replaces it with `cos` of that same argument. The only significance of the name `x` is that the same name is used on both sides of the rule.

Rewrite rules rearrange formulas already in Calc's memory. See section Syntax Tables, to read about syntax rules, which are similar to algebraic rewrite rules but operate when new algebraic entries are being parsed, converting strings of characters into Calc formulas.

Entering Rewrite Rules

Rewrite rules normally use the "assignment" operator `old := new'. This operator is equivalent to the function call `assign(old, new)'. The `assign` function is undefined by itself in Calc, so an assignment formula such as a rewrite rule will be left alone by ordinary Calc commands. But certain commands, like the rewrite system, interpret assignments in special ways.

For example, the rule `sin(x)^2 := 1-cos(x)^2' says to replace every occurrence of the sine of something, squared, with one minus the square of the cosine of that same thing. All by itself as a formula on the stack it does nothing, but when given to the a r command it turns that command into a sine-squared-to-cosine-squared converter.

To specify a set of rules to be applied all at once, make a vector of rules.

When a r prompts you to enter the rewrite rules, you can answer in several ways:

1. With a rule: f(x) := g(x) RET.
2. With a vector of rules: [f1(x) := g1(x), f2(x) := g2(x)] RET. (You can omit the enclosing square brackets if you wish.)
3. With the name of a variable that contains the rule or rules vector: myrules RET.
4. With any formula except a rule, a vector, or a variable name; this will be interpreted as the old half of a rewrite rule, and you will be prompted a second time for the new half: f(x) RET g(x) RET.
5. With a blank line, in which case the rule, rules vector, or variable will be taken from the top of the stack (and the formula to be rewritten will come from the second-to-top position).

If you enter the rules directly (as opposed to using rules stored in a variable), those rules will be put into the Trail so that you can retrieve them later. See section Trail Commands.

It is most convenient to store rules you use often in a variable and invoke them by giving the variable name. The s e (`calc-edit-variable`) command is an easy way to create or edit a rule set stored in a variable. You may also wish to use s p (`calc-permanent-variable`) to save your rules permanently; see section Other Operations on Variables.

Rewrite rules are compiled into a special internal form for faster matching. If you enter a rule set directly it must be recompiled every time. If you store the rules in a variable and refer to them through that variable, they will be compiled once and saved away along with the variable for later reference. This is another good reason to store your rules in a variable.

Calc also accepts an obsolete notation for rules, as vectors `[old, new]'. But because it is easily confused with a vector of two rules, the use of this notation is no longer recommended.

Basic Rewrite Rules

To match a particular formula x with a particular rewrite rule `old := new', Calc compares the structure of x with the structure of old. Variables that appear in old are treated as meta-variables; the corresponding positions in x may contain any sub-formulas. For example, the pattern `f(x,y)' would match the expression `f(12, a+1)' with the meta-variable `x' corresponding to 12 and with `y' corresponding to `a+1'. However, this pattern would not match `f(12)' or `g(12, a+1)', since there is no assignment of the meta-variables that will make the pattern match these expressions. Notice that if the pattern is a single meta-variable, it will match any expression.

If a given meta-variable appears more than once in old, the corresponding sub-formulas of x must be identical. Thus the pattern `f(x,x)' would match `f(12, 12)' and `f(a+1, a+1)' but not `f(12, a+1)' or `f(a+b, b+a)'. (See section Conditional Rewrite Rules, for a way to match the latter.)

Things other than variables must match exactly between the pattern and the target formula. To match a particular variable exactly, use the pseudo-function `quote(v)' in the pattern. For example, the pattern `x+quote(y)' matches `x+y', `2+y', or `sin(a)+y'.

The special variable names `e', `pi', `i', `phi', `gamma', `inf', `uinf', and `nan' always match literally. Thus the pattern `sin(d + e + f)' acts exactly like `sin(d + quote(e) + f)'.

If the old pattern is found to match a given formula, that formula is replaced by new, where any occurrences in new of meta-variables from the pattern are replaced with the sub-formulas that they matched. Thus, applying the rule `f(x,y) := g(y+x,x)' to `f(12, a+1)' would produce `g(a+13, 12)'.

The normal a r command applies rewrite rules over and over throughout the target formula until no further changes are possible (up to a limit of 100 times). Use C-u 1 a r to make only one change at a time.

Conditional Rewrite Rules

A rewrite rule can also be conditional, written in the form `old := new :: cond'. (There is also the obsolete form `[old, new, cond]'.) If a cond part is present in the rule, this is an additional condition that must be satisfied before the rule is accepted. Once old has been successfully matched to the target expression, cond is evaluated (with all the meta-variables substituted for the values they matched) and simplified with a s (`calc-simplify`). If the result is a nonzero number or any other object known to be nonzero (see section Declarations), the rule is accepted. If the result is zero or if it is a symbolic formula that is not known to be nonzero, the rule is rejected. See section Logical Operations, for a number of functions that return 1 or 0 according to the results of various tests.

For example, the formula `n > 0' simplifies to 1 or 0 if n is replaced by a positive or nonpositive number, respectively (or if n has been declared to be positive or nonpositive). Thus, the rule `f(x,y) := g(y+x,x) :: x+y > 0' would apply to `f(0, 4)' but not to `f(-3, 2)' or `f(12, a+1)' (assuming no outstanding declarations for a). In the case of `f(-3, 2)', the condition can be shown not to be satisfied; in the case of `f(12, a+1)', the condition merely cannot be shown to be satisfied, but that is enough to reject the rule.

While Calc will use declarations to reason about variables in the formula being rewritten, declarations do not apply to meta-variables. For example, the rule `f(a) := g(a+1)' will match for any values of `a', such as complex numbers, vectors, or formulas, even if `a' has been declared to be real or scalar. If you want the

meta-variable `a' to match only literal real numbers, use `f(a) := g(a+1) :: real(a)'. If you want `a' to match only reals and formulas which are provably real, use `dreal(a)' as the condition.

The `::' operator is a shorthand for the `condition` function; `old := new :: cond' is equivalent to the formula `condition(assign(old, new), cond)'.

If you have several conditions, you can use `... :: c1 :: c2 :: c3' or `... :: c1 && c2 && c3'. The two are entirely equivalent.

It is also possible to embed conditions inside the pattern: `f(x :: x>0, y) := g(y+x, x)'. This is purely a notational convenience, though; where a condition appears in a rule has no effect on when it is tested. The rewrite-rule compiler automatically decides when it is best to test each condition while a rule is being matched.

Certain conditions are handled as special cases by the rewrite rule system and are tested very efficiently: Where x is any meta-variable, these conditions are `integer(x)', `real(x)', `constant(x)', `negative(x)', `x >= y' where y is either a constant or another meta-variable and `>=' may be replaced by any of the six relational operators, and `x % a = b' where a and b are constants. Other conditions, like `x >= y+1' or `dreal(x)', will be less efficient to check since Calc must bring the whole evaluator and simplifier into play.

An interesting property of `::' is that neither of its arguments will be touched by Calc's default simplifications. This is important because conditions often are expressions that cannot safely be evaluated early. For example, the `typeof` function never remains in symbolic form; entering `typeof(a)' will put the number 100 (the type code for variables like `a') on the stack. But putting the condition `... :: typeof(a) = 6' on the stack is safe since `::' prevents the `typeof` from being evaluated until the condition is actually used by the rewrite system.

Since `::' protects its lefthand side, too, you can use a dummy condition to protect a rule that must itself not evaluate early. For example, it's not safe to put `a(f,x) := apply(f, [x])' on the stack because it will immediately evaluate to `a(f,x) := f(x)', where the meta-variable-ness of `f` on the righthand side has been lost. But `a(f,x) := apply(f, [x]) :: 1' is safe, and of course the condition `1' is always true (nonzero) so it has no effect on the functioning of the rule. (The rewrite compiler will ensure that it doesn't even impact the speed of matching the rule.)

Algebraic Properties of Rewrite Rules

The rewrite mechanism understands the algebraic properties of functions like `+' and `*'. In particular, pattern matching takes the associativity and commutativity of the following functions into account:

```
+ - *  = != && ||  and or xor  vint vunion vxor  gcd lcm  max min  beta
```

For example, the rewrite rule:

```
a x + b x  :=  (a + b) x
```

will match formulas of the form,

```
a x + b x,  x a + x b,  a x + x b,  x a + b x
```

Rewrites also understand the relationship between the `+' and `-' operators. The above rewrite rule will match the formulas,

```
a x - b x,  x a - x b,  a x - x b,  x a - b x
```

by matching `b' in the pattern to `-b' from the formula.

Applied to a sum of many terms like `r + a x + s + b x + t', this pattern will check all pairs of terms for possible matches. The rewrite will take whichever suitable pair it discovers first.

In general, a pattern using an associative operator like `a + b' will try *2 n* different ways to match a sum of *n* terms like `x + y + z - w'. First, `a' is matched against each of `x', `y', `z', and `-w' in turn, with `b' being matched to the remainders `y + z - w', `x + z - w', etc. If none of these succeed, then `b' is matched against each of the four terms with `a' matching the remainder. Half-and-half matches, like `(x + y) + (z - w)', are not tried.

Note that `*' is not commutative when applied to matrices, but rewrite rules pretend that it is. If you type m v to enable matrix mode (see section Matrix and Scalar Modes), rewrite rules will match `*' literally, ignoring its usual commutativity property. (In the current implementation, the associativity also vanishes--it is as if the pattern had been enclosed in a `plain` marker; see below.) If you are applying rewrites to formulas with matrices, it's best to enable matrix mode first to prevent algebraically incorrect rewrites from occurring.

The pattern `-x' will actually match any expression. For example, the rule

```
f(-x)  :=  -f(x)
```

will rewrite `f(a)' to `-f(-a)'. To avoid this, either use a `plain` marker as described below, or add a `negative(x)' condition. The `negative` function is true if its argument "looks" negative, for example, because it is a negative number or because it is a formula like `-x'. The new rule using this condition is:

```
f(x)  :=  -f(-x)  :: negative(x)    or, equivalently,
f(-x)  :=  -f(x)  :: negative(-x)
```

In the same way, the pattern `x - y' will match the sum `a + b' by matching `y' to `-b'.

The pattern `a b' will also match the formula `x/y' if `y' is a number. Thus the rule `a x + b x := (a+b) x' will also convert `a x + x / 2' to `(a + 0.5) x' (or `(a + 1:2) x', depending on the current fraction mode).

Calc will *not* take other liberties with `*', `/', and `^'. For example, the pattern `f(a b)' will not match `f(x^2)', and `f(a + b)' will not match `f(2 x)', even though conceivably these patterns could match with `a = b = x'. Nor will `f(a b)' match `f(x / y)' if `y' is not a constant, even though it could be considered to match with `a = x' and `b = 1/y'. The reasons are partly for efficiency, and partly because while few mathematical operations are substantively different for addition and subtraction, often it is preferable to treat the cases of multiplication, division, and integer powers separately.

Even more subtle is the rule set

```
[ f(a) + f(b) := f(a + b),  -f(a) := f(-a) ]
```

attempting to match `f(x) - f(y)'. You might think that Calc will view this subtraction as `f(x) + (-f(y))' and then apply the above two rules in turn, but actually this will not work because Calc only does this when considering rules for `+' (like the first rule in this set). So it will see first that `f(x) + (-f(y))' does not match `f(a) + f(b)' for any assignments of the meta-variables, and then it will see that `f(x) - f(y)' does not match `-f(a)' for any assignment of `a'. Because Calc tries only one rule at a time, it will not be able to rewrite `f(x) - f(y)' with this rule set. An explicit `f(a) - f(b)' rule will have to be added.

Another thing patterns will *not* do is break up complex numbers. The pattern `myconj(a + b i) := a - b i' will work for formulas involving the special constant `i' (such as `3 - 4 i'), but it will not match actual complex numbers like `(3, -4)'. A version of the above rule for complex numbers would be

```
myconj(a)  :=  re(a) - im(a) (0,1)  :: im(a) != 0
```

(Because the `re` and `im` functions understand the properties of the special constant `i', this rule will also work for `3 - 4 i'. In fact, this particular rule would probably be better without the `im(a) != 0' condition, since if `im(a) = 0' the righthand side of the rule will still give the correct answer for the conjugate of a real number.)

It is also possible to specify optional arguments in patterns. The rule

```
opt(a) x + opt(b) (x^opt(c) + opt(d))  :=  f(a, b, c, d)
```

will match the formula

```
5 (x^2 - 4) + 3 x
```

in a fairly straightforward manner, but it will also match reduced formulas like

```
x + x^2,    2(x + 1) - x,    x + x
```

producing, respectively,

```
f(1, 1, 2, 0),   f(-1, 2, 1, 1),   f(1, 1, 1, 0)
```

(The latter two formulas can be entered only if default simplifications have been turned off with m O.)

The default value for a term of a sum is zero. The default value for a part of a product, for a power, or for the denominator of a quotient, is one. Also, `-x' matches the pattern `opt(a) b' with `a = -1'.

In particular, the distributive-law rule can be refined to

```
opt(a) x + opt(b) x  :=  (a + b) x
```

so that it will convert, e.g., `a x - x', to `(a - 1) x'.

The pattern `opt(a) + opt(b) x' matches almost any formulas which are linear in `x'. You can also use the `lin` and `islin` functions with rewrite conditions to test for this; see section Logical Operations. These functions are not as convenient to use in rewrite rules, but they recognize more kinds of formulas as linear: `x/z' is considered linear with b = 1/z by `lin`, but it will not match the above pattern because that pattern

calls for a multiplication, not a division.

As another example, the obvious rule to replace `sin(x)^2 + cos(x)^2' by 1,

```
sin(x)^2 + cos(x)^2  :=  1
```

misses many cases because the sine and cosine may both be multiplied by an equal factor. Here's a more successful rule:

```
opt(a) sin(x)^2 + opt(a) cos(x)^2  :=  a
```

Note that this rule will *not* match `sin(x)^2 + 6 cos(x)^2' because one a would have "matched" 1 while the other matched 6.

Calc automatically converts a rule like

```
f(x-1, x)  :=  g(x)
```

into the form

```
f(temp, x)  :=  g(x)  :: temp = x-1
```

(where `temp` stands for a new, invented meta-variable that doesn't actually have a name). This modified rule will successfully match `f(6, 7)', binding `temp' and `x' to 6 and 7, respectively, then verifying that they differ by one even though `6' does not superficially look like `x-1'.

However, Calc does not solve equations to interpret a rule. The following rule,

```
f(x-1, x+1)  :=  g(x)
```

will not work. That is, it will match `f(a - 1 + b, a + 1 + b)' but not `f(6, 8)'. Calc always interprets at least one occurrence of a variable by literal matching. If the variable appears "isolated" then Calc is smart enough to use it for literal matching. But in this last example, Calc is forced to rewrite the rule to `f(x-1, temp) := g(x) :: temp = x+1' where the `x-1' term must correspond to an actual "something-minus-one" in the target formula.

A successful way to write this would be `f(x, x+2) := g(x+1)'. You could make this resemble the original form more closely by using `let` notation, which is described in the next section:

```
f(xm1, x+1)  :=  g(x)  :: let(x := xm1+1)
```

Calc does this rewriting or "conditionalizing" for any sub-pattern which involves only the functions in the following list, operating only on constants and meta-variables which have already been matched elsewhere in the pattern. When matching a function call, Calc is careful to match arguments which are plain variables before arguments which are calls to any of the functions below, so that a pattern like `f(x-1, x)' can be conditionalized even though the isolated `x' comes after the `x-1'.

```
+ - * / \ % ^  abs sign  round rounde roundu trunc floor ceil
```


```
max min  re im conj arg
```

You can suppress all of the special treatments described in this section by surrounding a function call with a `plain` marker. This marker causes the function call which is its argument to be matched literally, without regard to commutativity, associativity, negation, or conditionalization. When you use `plain`, the "deep structure" of the formula being matched can show through. For example,

```
plain(a - a b)  :=  f(a, b)
```

will match only literal subtractions. However, the `plain` marker does not affect its arguments' arguments. In this case, commutativity and associativity is still considered while matching the `a b' sub-pattern, so the whole pattern will match `x - y x' as well as `x - x y'. We could go still further and use

```
plain(a - plain(a b))  :=  f(a, b)
```

which would do a completely strict match for the pattern.

By contrast, the `quote` marker means that not only the function name but also the arguments must be literally the same. The above pattern will match `x - x y' but

```
quote(a - a b)  :=  f(a, b)
```

will match only the single formula `a - a b'. Also,

```
quote(a - quote(a b))  :=  f(a, b)
```

will match only `a - quote(a b)'---probably not the desired effect!

A certain amount of algebra is also done when substituting the meta-variables on the righthand side of a rule. For example, in the rule

```
a + f(b)  :=  f(a + b)
```

matching `f(x) - y' would produce `f((-y) + x)' if taken literally, but the rewrite mechanism will simplify the righthand side to `f(x - y)' automatically. (Of course, the default simplifications would do this anyway, so this special simplification is only noticeable if you have turned the default simplifications off.) This rewriting is done only when a meta-variable expands to a "negative-looking" expression. If this simplification is not desirable, you can use a `plain` marker on the righthand side:

```
a + f(b)  :=  f(plain(a + b))
```

In this example, we are still allowing the pattern-matcher to use all the algebra it can muster, but the righthand side will always simplify to a literal addition like `f((-y) + x)'.

Other Features of Rewrite Rules

Certain "function names" serve as markers in rewrite rules. Here is a complete list of these markers. First are listed the markers that work inside a pattern; then come the markers that work in the righthand side of a rule.

One kind of marker, `import(x)', takes the place of a whole rule. Here x is the name of a variable containing another rule set; those rules are "spliced into" the rule set that imports them. For example, if `[f(a+b) := f(a) + f(b), f(a b) := a f(b) :: real(a)]' is stored in variable `linearF', then the rule set `[f(0) := 0, import(linearF)]' will apply all three rules. It is possible to modify the imported rules slightly: `import(x, v1, x1, v2, x2, ...)' imports the rule set x with all occurrences of @c{$v_1$} v1, as either a variable name or a function name, replaced with @c{$x_1$} x1 and so on. (If @c{$v_1$} v1 is used as a function name, then @c{$x_1$} x1 must be either a function name itself or a `< >' nameless function; see section Specifying Operators.) For example, `[g(0) := 0, import(linearF, f, g)]' applies the linearity rules to the function `g' instead of `f'. Imports can be nested, but the import-with-renaming feature may fail to rename sub-imports properly.

The special functions allowed in patterns are:

`quote(x)'

> This pattern matches exactly x; variable names in x are not interpreted as meta-variables. The only flexibility is that numbers are compared for numeric equality, so that the pattern `f(quote(12))' will match both `f(12)' and `f(12.0)'. (Numbers are always treated this way by the rewrite mechanism: The rule `f(x,x) := g(x)' will match `f(12, 12.0)'. The rewrite may produce either `g(12)' or `g(12.0)' as a result in this case.)

`plain(x)'

> Here x must be a function call `f(x1,x2,...)'. This pattern matches a call to function f with the specified argument patterns. No special knowledge of the properties of the function f is used in this case; `+' is not commutative or associative. Unlike `quote`, the arguments `x1,x2,...' are treated as patterns. If you wish them to be treated "plainly" as well, you must enclose them with more `plain` markers: `plain(plain(-a) + plain(b c))'.

`opt(x,def)'

> Here x must be a variable name. This must appear as an argument to a function or an element of a vector; it specifies that the argument or element is optional. As an argument to `+', `-', `*', `&&', or `||', or as the second argument to `/' or `^', the value def may be omitted. The pattern `x + opt(y)' matches a sum by binding one summand to x and the other to y, and it matches anything else by binding the whole expression to x and zero to y. The other operators above work similarly.
>
> For general miscellanous functions, the default value `def` must be specified. Optional arguments are dropped starting with the rightmost one during matching. For example, the pattern `f(opt(a,0), b, opt(c,b))' will match `f(b)', `f(a,b)', or `f(a,b,c)'. Default values of zero and b are supplied in this example for the omitted arguments. Note that the literal variable b will be the default in the latter case, *not* the value that matched the meta-variable b. In other words, the default def is effectively quoted.

`condition(x,c)'

> This matches the pattern x, with the attached condition c. It is the same as `x :: c'.

`pand(x,y)'

This matches anything that matches both pattern x and pattern y. It is the same as `x &&& y'. see section Composing Patterns in Rewrite Rules.

`por(x,y)'

This matches anything that matches either pattern x or pattern y. It is the same as `x ||| y'.

`pnot(x)'

This matches anything that does not match pattern x. It is the same as `!!! x'.

`cons(h,t)'

This matches any vector of one or more elements. The first element is matched to h; a vector of the remaining elements is matched to t. Note that vectors of fixed length can also be matched as actual vectors: The rule `cons(a,cons(b,[])) := cons(a+b,[])' is equivalent to the rule `[a,b] := [a+b]'.

`rcons(t,h)'

This is like `cons`, except that the *last* element is matched to h, with the remaining elements matched to t.

`apply(f,args)'

This matches any function call. The name of the function, in the form of a variable, is matched to f. The arguments of the function, as a vector of zero or more objects, are matched to `args'. Constants, variables, and vectors do *not* match an `apply` pattern. For example, `apply(f,x)' matches any function call, `apply(quote(f),x)' matches any call to the function `f', `apply(f,[a,b])' matches any function call with exactly two arguments, and `apply(quote(f), cons(a,cons(b,x)))' matches any call to the function `f' with two or more arguments. Another way to implement the latter, if the rest of the rule does not need to refer to the first two arguments of `f' by name, would be `apply(quote(f), x :: vlen(x) >= 2)'. Here's a more interesting sample use of `apply`:

```
apply(f,[x+n])  :=  n + apply(f,[x])
   :: in(f, [floor,ceil,round,trunc]) :: integer(n)
```

Note, however, that this will be slower to match than a rule set with four separate rules. The reason is that Calc sorts the rules of a rule set according to top-level function name; if the top-level function is `apply`, Calc must try the rule for every single formula and sub-formula. If the top-level function in the pattern is, say, `floor`, then Calc invokes the rule only for sub-formulas which are calls to `floor`.

Formulas normally written with operators like + are still considered function calls: `apply(f,x)` matches `a+b' with `f = add', `x = [a,b]'.

You must use `apply` for meta-variables with function names on both sides of a rewrite rule: `apply(f, [x]) := f(x+1)' is *not* correct, because it rewrites `spam(6)' into `f(7)'. The righthand side should be `apply(f, [x+1])'. Also note that you will have to use no-simplify (m O) mode when entering this rule so that the `apply` isn't evaluated immediately to get the new rule `f(x) := f(x+1)'. Or, use s e to enter the rule without going through the stack, or enter the rule as `apply(f, [x]) := apply(f, [x+1]) :: 1'. See section Conditional Rewrite Rules.

`select(x)'

This is used for applying rules to formulas with selections; see section Selections with Rewrite Rules.

Special functions for the righthand sides of rules are:

`quote(x)'

The notation `quote(x)' is changed to `x' when the righthand side is used. As far as the rewrite rule is concerned, `quote` is invisible. However, `quote` has the special property in Calc that its argument is not evaluated. Thus, while it will not work to put the rule `t(a) := typeof(a)' on the stack because `typeof(a)' is evaluated immediately to produce `t(a) := 100', you can use `quote` to protect the righthand side: `t(a) := quote(typeof(a))'. (See section Conditional Rewrite Rules, for another trick for protecting rules from evaluation.)

`plain(x)'

Special properties of and simplifications for the function call x are not used. One interesting case where `plain` is useful is the rule, `q(x) := quote(x)', trying to expand a shorthand notation for the `quote` function. This rule will not work as shown; instead of replacing `q(foo)' with `quote(foo)', it will replace it with `foo'! The correct rule would be `q(x) := plain(quote(x))'.

`cons(h,t)'

Where t is a vector, this is converted into an expanded vector during rewrite processing. Note that `cons` is a regular Calc function which normally does this anyway; the only way `cons` is treated specially by rewrites is that `cons` on the righthand side of a rule will be evaluated even if default simplifications have been turned off.

`rcons(t,h)'

Analogous to `cons` except putting h at the *end* of the vector t.

`apply(f,args)'

Where f is a variable and args is a vector, this is converted to a function call. Once again, note that `apply` is also a regular Calc function.

`eval(x)'

The formula x is handled in the usual way, then the default simplifications are applied to it even if they have been turned off normally. This allows you to treat any function similarly to the way `cons` and `apply` are always treated. However, there is a slight difference: `cons(2+3, [])' with default simplifications off will be converted to `[2+3]', whereas `eval(cons(2+3, []))' will be converted to `[5]'.

`evalsimp(x)'

The formula x has meta-variables substituted in the usual way, then algebraically simplified as if by the a s command.

`evalextsimp(x)'

The formula x has meta-variables substituted in the normal way, then "extendedly" simplified as if by the a e command.

`select(x)'

See section Selections with Rewrite Rules.

There are also some special functions you can use in conditions.

`let(v := x)'

The expression x is evaluated with meta-variables substituted. The a s command's simplifications are

not applied by default, but x can include calls to `evalsimp` or `evalextsimp` as described above to invoke higher levels of simplification. The result of x is then bound to the meta-variable v. As usual, if this meta-variable has already been matched to something else the two values must be equal; if the meta-variable is new then it is bound to the result of the expression. This variable can then appear in later conditions, and on the righthand side of the rule. In fact, v may be any pattern in which case the result of evaluating x is matched to that pattern, binding any meta-variables that appear in that pattern. Note that `let` can only appear by itself as a condition, or as one term of an `&&' which is a whole condition: It cannot be inside an `||' term or otherwise buried.

The alternate, equivalent form `let(v, x)' is also recognized. Note that the use of `:=' by `let`, while still being assignment-like in character, is unrelated to the use of `:=' in the main part of a rewrite rule.

As an example, `f(a) := g(ia) :: let(ia := 1/a) :: constant(ia)' replaces `f(a)' with `g' of the inverse of `a', if that inverse exists and is constant. For example, if `a' is a singular matrix the operation `1/a' is left unsimplified and `constant(ia)' fails, but if `a' is an invertible matrix then the rule succeeds. Without `let` there would be no way to express this rule that didn't have to invert the matrix twice. Note that, because the meta-variable `ia' is otherwise unbound in this rule, the `let` condition itself always "succeeds" because no matter what `1/a' evaluates to, it can successfully be bound to `ia`.

Here's another example, for integrating cosines of linear terms: `myint(cos(y),x) := sin(y)/b :: let([a,b,x] := lin(y,x))'. The `lin` function returns a 3-vector if its argument is linear, or leaves itself unevaluated if not. But an unevaluated `lin` call will not match the 3-vector on the lefthand side of the `let`, so this `let` both verifies that `y` is linear, and binds the coefficients `a` and `b` for use elsewhere in the rule. (It would have been possible to use `sin(a x + b)/b' for the righthand side instead, but using `sin(y)/b' avoids gratuitous rearrangement of the argument of the sine.)

Similarly, here is a rule that implements an inverse-`erf` function. It uses `root` to search for a solution. If `root` succeeds, it will return a vector of two numbers where the first number is the desired solution. If no solution is found, `root` remains in symbolic form. So we use `let` to check that the result was indeed a vector.

```
ierf(x)  :=  y  :: let([y,z] := root(erf(a) = x, a, .5))
```

`matches(v,p)'

The meta-variable v, which must already have been matched to something elsewhere in the rule, is compared against pattern p. Since `matches` is a standard Calc function, it can appear anywhere in a condition. But if it appears alone or as a term of a top-level `&&', then you get the special extra feature that meta-variables which are bound to things inside p can be used elsewhere in the surrounding rewrite rule.

The only real difference between `let(p := v)' and `matches(v, p)' is that the former evaluates `v' using the default simplifications, while the latter does not.

`remember'

This is actually a variable, not a function. If `remember` appears as a condition in a rule, then when that rule succeeds the original expression and rewritten expression are added to the front of the rule set that contained the rule. If the rule set was not stored in a variable, `remember` is ignored. The lefthand side is enclosed in `quote` in the added rule if it contains any variables.

For example, the rule `f(n) := n f(n-1) :: remember' applied to `f(7)' will add the rule `f(7) := 7 f(6)' to the front of the rule set. The rule set `EvalRules` works slightly differently: There, the evaluation of `f(6)' will complete before the result is added to the rule set, in this case as `f(7) := 5040'. Thus `remember` is most useful inside `EvalRules`.

It is up to you to ensure that the optimization performed by `remember` is safe. For example, the rule `foo(n) := n :: evalv(eatfoo) > 0 :: remember' is a bad idea (`evalv` is the function equivalent of the = command); if the variable `eatfoo` ever contains 1, rules like `foo(7) := 7' will be added to the rule set and will continue to operate even if `eatfoo` is later changed to 0.

`remember(c)'

Remember the match as described above, but only if condition c is true. For example, `remember(n % 4 = 0)' in the above factorial rule remembers only every fourth result. Note that `remember(1)' is equivalent to `remember', and `remember(0)' has no effect.

Composing Patterns in Rewrite Rules

There are three operators, `&&&', `|||', and `!!!', that combine rewrite patterns to make larger patterns. The combinations are "and," "or," and "not," respectively, and these operators are the pattern equivalents of `&&', `||' and `!' (which operate on zero-or-nonzero logical values).

Note that `&&&', `|||', and `!!!' are left in symbolic form by all regular Calc features; they have special meaning only in the context of rewrite rule patterns.

The pattern `p1 &&& p2' matches anything that matches both p1 and p2. One especially useful case is when one of p1 or p2 is a meta-variable. For example, here is a rule that operates on error forms:

```
f(x &&& a +/- b, x)  :=  g(x)
```

This does the same thing, but is arguably simpler than, the rule

```
f(a +/- b, a +/- b)  :=  g(a +/- b)
```

Here's another interesting example:

```
ends(cons(a, x) &&& rcons(y, b))  :=  [a, b]
```

which effectively clips out the middle of a vector leaving just the first and last elements. This rule will change a one-element vector `[a]' to `[a, a]'. The similar rule

```
ends(cons(a, rcons(y, b)))  :=  [a, b]
```

would do the same thing except that it would fail to match a one-element vector.

The pattern `p1 ||| p2' matches anything that matches either p1 or p2. Calc first tries matching against p1; if that fails, it goes on to try p2.

A simple example of `|||' is


```
curve(inf ||| -inf)  :=  0
```

which converts both `curve(inf)' and `curve(-inf)' to zero.

Here is a larger example:

```
log(a, b) ||| (ln(a) :: let(b := e))  :=  mylog(a, b)
```

This matches both generalized and natural logarithms in a single rule. Note that the `::' term must be enclosed in parentheses because that operator has lower precedence than `|||' or `:='.

(In practice this rule would probably include a third alternative, omitted here for brevity, to take care of `log10'.)

While Calc generally treats interior conditions exactly the same as conditions on the outside of a rule, it does guarantee that if all the variables in the condition are special names like `e', or already bound in the pattern to which the condition is attached (say, if `a' had appeared in this condition), then Calc will process this condition right after matching the pattern to the left of the `::'. Thus, we know that `b' will be bound to `e' only if the `ln' branch of the `|||' was taken.

Note that this rule was careful to bind the same set of meta-variables on both sides of the `|||'. Calc does not check this, but if you bind a certain meta-variable only in one branch and then use that meta-variable elsewhere in the rule, results are unpredictable:

```
f(a,b) ||| g(b)  :=  h(a,b)
```

Here if the pattern matches `g(17)', Calc makes no promises about the value that will be substituted for `a' on the righthand side.

The pattern `!!! pat' matches anything that does not match pat. Any meta-variables that are bound while matching pat remain unbound outside of pat.

For example,

```
f(x &&& !!! a +/- b, !!![])  :=  g(x)
```

converts `f' whose first argument is anything *except* an error form, and whose second argument is not the empty vector, into a similar call to `g' (but without the second argument).

If we know that the second argument will be a vector (empty or not), then an equivalent rule would be:

```
f(x, y)  :=  g(x)  :: typeof(x) != 7 :: vlen(y) > 0
```

where of course 7 is the `typeof' code for error forms. Another final condition, that works for any kind of `y', would be `!istrue(y == [])'. (The `istrue' function returns an explicit 0 if its argument was left in symbolic form; plain `!(y == [])' or `y != []' would not work to replace `!!![]' since these would be left unsimplified, and thus cause the rule to fail, if `y' was something like a variable name.)

It is possible for a `!!!' to refer to meta-variables bound elsewhere in the pattern. For example,


```
f(a, !!!a)  :=  g(a)
```

matches any call to `f` with different arguments, changing this to `g` with only the first argument.

If a function call is to be matched and one of the argument patterns contains a `!!!' somewhere inside it, that argument will be matched last. Thus

```
f(!!!a, a)  :=  g(a)
```

will be careful to bind `a' to the second argument of `f` before testing the first argument. If Calc had tried to match the first argument of `f` first, the results would have been disasterous: Since `a` was unbound so far, the pattern `a' would have matched anything at all, and the pattern `!!!a' therefore would *not* have matched anything at all!

Nested Formulas with Rewrite Rules

When a r (`calc-rewrite`) is used, it takes an expression from the top of the stack and attempts to match any of the specified rules to any part of the expression, starting with the whole expression and then, if that fails, trying deeper and deeper sub-expressions. For each part of the expression, the rules are tried in the order they appear in the rules vector. The first rule to match the first sub-expression wins; it replaces the matched sub-expression according to the new part of the rule.

Often, the rule set will match and change the formula several times. The top-level formula is first matched and substituted repeatedly until it no longer matches the pattern; then, sub-formulas are tried, and so on. Once every part of the formula has gotten its chance, the rewrite mechanism starts over again with the top-level formula (in case a substitution of one of its arguments has caused it again to match). This continues until no further matches can be made anywhere in the formula.

It is possible for a rule set to get into an infinite loop. The most obvious case, replacing a formula with itself, is not a problem because a rule is not considered to "succeed" unless the righthand side actually comes out to something different than the original formula or sub-formula that was matched. But if you accidentally had both `ln(a b) := ln(a) + ln(b)' and the reverse `ln(a) + ln(b) := ln(a b)' in your rule set, Calc would run forever switching a formula back and forth between the two forms.

To avoid disaster, Calc normally stops after 100 changes have been made to the formula. This will be enough for most multiple rewrites, but it will keep an endless loop of rewrites from locking up the computer forever. (On most systems, you can also type C-g to halt any Emacs command prematurely.)

To change this limit, give a positive numeric prefix argument. In particular, M-1 a r applies only one rewrite at a time, useful when you are first testing your rule (or just if repeated rewriting is not what is called for by your application).

You can also put a "function call" `iterations(n)' in place of a rule anywhere in your rules vector (but usually at the top). Then, n will be used instead of 100 as the default number of iterations for this rule set. You can use `iterations(inf)' if you want no iteration limit by default. A prefix argument will override the `iterations` limit in the rule set.

```
[ iterations(1),
  f(x) := f(x+1) ]
```

More precisely, the limit controls the number of "iterations," where each iteration is a successful match of a rule pattern whose righthand side, after substituting meta-variables and applying the default simplifications, is different from the original sub-formula that was matched.

A prefix argument of zero sets the limit to infinity. Use with caution!

Given a negative numeric prefix argument, a r will match and substitute the top-level expression up to that many times, but will not attempt to match the rules to any sub-expressions.

In a formula, `rewrite(expr, rules, n)` does a rewriting operation. Here expr is the expression being rewritten, rules is the rule, vector of rules, or variable containing the rules, and n is the optional iteration limit, which may be a positive integer, a negative integer, or `inf' or `-inf'. If n is omitted the `iterations` value from the rule set is used; if both are omitted, 100 is used.

Multi-Phase Rewrite Rules

It is possible to separate a rewrite rule set into several phases. During each phase, certain rules will be enabled while certain others will be disabled. A phase schedule controls the order in which phases occur during the rewriting process.

If a call to the marker function `phase` appears in the rules vector in place of a rule, all rules following that point will be members of the phase(s) identified in the arguments to `phase`. Phases are given integer numbers. The markers `phase()' and `phase(all)' both mean the following rules belong to all phases; this is the default at the start of the rule set.

If you do not explicitly schedule the phases, Calc sorts all phase numbers that appear in the rule set and executes the phases in ascending order. For example, the rule set

```
[ f0(x) := g0(x),
  phase(1),
  f1(x) := g1(x),
  phase(2),
  f2(x) := g2(x),
  phase(3),
  f3(x) := g3(x),
  phase(1,2),
  f4(x) := g4(x) ]
```

has three phases, 1 through 3. Phase 1 consists of the `f0`, `f1`, and `f4` rules (in that order). Phase 2 consists of `f0`, `f2`, and `f4`. Phase 3 consists of `f0` and `f3`.

When Calc rewrites a formula using this rule set, it first rewrites the formula using only the phase 1 rules until no further changes are possible. Then it switches to the phase 2 rule set and continues until no further changes occur, then finally rewrites with phase 3. When no more phase 3 rules apply, rewriting finishes. (This is assuming a r with a large enough prefix argument to allow the rewriting to run to completion; the sequence just described stops early if the number of iterations specified in the prefix argument, 100 by default, is reached.)

During each phase, Calc descends through the nested levels of the formula as described previously. (See

section Nested Formulas with Rewrite Rules.) Rewriting starts at the top of the formula, then works its down to the parts, then goes back to the top and works down again. The phase 2 rules do not begin until its phase 1 rules apply anywhere in the formula.

A `schedule` marker appearing in the rule set (anywhere, but conventionally at the top) changes the default schedule of phases. In the simplest case, `schedule` has a sequence of phase numbers for arguments; each phase number is invoked in turn until the arguments to `schedule` are exhausted. Thus adding `schedule(3,2,1)' at the top of the above rule set would reverse the order of the phases; `schedule(1,2,3)' would have no effect since this is the default schedule; and `schedule(1,2,1,3)' would give phase 1 a second chance after phase 2 has completed, before moving on to phase 3.

Any argument to `schedule` can instead be a vector of phase numbers (or even of sub-vectors). Then the sub-sequence of phases described by the vector are tried repeatedly until no change occurs in any phase in the sequence. For example, `schedule([1, 2], 3)' tries phase 1, then phase 2, then, if either phase made any changes to the formula, repeats these two phases until they can make no further progress. Finally, it goes on to phase 3 for finishing touches.

Also, items in `schedule` can be variable names as well as numbers. A variable name is interpreted as the name of a function to call on the whole formula. For example, `schedule(1, simplify)' says to apply the phase-1 rules (presumably, all of them), then to call `simplify` which is the function name equivalent of a s. Likewise, `schedule([1, simplify])' says to alternate between phase 1 and a s until no further changes occur.

Phases can be used purely to improve efficiency; if it is known that a certain group of rules will apply only at the beginning of rewriting, and a certain other group will apply only at the end, then rewriting will be faster if these groups are identified as separate phases. Once the phase 1 rules are done, Calc can put them aside and no longer spend any time on them while it works on phase 2.

There are also some problems that can only be solved with several rewrite phases. For a real-world example of a multi-phase rule set, examine the set `FitRules`, which is used by the curve-fitting command to convert a model expression to linear form. See section Curve Fitting Details. This set is divided into four phases. The first phase rewrites certain kinds of expressions to be more easily linearizable, but less computationally efficient. After the linear components have been picked out, the final phase includes the opposite rewrites to put each component back into an efficient form. If both sets of rules were included in one big phase, Calc could get into an infinite loop going back and forth between the two forms.

Elsewhere in `FitRules`, the components are first isolated, then recombined where possible to reduce the complexity of the linear fit, then finally packaged one component at a time into vectors. If the packaging rules were allowed to begin before the recombining rules were finished, some components might be put away into vectors before they had a chance to recombine. By putting these rules in two separate phases, this problem is neatly avoided.

Selections with Rewrite Rules

If a sub-formula of the current formula is selected (as by j s; see section Selecting Sub-Formulas), the a r (`calc-rewrite`) command applies only to that sub-formula. Together with a negative prefix argument, you can use this fact to apply a rewrite to one specific part of a formula without affecting any other parts.

The j r (`calc-rewrite-selection`) command allows more sophisticated operations on selections.

This command prompts for the rules in the same way as a r, but it then applies those rules to the whole formula in question even though a sub-formula of it has been selected. However, the selected sub-formula will first have been surrounded by a `select( )' function call. (Calc's evaluator does not understand the function name `select'; this is only a tag used by the j r command.)

For example, suppose the formula on the stack is `2 (a + b)^2' and the sub-formula `a + b' is selected. This formula will be rewritten to `2 select(a + b)^2' and then the rewrite rules will be applied in the usual way. The rewrite rules can include references to `select' to tell where in the pattern the selected sub-formula should appear.

If there is still exactly one `select( )' function call in the formula after rewriting is done, it indicates which part of the formula should be selected afterwards. Otherwise, the formula will be unselected.

You can make j r act much like a r by enclosing both parts of the rewrite rule with `select()'. However, j r allows you to use the current selection in more flexible ways. Suppose you wished to make a rule which removed the exponent from the selected term; the rule `select(a)^x := select(a)' would work. In the above example, it would rewrite `2 select(a + b)^2' to `2 select(a + b)'. This would then be returned to the stack as `2 (a + b)' with the `a + b' selected.

The j r command uses one iteration by default, unlike a r which defaults to 100 iterations. A numeric prefix argument affects j r in the same way as a r. See section Nested Formulas with Rewrite Rules.

As with other selection commands, j r operates on the stack entry that contains the cursor. (If the cursor is on the top-of-stack `.' marker, it works as if the cursor were on the formula at stack level 1.)

If you don't specify a set of rules, the rules are taken from the top of the stack, just as with a r. In this case, the cursor must indicate stack entry 2 or above as the formula to be rewritten (otherwise the same formula would be used as both the target and the rewrite rules).

If the indicated formula has no selection, the cursor position within the formula temporarily selects a sub-formula for the purposes of this command. If the cursor is not on any sub-formula (e.g., it is in the line-number area to the left of the formula), the `select( )' markers are ignored by the rewrite mechanism and the rules are allowed to apply anywhere in the formula.

As a special feature, the normal a r command also ignores `select( )' calls in rewrite rules. For example, if you used the above rule `select(a)^x := select(a)' with a r, it would apply the rule as if it were `a^x := a'. Thus, you can write general purpose rules with `select( )' hints inside them so that they will "do the right thing" in both a r and j r, both with and without selections.

Matching Commands

The a m (`calc-match') [`match'] function takes a vector of formulas and a rewrite-rule-style pattern, and produces a vector of all formulas which match the pattern. The command prompts you to enter the pattern; as for a r, you can enter a single pattern (i.e., a formula with meta-variables), or a vector of patterns, or a variable which contains patterns, or you can give a blank response in which case the patterns are taken from the top of the stack. The pattern set will be compiled once and saved if it is stored in a variable. If there are several patterns in the set, vector elements are kept if they match any of the patterns.

For example, `match(a+b, [x, x+y, x-y, 7, x+y+z])' will return `[x+y, x-y, x+y+z]'.

The `import` mechanism is not available for pattern sets.

The a m command can also be used to extract all vector elements which satisfy any condition: The pattern `x :: x>0' will select all the positive vector elements.

With the Inverse flag [`matchnot`], this command extracts all vector elements which do *not* match the given pattern.

There is also a function `matches(x, p)' which evaluates to 1 if expression x matches pattern p, or to 0 otherwise. This is sometimes useful for including into the conditional clauses of other rewrite rules.

The function `vmatches` is just like `matches`, except that if the match succeeds it returns a vector of assignments to the meta-variables instead of the number 1. For example, `vmatches(f(1,2), f(a,b))' returns `[a := 1, b := 2]'. If the match fails, the function returns the number 0.

Automatic Rewrites

It is possible to get Calc to apply a set of rewrite rules on all results, effectively adding to the built-in set of default simplifications. To do this, simply store your rule set in the variable `EvalRules`. There is a convenient s E command for editing `EvalRules`; see section Other Operations on Variables.

For example, suppose you want `sin(a + b)' to be expanded out to `sin(b) cos(a) + cos(b) sin(a)' wherever it appears, and similarly for `cos(a + b)'. The corresponding rewrite rule set would be,

```
[ sin(a + b)  :=  cos(a) sin(b) + sin(a) cos(b),
  cos(a + b)  :=  cos(a) cos(b) - sin(a) sin(b) ]
```

To apply these manually, you could put them in a variable called `trigexp` and then use a r trigexp every time you wanted to expand trig functions. But if instead you store them in the variable `EvalRules`, they will automatically be applied to all sines and cosines of sums. Then, with `2 x' and `45' on the stack, typing + S will (assuming degrees mode) result in `0.7071 sin(2 x) + 0.7071 cos(2 x)' automatically.

As each level of a formula is evaluated, the rules from `EvalRules` are applied before the default simplifications. Rewriting continues until no further `EvalRules` apply. Note that this is different from the usual order of application of rewrite rules: `EvalRules` works from the bottom up, simplifying the arguments to a function before the function itself, while a r applies rules from the top down.

Because the `EvalRules` are tried first, you can use them to override the normal behavior of any built-in Calc function.

It is important not to write a rule that will get into an infinite loop. For example, the rule set `[f(0) := 1, f(n) := n f(n-1)]' appears to be a good definition of a factorial function, but it is unsafe. Imagine what happens if `f(2.5)' is simplified. Calc will continue to subtract 1 from this argument forever without reaching zero. A safer second rule would be `f(n) := n f(n-1) :: n>0'. Another dangerous rule is `g(x, y) := g(y, x)'. Rewriting `g(2, 4)', this would bounce back and forth between that and `g(4, 2)' forever. If an infinite loop in `EvalRules` occurs, Emacs will eventually stop with a "Computation got stuck or ran too long" message.

Another subtle difference between `EvalRules` and regular rewrites concerns rules that rewrite a formula into an identical formula. For example, `f(n) := f(floor(n))' "fails to match" when n is already an integer. But in `EvalRules` this case is detected only if the righthand side literally becomes the original formula

before any further simplification. This means that `f(n) := f(floor(n))' will get into an infinite loop if it occurs in `EvalRules'. Calc will replace `f(6)' with `f(floor(6))', which is different from `f(6)', so it will consider the rule to have matched and will continue simplifying that formula; first the argument is simplified to get `f(6)', then the rule matches again to get `f(floor(6))' again, ad infinitum. A much safer rule would check its argument first, say, with `f(n) := f(floor(n)) :: !dint(n)'.

(What really happens is that the rewrite mechanism substitutes the meta-variables in the righthand side of a rule, compares to see if the result is the same as the original formula and fails if so, then uses the default simplifications to simplify the result and compares again (and again fails if the formula has simplified back to its original form). The only special wrinkle for the `EvalRules' is that the same rules will come back into play when the default simplifications are used. What Calc wants to do is build `f(floor(6))', see that this is different from the original formula, simplify to `f(6)', see that this is the same as the original formula, and thus halt the rewriting. But while simplifying, `f(6)' will again trigger the same `EvalRules' rule and Calc will get into a loop inside the rewrite mechanism itself.)

The `phase', `schedule', and `iterations' markers do not work in `EvalRules'. If the rule set is divided into phases, only the phase 1 rules are applied, and the schedule is ignored. The rules are always repeated as many times as possible.

The `EvalRules' are applied to all function calls in a formula, but not to numbers (and other number-like objects like error forms), nor to vectors or individual variable names. (Though they will apply to *components* of vectors and error forms when appropriate.) You might try to make a variable `phihat' which automatically expands to its definition without the need to press = by writing the rule `quote(phihat) := (1-sqrt(5))/2', but unfortunately this rule will not work as part of `EvalRules'.

Finally, another limitation is that Calc sometimes calls its built-in functions directly rather than going through the default simplifications. When it does this, `EvalRules' will not be able to override those functions. For example, when you take the absolute value of the complex number (2, 3), Calc computes `sqrt(2*2 + 3*3)' by calling the multiplication, addition, and square root functions directly rather than applying the default simplifications to this formula. So an `EvalRules' rule that (perversely) rewrites `sqrt(13) := 6' would not apply. (However, if you put Calc into symbolic mode so that `sqrt(13)' will be left in symbolic form by the built-in square root function, your rule will be able to apply. But if the complex number were (3,4), so that `sqrt(25)' must be calculated, then symbolic mode will not help because `sqrt(25)' can be evaluated exactly to 5.)

One subtle restriction that normally only manifests itself with `EvalRules' is that while a given rewrite rule is in the process of being checked, that same rule cannot be recursively applied. Calc effectively removes the rule from its rule set while checking the rule, then puts it back once the match succeeds or fails. (The technical reason for this is that compiled pattern programs are not reentrant.) For example, consider the rule `foo(x) := x :: foo(x/2) > 0' attempting to match `foo(8)'. This rule will be inactive while the condition `foo(4) > 0' is checked, even though it might be an integral part of evaluating that condition. Note that this is not a problem for the more usual recursive type of rule, such as `foo(x) := foo(x/2)', because there the rule has succeeded and been reactivated by the time the righthand side is evaluated.

If `EvalRules' has no stored value (its default state), or if anything but a vector is stored in it, then it is ignored.

Even though Calc's rewrite mechanism is designed to compare rewrite rules to formulas as quickly as possible, storing rules in `EvalRules' may make Calc run substantially slower. This is particularly true of

rules where the top-level call is a commonly used function, or is not fixed. The rule `f(n) := n f(n-1) :: r will only activate the rewrite mechanism for calls to the function `f`, but `lg(n) + lg(m) := lg(n m)' will check every `+' operator. And `apply(f, [a*b]) := apply(f, [a]) + apply(f, [b]) :: in(f, [ln, log10])' may seem more "efficient" than two separate rules for `ln` and `log10`, but actually it is vastly less efficient because rules with `apply` as the top-level pattern must be tested against *every* function call that is simplified.

Suppose you want `sin(a + b)' to be expanded out not all the time, but only when a s is used to simplify the formula. The variable `AlgSimpRules` holds rules for this purpose. The a s command will apply `EvalRules` and `AlgSimpRules` to the formula, as well as all of its built-in simplifications.

Most of the special limitations for `EvalRules` don't apply to `AlgSimpRules`. Calc simply does an a r AlgSimpRules command with an infinite repeat count as the first step of a s. It then applies its own built-in simplifications throughout the formula, and then repeats these two steps (along with applying the default simplifications) until no further changes are possible.

There are also `ExtSimpRules` and `UnitSimpRules` variables that are used by a e and u s, respectively; these commands also apply `EvalRules` and `AlgSimpRules`. The variable `IntegSimpRules` contains simplification rules that are used only during integration by a i.

Debugging Rewrites

If a buffer named `*Trace*' exists, the rewrite mechanism will record some useful information there as it operates. The original formula is written there, as is the result of each successful rewrite, and the final result of the rewriting. All phase changes are also noted.

Calc always appends to `*Trace*'. You must empty this buffer yourself periodically if it is in danger of growing unwieldy.

Note that the rewriting mechanism is substantially slower when the `*Trace*' buffer exists, even if the buffer is not visible on the screen. Once you are done, you will probably want to kill this buffer (with C-x k *Trace* RET). If you leave it in existence and forget about it, all your future rewrite commands will be needlessly slow.

Examples of Rewrite Rules

Returning to the example of substituting the pattern `sin(x)^2 + cos(x)^2' with 1, we saw that the rule `opt(a) sin(x)^2 + opt(a) cos(x)^2 := a' does a good job of finding suitable cases. Another solution would be to use the rule `cos(x)^2 := 1 - sin(x)^2', followed by algebraic simplification if necessary. This rule will be the most effective way to do the job, but at the expense of making some changes that you might not desire.

Another algebraic rewrite rule is `exp(x+y) := exp(x) exp(y)'. To make this work with the j r command so that it can be easily targeted to a particular exponential in a large formula, you might wish to write the rule as `select(exp(x+y)) := select(exp(x) exp(y))'. The `select' markers will be ignored by the regular a r command (see section Selections with Rewrite Rules).

A surprisingly useful rewrite rule is `a/(b-c) := a*(b+c)/(b^2-c^2)'. This will simplify the formula whenever b and/or c can be made simpler by squaring. For example, applying this rule to `2 / (sqrt(2) + 3)' yields `6:7 - 2:7 sqrt(2)' (assuming Symbolic Mode has been enabled to keep the square root from being evaulated to a floating-point approximation). This rule is also useful when working with symbolic complex numbers, e.g.,

`(a + b i) / (c + d i)'.

As another example, we could define our own "triangular numbers" function with the rules `[tri(0) := 0, tri(n) := n + tri(n-1) :: n>0]'. Enter this vector and store it in a variable: s t trirules. Now, given a suitable formula like `tri(5)' on the stack, type `a r trirules' to apply these rules repeatedly. After six applications, a r will stop with 15 on the stack. Once these rules are debugged, it would probably be most useful to add them to `EvalRules` so that Calc will evaluate the new `tri` function automatically. We could then use Z K on the keyboard macro ' tri($) RET to make a command that applies `tri` to the value on the top of the stack. See section Programming.

The following rule set, contributed by @c{Fran\c cois} Francois Pinard, implements quaternions, a generalization of the concept of complex numbers. Quaternions have four components, and are here represented by function calls `quat(w, [x, y, z])' with "real part" w and the three "imaginary" parts collected into a vector. Various arithmetical operations on quaternions are supported. To use these rules, either add them to `EvalRules`, or create a command based on a r for simplifying quaternion formulas. A convenient way to enter quaternions would be a command defined by a keyboard macro containing: ' quat($$$$, [$$$, $$, $]) RET.

```
[ quat(w, x, y, z) := quat(w, [x, y, z]),
  quat(w, [0, 0, 0]) := w,
  abs(quat(w, v)) := hypot(w, v),
  -quat(w, v) := quat(-w, -v),
  r + quat(w, v) := quat(r + w, v) :: real(r),
  r - quat(w, v) := quat(r - w, -v) :: real(r),
  quat(w1, v1) + quat(w2, v2) := quat(w1 + w2, v1 + v2),
  r * quat(w, v) := quat(r * w, r * v) :: real(r),
  plain(quat(w1, v1) * quat(w2, v2))
     := quat(w1 * w2 - v1 * v2, w1 * v2 + w2 * v1 + cross(v1, v2)),
  quat(w1, v1) / r := quat(w1 / r, v1 / r) :: real(r),
  z / quat(w, v) := z * quatinv(quat(w, v)),
  quatinv(quat(w, v)) := quat(w, -v) / (w^2 + v^2),
  quatsqr(quat(w, v)) := quat(w^2 - v^2, 2 * w * v),
  quat(w, v)^k := quatsqr(quat(w, v)^(k / 2))
                  :: integer(k) :: k > 0 :: k % 2 = 0,
  quat(w, v)^k := quatsqr(quat(w, v)^((k - 1) / 2)) * quat(w, v)
                  :: integer(k) :: k > 2,
  quat(w, v)^-k := quatinv(quat(w, v)^k) :: integer(k) :: k > 0 ]
```

Quaternions, like matrices, have non-commutative multiplication. In other words, q1 * q2 = q2 * q1 is not necessarily true if q1 and q2 are `quat` forms. The `quat*quat' rule above uses `plain` to prevent Calc from rearranging the product. It may also be wise to add the line `[quat(), matrix]' to the `Decls` matrix, to ensure that Calc's other algebraic operations will not rearrange a quaternion product. See section Declarations.

These rules also accept a four-argument `quat` form, converting it to the preferred form in the first rule. If you would rather see results in the four-argument form, just append the two items `phase(2), quat(w, [x, y, z]) := quat(w, x, y, z)' to the end of the rule set. (But remember that multi-phase rule sets don't work in

TOC

`EvalRules`.)

Go to the previous, next section.

Go to the previous, next section.

Operating on Units

One special interpretation of algebraic formulas is as numbers with units. For example, the formula `5 m / s^2' can be read "five meters per second squared." The commands in this chapter help you manipulate units expressions in this form. Units-related commands begin with the u prefix key.

Basic Operations on Units

A units expression is a formula which is basically a number multiplied and/or divided by one or more unit names, which may optionally be raised to integer powers. Actually, the value part need not be a number; any product or quotient involving unit names is a units expression. Many of the units commands will also accept any formula, where the command applies to all units expressions which appear in the formula.

A unit name is a variable whose name appears in the unit table, or a variable whose name is a prefix character like `k' (for "kilo") or `u' (for "micro") followed by a name in the unit table. A substantial table of built-in units is provided with Calc; see section Predefined Units. You can also define your own unit names; see section User-Defined Units.

Note that if the value part of a units expression is exactly `1', it will be removed by the Calculator's automatic algebra routines: The formula `1 mm' is "simplified" to `mm'. This is only a display anomaly, however; `mm' will work just fine as a representation of one millimeter.

You may find that Algebraic Mode (see section Algebraic Entry) makes working with units expressions easier. Otherwise, you will have to remember to hit the apostrophe key every time you wish to enter units.

The u s (`calc-simplify-units`) [`usimplify`] command simplifies a units expression. It uses a s (`calc-simplify`) to simplify the expression first as a regular algebraic formula; it then looks for features that can be further simplified by converting one object's units to be compatible with another's. For example, `5 m + 23 mm' will simplify to `5.023 m'. When different but compatible units are added, the righthand term's units are converted to match those of the lefthand term. See section Simplification Modes, for a way to have this done automatically at all times.

Units simplification also handles quotients of two units with the same dimensionality, as in `2 in s/L cm' to `5.08 s/L'; fractional powers of unit expressions, as in `sqrt(9 mm^2)' to `3 mm' and `sqrt(9 acre)' to a quantity in meters; and `floor`, `ceil`, `round`, `rounde`, `roundu`, `trunc`, `float`, `frac`, `abs`, and `clean` applied to units expressions, in which case the operation in question is applied only to the numeric part of the expression. Finally, trigonometric functions of quantities with units of angle are evaluated, regardless of the current angular mode.

The u c (`calc-convert-units`) command converts a units expression to new, compatible units. For example, given the units expression `55 mph', typing u c m/s RET produces `24.5872 m/s'. If the units

you request are inconsistent with the original units, the number will be converted into your units times whatever "remainder" units are left over. For example, converting `55 mph' into acres produces `6.08 acre / m s'. (Recall that multiplication binds more strongly than division in Calc formulas, so the units here are acres per meter-second.) Remainder units are expressed in terms of "fundamental" units like `m' and `s', regardless of the input units.

One special exception is that if you specify a single unit name, and a compatible unit appears somewhere in the units expression, then that compatible unit will be converted to the new unit and the remaining units in the expression will be left alone. For example, given the input `980 cm/s^2', the command u c ms will change the `s' to `ms' to get `9.8e-4 cm/ms^2'. The "remainder unit" `cm' is left alone rather than being changed to the base unit `m'.

You can use explicit unit conversion instead of the u s command to gain more control over the units of the result of an expression. For example, given `5 m + 23 mm', you can type u c m or u c mm to express the result in either meters or millimeters. (For that matter, you could type u c fath to express the result in fathoms, if you preferred!)

In place of a specific set of units, you can also enter one of the units system names `si`, `mks` (equivalent), or `cgs`. For example, u c si RET converts the expression into International System of Units (SI) base units. Also, u c base converts to Calc's base units, which are the same as `si` units except that `base` uses `g' as the fundamental unit of mass whereas `si` uses `kg'.

The u c command also accepts composite units, which are expressed as the sum of several compatible unit names. For example, converting `30.5 in' to units `mi+ft+in' (miles, feet, and inches) produces `2 ft + 6.5 in'. Calc first sorts the unit names into order of decreasing relative size. It then accounts for as much of the input quantity as it can using an integer number times the largest unit, then moves on to the next smaller unit, and so on. Only the smallest unit may have a non-integer amount attached in the result. A few standard unit names exist for common combinations, such as `mfi` for `mi+ft+in', and `tpo` for `ton+lb+oz'. Composite units are expanded as if by a x, so that `(ft+in)/hr' is first converted to `ft/hr+in/hr'.

If the value on the stack does not contain any units, u c will prompt first for the old units which this value should be considered to have, then for the new units. Assuming the old and new units you give are consistent with each other, the result also will not contain any units. For example, u c cm RET in RET converts the number 2 on the stack to 5.08.

The u b (`calc-base-units`) command is shorthand for u c base; it converts the units expression on the top of the stack into `base` units. If u s does not simplify a units expression as far as you would like, try u b.

The u c and u b commands treat temperature units (like `degC' and `K') as relative temperatures. For example, u c converts `10 degC' to `18 degF': A change of 10 degrees Celsius corresponds to a change of 18 degrees Fahrenheit.

The u t (`calc-convert-temperature`) command converts absolute temperatures. The value on the stack must be a simple units expression with units of temperature only. This command would convert `10 degC' to `50 degF', the equivalent temperature on the Fahrenheit scale.

The u r (`calc-remove-units`) command removes units from the formula at the top of the stack. The

u x (`calc-extract-units`) command extracts only the units portion of a formula. These comm essentially replace every term of the formula that does or doesn't (respectively) look like a unit name the constant 1, then resimplify the formula.

The u a (`calc-autorange-units`) command turns on and off a mode in which unit prefixes like `k` ("kilo") are automatically applied to keep the numeric part of a units expression in a reasonable range. This mode affects u s and all units conversion commands except u b. For example, with autoranging on, `12345 Hz' will be simplified to `12.345 kHz'. Autoranging is useful for some kinds of units (like `Hz` and `m`), but is probably undesirable for non-metric units like `ft` and `tbsp`. (Composite units are more appropriate for those; see above.)

Autoranging always applies the prefix to the leftmost unit name. Calc chooses the largest prefix that causes the number to be greater than or equal to 1.0. Thus an increasing sequence of adjusted times would be `1 ms, 10 ms, 100 ms, 1 s, 10 s, 100 s, 1 ks'. Generally the rule of thumb is that the number will be adjusted to be in the interval `[1 .. 1000)', although there are several exceptions to this rule. First, if the unit has a power then this is not possible; `0.1 s^2' simplifies to `100000 ms^2'. Second, the "centi-" prefix is allowed to form `cm` (centimeters), but will not apply to other units. The "deci-," "deka-," and "hecto-" prefixes are never used. Thus the allowable interval is `[1 .. 10)' for millimeters and `[1 .. 100)' for centimeters. Finally, a prefix will not be added to a unit if the resulting name is also the actual name of another unit; `1e-15 t' would normally be considered a "femto-ton," but it is written as `1000 at' (1000 atto-tons) instead because `ft` would be confused with feet.

The Units Table

The u v (`calc-enter-units-table`) command displays the units table in another buffer called `*Units Table*`. Each entry in this table gives the unit name as it would appear in an expression, the definition of the unit in terms of simpler units, and a full name or description of the unit. Fundamental units are defined as themselves; these are the units produced by the u b command. The fundamental units are meters, seconds, grams, kelvins, amperes, candelas, moles, radians, and steradians.

The Units Table buffer also displays the Unit Prefix Table. Note that two prefixes, "kilo" and "hecto," accept either upper- or lower-case prefix letters. `Meg' is also accepted as a synonym for the `M' prefix. Whenever a unit name can be interpreted as either a built-in name or a prefix followed by another built-in name, the former interpretation wins. For example, `2 pt' means two pints, not two pico-tons.

The Units Table buffer, once created, is not rebuilt unless you define new units. To force the buffer to be rebuilt, give any numeric prefix argument to u v.

The u V (`calc-view-units-table`) command is like u v except that the cursor is not moved into the Units Table buffer. You can type u V again to remove the Units Table from the display. To return from the Units Table buffer after a u v, type M-# c again or use the regular Emacs C-x o (`other-window`) command. You can also kill the buffer with C-x k if you wish; the actual units table is safely stored inside the Calculator.

The u g (`calc-get-unit-definition`) command retrieves a unit's defining expression and pushes it onto the Calculator stack. For example, u g in will produce the expression `2.54 cm'. This is the same definition for the unit that would appear in the Units Table buffer. Note that this command works only

for actual unit names; u g km will report that no such unit exists, for example, because `km` is really th unit `m` with a `k` ("kilo") prefix. To see a definition of a unit in terms of base units, it is easier to push unit name on the stack and then reduce it to base units with u b.

The u e (`calc-explain-units`) command displays an English description of the units of the expression on the stack. For example, for the expression `62 km^2 g / s^2 mol K', the description is "Square-Kilometer Gram per (Second-squared Mole Degree-Kelvin)." This command uses the English descriptions that appear in the righthand column of the Units Table.

Predefined Units

Since the exact definitions of many kinds of units have evolved over the years, and since certain countries sometimes have local differences in their definitions, it is a good idea to examine Calc's definition of a unit before depending on its exact value. For example, there are three different units for gallons, corresponding to the US (`gal`), Canadian (`galC`), and British (`galUK`) definitions. Also, note that `oz` is a standard ounce of mass, `ozt` is a Troy ounce, and `ozfl` is a fluid ounce.

The temperature units corresponding to degrees Kelvin and Centigrade (Celsius) are the same in this table, since most units commands treat temperatures as being relative. The `calc-convert-temperature` command has special rules for handling the different absolute magnitudes of the various temperature scales.

The unit of volume "liters" can be referred to by either the lower-case `l` or the upper-case `L`.

The unit `A` stands for Amperes; the name `Ang` is used

The unit `pt` stands for pints; the name `point` stands for a typographical point, defined by `72 point = 1 in'. There is also `tpt`, which stands for a printer's point as defined by the TeX typesetting system: `72.27 tpt = 1 in'.

The unit `e` stands for the elementary (electron) unit of charge; because algebra command could mistake this for the special constant e, Calc provides the alternate unit name `ech` which is preferable to `e`.

The name `g` stands for one gram of mass; there is also `gf`, one gram of force. (Likewise for lb, pounds, and lbf.) Meanwhile, one "g" of acceleration is denoted `ga`.

The unit `ton` is a U.S. ton of `2000 lb', and `t` is a metric ton of `1000 kg'.

The names `s` (or `sec`) and `min` refer to units of time; `arcsec` and `arcmin` are units of angle.

Some "units" are really physical constants; for example, `c` represents the speed of light, and `h` represents Planck's constant. You can use these just like other units: converting `.5 c' to `m/s' expresses one-half the speed of light in meters per second. You can also use this merely as a handy reference; the u g command gets the definition of one of these constants in its normal terms, and u b expresses the definition in base units.

Two units, `pi` and `fsc` (the fine structure constant, approximately *1/137*) are dimensionless. The units simplification commands simply treat these names as equivalent to their corresponding values. However you can, for example, use u c to convert a pure number into multiples of the fine structure constant, or u

b to convert this back into a pure number. (When u c prompts for the "old units," just enter a blank li signify that the value really is unitless.)

User-Defined Units

Calc provides ways to get quick access to your selected "favorite" units, as well as ways to define your own new units.

To select your favorite units, store a vector of unit names or expressions in the Calc variable `Units`. The u 1 through u 9 commands (`calc-quick-units`) provide access to these units. If the value on the top of the stack is a plain number (with no units attached), then u 1 gives it the specified units. (Basically, it multiplies the number by the first item in the `Units` vector.) If the number on the stack *does* have units, then u 1 converts that number to the new units. For example, suppose the vector `[in, ft]' is stored in `Units`. Then 30 u 1 will create the expression `30 in', and u 2 will convert that expression to `2.5 ft'.

The u 0 command accesses the tenth element of `Units`. Only ten quick units may be defined at a time. If the `Units` variable has no stored value (the default), or if its value is not a vector, then the quick-units commands will not function. The s U command is a convenient way to edit the `Units` variable; see section Other Operations on Variables.

The u d (`calc-define-unit`) command records the units expression on the top of the stack as the definition for a new, user-defined unit. For example, putting `16.5 ft' on the stack and typing u d rod defines the new unit `rod' to be equivalent to 16.5 feet. The unit conversion and simplification commands will now treat `rod` just like any other unit of length. You will also be prompted for an optional English description of the unit, which will appear in the Units Table.

The u u (`calc-undefine-unit`) command removes a user-defined unit. It is not possible to remove one of the predefined units, however.

If you define a unit with an existing unit name, your new definition will replace the original definition of that unit. If the unit was a predefined unit, the old definition will not be replaced, only "shadowed." The built-in definition will reappear if you later use u u to remove the shadowing definition.

To create a new fundamental unit, use either 1 or the unit name itself as the defining expression. Otherwise the expression can involve any other units that you like (except for composite units like `mfi'). You can create a new composite unit with a sum of other units as the defining expression. The next unit operation like u c or u v will rebuild the internal unit table incorporating your modifications. Note that erroneous definitions (such as two units defined in terms of each other) will not be detected until the unit table is next rebuilt; u v is a convenient way to force this to happen.

Temperature units are treated specially inside the Calculator; it is not possible to create user-defined temperature units.

The u p (`calc-permanent-units`) command stores the user-defined units in your `.emacs' file, so that the units will still be available in subsequent Emacs sessions. If there was already a set of user-defined units in your `.emacs' file, it is replaced by the new set. (See section General Mode Commands, for a way to tell Calc to use a different file instead of `.emacs'.)

Go to the previous, next section.

Go to the previous, next section.

Storing and Recalling

Calculator variables are really just Lisp variables that contain numbers or formulas in a form that Calc can understand. The commands in this section allow you to manipulate variables conveniently. Commands related to variables use the s prefix key.

Storing Variables

The s s (`calc-store`) command stores the value at the top of the stack into a specified variable. It prompts you to enter the name of the variable. If you press a single digit, the value is stored immediately in one of the "quick" variables `var-q0` through `var-q9`. Or you can enter any variable name. The prefix `var-' is supplied for you; when a name appears in a formula (as in `a+q2') the prefix `var-' is also supplied there, so normally you can simply forget about `var-' everywhere. Its only purpose is to enable you to use Calc variables without fear of accidentally clobbering some variable in another Emacs package. If you really want to store in an arbitrary Lisp variable, just backspace over the `var-'.

The s s command leaves the stored value on the stack. There is also an s t (`calc-store-into`) command, which removes a value from the stack and stores it in a variable.

If the top of stack value is an equation `a = 7' or assignment `a := 7' with a variable on the lefthand side, then Calc will assign that variable with that value by default, i.e., if you type s s RET or s t RET. In this example, the value 7 would be stored in the variable `a'. (If you do type a variable name at the prompt, the top-of-stack value is stored in its entirety, even if it is an equation: `s s b RET' with `a := 7' on the stack stores `a := 7' in `b`.)

In fact, the top of stack value can be a vector of equations or assignments with different variables on their lefthand sides; the default will be to store all the variables with their corresponding righthand sides simultaneously.

It is also possible to type an equation or assignment directly at the prompt for the s s or s t command: s s foo = 7. In this case the expression to the right of the = or := symbol is evaluated as if by the = command, and that value is stored in the variable. No value is taken from the stack; s s and s t are equivalent when used in this way.

The prefix keys s and t may be followed immediately by a digit; s 9 is equivalent to s s 9, and t 9 is equivalent to s t 9. (The t prefix is otherwise used for trail and time/date commands.)

There are also several "arithmetic store" commands. For example, s + removes a value from the stack and adds it to the specified variable. The other arithmetic stores are s -, s *, s /, s ^, and s | (vector concatenation), plus s n and s & which negate or invert the value in a variable, and s [and s] which decrease or increase a variable by one.

All the arithmetic stores accept the Inverse prefix to reverse the order of the operands. If v represents the contents of the variable, and a is the value drawn from the stack, then regular s - assigns @c{$v \coloneq

v - a$} v := v - a, but I s - assigns v := a - v. While I s * might seem pointless, it is useful if matrix multiplication is involved. Actually, all the arithmetic stores use formulas designed to behave usefully both forwards and backwards:

```
s +         v := v + a          v := a + v
s -         v := v - a          v := a - v
s *         v := v * a          v := a * v
s /         v := v / a          v := a / v
s ^         v := v ^ a          v := a ^ v
s |         v := v | a          v := a | v
s n         v := v / (-1)       v := (-1) / v
s &         v := v ^ (-1)       v := (-1) ^ v
s [         v := v - 1          v := 1 - v
s ]         v := v - (-1)       v := (-1) - v
```

In the last four cases, a numeric prefix argument will be used in place of the number one. (For example, M-2 s] increases a variable by 2, and M-2 I s] replaces a variable by minus-two minus the variable.

The first six arithmetic stores can also be typed s t +, s t -, etc. The commands s s +, s s -, and so on are analogous arithmetic stores that don't remove the value a from the stack.

All arithmetic stores report the new value of the variable in the Trail for your information. They signal an error if the variable previously had no stored value. If default simplifications have been turned off, the arithmetic stores temporarily turn them on for numeric arguments only (i.e., they temporarily do an m N command). See section Simplification Modes. Large vectors put in the trail by these commands always use abbreviated (t .) mode.

The s m command is a general way to adjust a variable's value using any Calc function. It is a "mapping" command analogous to V M, V R, etc. See section Reducing and Mapping Vectors, to see how to specify a function for a mapping command. Basically, all you do is type the Calc command key that would invoke that function normally. For example, s m n applies the n key to negate the contents of the variable, so s m n is equivalent to s n. Also, s m Q takes the square root of the value stored in a variable, s m v v uses v v to reverse the vector stored in the variable, and s m H I S takes the hyperbolic arcsine of the variable contents.

If the mapping function takes two or more arguments, the additional arguments are taken from the stack; the old value of the variable is provided as the first argument. Thus s m - with a on the stack computes v - a, just like s -. With the Inverse prefix, the variable's original value becomes the *last* argument instead of the first. Thus I s m - is also equivalent to I s -.

The s x (`calc-store-exchange`) command exchanges the value of a variable with the value on the top of the stack. Naturally, the variable must already have a stored value for this to work.

You can type an equation or assignment at the s x prompt. The command s x a=6 takes no values from the stack; instead, it pushes the old value of `a' on the stack and stores `a = 6'.

Until you store something in them, variables are "void," that is, they contain no value at all. If they appear in an algebraic formula they will be left alone even if you press = (`calc-evaluate`). The s u

(`calc-unstore`) command returns a variable to the void state.

The only variables with predefined values are the "special constants" `pi`, `e`, `i`, `phi`, and `gamma`. You are free to unstore these variables or to store new values into them if you like, although some of the algebraic-manipulation functions may assume these variables represent their standard values. Calc displays a warning if you change the value of one of these variables, or of one of the other special variables `inf`, `uinf`, and `nan` (which are normally void).

Note that `var-pi` doesn't actually have 3.14159265359 stored in it, but rather a special magic value that evaluates to @c{π} pi at the current precision. Likewise `var-e`, `var-i`, and `var-phi` evaluate according to the current precision or polar mode. If you recall a value from `pi` and store it back, this magic property will be lost.

The s c (`calc-copy-variable`) command copies the stored value of one variable to another. It differs from a simple s r followed by an s t in two important ways. First, the value never goes on the stack and thus is never rounded, evaluated, or simplified in any way; it is not even rounded down to the current precision. Second, the "magic" contents of a variable like `var-e` can be copied into another variable with this command, perhaps because you need to unstore `var-e` right now but you wish to put it back when you're done. The s c command is the only way to manipulate these magic values intact.

Recalling Variables

The most straightforward way to extract the stored value from a variable is to use the s r (`calc-recall`) command. This command prompts for a variable name (similarly to `calc-store`), looks up the value of the specified variable, and pushes that value onto the stack. It is an error to try to recall a void variable.

It is also possible to recall the value from a variable by evaluating a formula containing that variable. For example, ' a RET = is the same as s r a RET except that if the variable is void, the former will simply leave the formula `a' on the stack whereas the latter will produce an error message.

The r prefix may be followed by a digit, so that r 9 is equivalent to s r 9. (The r prefix is otherwise unused in the current version of Calc.)

Other Operations on Variables

The s e (`calc-edit-variable`) command edits the stored value of a variable without ever putting that value on the stack or simplifying or evaluating the value. It prompts for the name of the variable to edit. If the variable has no stored value, the editing buffer will start out empty. If the editing buffer is empty when you press M-# M-# to finish, the variable will be made void. See section Editing Stack Entries, for a general description of editing.

The s e command is especially useful for creating and editing rewrite rules which are stored in variables. Sometimes these rules contain formulas which must not be evaluated until the rules are actually used. (For example, they may refer to `deriv(x,y)', where `x` will someday become some expression involving `y`; if you let Calc evaluate the rule while you are defining it, Calc will replace `deriv(x,y)' with 0 because

the formula `x` does not itself refer to `y`.) By contrast, recalling the variable, editing with `, and storing will evaluate the variable's value as a side effect of putting the value on the stack.

There are several special-purpose variable-editing commands that use the s prefix followed by a shifted letter:

s A
: Edit `AlgSimpRules`. See section Algebraic Simplifications.

s D
: Edit `Decls`. See section Declarations.

s E
: Edit `EvalRules`. See section Default Simplifications.

s F
: Edit `FitRules`. See section Curve Fitting.

s G
: Edit `GenCount`. See section Solving Equations.

s H
: Edit `Holidays`. See section Business Days.

s I
: Edit `IntegLimit`. See section Calculus.

s L
: Edit `LineStyles`. See section Graphics.

s P
: Edit `PointStyles`. See section Graphics.

s R
: Edit `PlotRejects`. See section Graphics.

s T
: Edit `TimeZone`. See section Time Zones.

s U
: Edit `Units`. See section User-Defined Units.

s X
: Edit `ExtSimpRules`. See section "Unsafe" Simplifications.

These commands are just versions of s e that use fixed variable names rather than prompting for the variable name.

The s p (`calc-permanent-variable`) command saves a variable's value permanently in your `.emacs' file, so that its value will still be available in future Emacs sessions. You can re-execute s p later on to update the saved value, but the only way to remove a saved variable is to edit your

`.emacs' file by hand. (See section General Mode Commands, for a way to tell Calc to use a diffe file instead of `.emacs'.)

If you do not specify the name of a variable to save (i.e., s p RET), all `var-' variables with defined values are saved except for the special constants `pi', `e', `i', `phi', and `gamma'; the variables `TimeZone' and `PlotRejects'; `FitRules', `DistribRules', and other built-in rewrite rules; and `PlotDatan' variables generated by the graphics commands. (You can still save these variables by explicitly naming them in an s p command.)

The s i (`calc-insert-variables') command writes the values of all `var-' variables into a specified buffer. The variables are written in the form of Lisp `setq' commands which store the values in string form. You can place these commands in your `.emacs' buffer if you wish, though in this case it would be easier to use s p RET. (Note that s i omits the same set of variables as s p RET; the difference is that s i will store the variables in any buffer, and it also stores in a more human-readable format.)

The Let Command

If you have an expression like `a+b^2' on the stack and you wish to compute its value where b=3, you can simply store 3 in b and then press = to reevaluate the formula. This has the side-effect of leaving the stored value of 3 in b for future operations.

The s l (`calc-let') command evaluates a formula under a *temporary* assignment of a variable. It stores the value on the top of the stack into the specified variable, then evaluates the second-to-top stack entry, then restores the original value (or lack of one) in the variable. Thus after ' a+b^2 RET 3 s l b RET, the stack will contain the formula `a + 9'. The subsequent command 5 s l a RET will replace this formula with the number 14. The variables `a' and `b' are not permanently affected in any way by these commands.

The value on the top of the stack may be an equation or assignment, or a vector of equations or assignments, in which case the default will be analogous to the case of s t RET. See section Storing Variables.

Also, you can answer the variable-name prompt with an equation or assignment: s l b=3 RET is the same as storing 3 on the stack and typing s l b RET.

The a b (`calc-substitute') command is another way to substitute a variable with a value in a formula. It does an actual substitution rather than temporarily assigning the variable and evaluating. For example, letting n=2 in `f(n pi)' with a b will produce `f(2 pi)', whereas s l would give `f(6.28)' since the evaluation step will also evaluate `pi'.

The Evaluates-To Operator

The special algebraic symbol `=>' is known as the evaluates-to operator. (It will show up as an `evalto' function call in other language modes like Pascal and TeX.) This is a binary operator, that is, it has a lefthand and a righthand argument, although it can be entered with the righthand argument omitted.

A formula like `a => b' is evaluated by Calc as follows: First, a is not simplified or modified in any w
The previous value of argument b is thrown away; the formula a is then copied and evaluated as if by
= command according to all current modes and stored variable values, and the result is installed as the
new value of b.

For example, suppose you enter the algebraic formula `2 + 3 => 17'. The number 17 is ignored, and the lefthand argument is left in its unevaluated form; the result is the formula `2 + 3 => 5'.

You can enter an `=>' formula either directly using algebraic entry (in which case the righthand side may be omitted since it is going to be replaced right away anyhow), or by using the s = (`calc-evalto`) command, which takes a from the stack and replaces it with `a => b'.

Calc keeps track of all `=>' operators on the stack, and recomputes them whenever anything changes that might affect their values, i.e., a mode setting or variable value. This occurs only if the `=>' operator is at the top level of the formula, or if it is part of a top-level vector. In other words, pushing `2 + (a => 17)' will change the 17 to the actual value of `a' when you enter the formula, but the result will not be dynamically updated when `a' is changed later because the `=>' operator is buried inside a sum. However, a vector of `=>' operators will be recomputed, since it is convenient to push a vector like `[a =>, b =>, c =>]' on the stack to make a concise display of all the variables in your problem. (Another way to do this would be to use `[a, b, c] =>', which provides a slightly different format of display. You can use whichever you find easiest to read.)

The m C (`calc-auto-recompute`) command allows you to turn this automatic recomputation on or off. If you turn recomputation off, you must explicitly recompute an `=>' operator on the stack in one of the usual ways, such as by pressing =. Turning recomputation off temporarily can save a lot of time if you will be changing several modes or variables before you look at the `=>' entries again.

Most commands are not especially useful with `=>' operators as arguments. For example, given `x + 2 => 17', it won't work to type 1 + to get `x + 3 => 18'. If you want to operate on the lefthand side of the `=>' operator on the top of the stack, type j 1 (that's the digit "one") to select the lefthand side, execute your commands, then type j u to unselect.

All current modes apply when an `=>' operator is computed, including the current simplification mode. Recall that the formula `x + y + x' is not handled by Calc's default simplifications, but the a s command will reduce it to the simpler form `y + 2 x'. You can also type m A to enable an algebraic-simplification mode in which the equivalent of a s is used on all of Calc's results. If you enter `x + y + x =>' normally, the result will be `x + y + x => x + y + x'. If you change to algebraic-simplification mode, the result will be `x + y + x => y + 2 x'. However, just pressing a s once will have no effect on `x + y + x => x + y + x', because the righthand side depends only on the lefthand side and the current mode settings, and the lefthand side is not affected by commands like a s.

The "let" command (s l) has an interesting interaction with the `=>' operator. The s l command evaluates the second-to-top stack entry with the top stack entry supplying a temporary value for a given variable. As you might expect, if that stack entry is an `=>' operator its righthand side will temporarily show this value for the variable. In fact, all `=>'s on the stack will be updated if they refer to that variable. But this change is temporary in the sense that the next command that causes Calc to look at those stack entries will make them revert to the old variable value.


```
2:  a => a              2:  a => 17          2:  a => a
1:  a + 1 => a + 1      1:  a + 1 => 18      1:  a + 1 => a + 1
    .                       .                    .

                            17 s l a RET         p 8 RET
```

Here the p 8 command changes the current precision, thus causing the `=>' forms to be recomputed after the influence of the "let" is gone. The d SPC command (`calc-refresh`) is a handy way to force the `=>' operators on the stack to be recomputed without any other side effects.

Embedded Mode also uses `=>' operators. In embedded mode, the lefthand side of an `=>' operator can refer to variables assigned elsewhere in the file by `:=' operators. The assignment operator `a := 17' does not actually do anything by itself. But Embedded Mode recognizes it and marks it as a sort of file-local definition of the variable. You can enter `:=' operators in algebraic mode, or by using the s : (`calc-assign`) [`assign`] command which takes a variable and value from the stack and replaces them with an assignment.

See section TeX Language Mode, for the way `=>' appears in TeX language output. The eqn mode gives similar treatment to `=>'.

Go to the previous, next section.

Go to the previous, next section.

Graphics

The commands for graphing data begin with the g prefix key. Calc uses GNUPLOT 2.0 or 3.0 to do graphics. These commands will only work if GNUPLOT is available on your system. (While GNUPLOT sounds like a relative of GNU Emacs, it is actually completely unrelated. However, it is free software and can be obtained from the Free Software Foundation's machine `prep.ai.mit.edu'.)

If you have GNUPLOT installed on your system but Calc is unable to find it, you may need to set the `calc-gnuplot-name` variable in your `.emacs' file. You may also need to set some Lisp variables to show Calc how to run GNUPLOT on your system; these are described under g D and g O below. If you are using the X window system, Calc will configure GNUPLOT for you automatically. If you have GNUPLOT 3.0 and you are not using X, Calc will configure GNUPLOT to display graphs using simple character graphics that will work on any terminal.

Basic Graphics

The easiest graphics command is g f (`calc-graph-fast`). This command takes two vectors of equal length from the stack. The vector at the top of the stack represents the "y" values of the various data points. The vector in the second-to-top position represents the corresponding "x" values. This command runs GNUPLOT (if it has not already been started by previous graphing commands) and displays the set of data points. The points will be connected by lines, and there will also be some kind of symbol to indicate the points themselves.

The "x" entry may instead be an interval form, in which case suitable "x" values are interpolated between the minimum and maximum values of the interval (whether the interval is open or closed is ignored).

The "x" entry may also be a number, in which case Calc uses the sequence of "x" values x, x+1, x+2, etc. (Generally the number 0 or 1 would be used for x in this case.)

The "y" entry may be any formula instead of a vector. Calc effectively uses N (`calc-eval-num`) to evaluate variables in the formula; the result of this must be a formula in a single (unassigned) variable. The formula is plotted with this variable taking on the various "x" values. Graphs of formulas by default use lines without symbols at the computed data points. Note that if neither "x" nor "y" is a vector, Calc guesses at a reasonable number of data points to use. See the g N command below. (The "x" values must be either a vector or an interval if "y" is a formula.)

If "y" is (or evaluates to) a formula of the form `xy(x, y)' then the result is a parametric plot. The two arguments of the fictitious `xy` function are used as the "x" and "y" coordinates of the curve, respectively. In this case the "x" vector or interval you specified is not directly visible in the graph. For example, if "x" is the interval `[0..360]' and "y" is the formula `xy(sin(t), cos(t))', the resulting graph will be a circle.

Also, "x" and "y" may each be variable names, in which case Calc looks for suitable vectors, intervals, or formulas stored in those variables.

The "x" and "y" values for the data points (as pulled from the vectors, calculated from the formulas, or interpolated from the intervals) should be real numbers (integers, fractions, or floats). If either the "x" value or the "y" value of a given data point is not a real number, that data point will be omitted from the graph. The points on either side of the invalid point will *not* be connected by a line.

See the documentation for g a below for a description of the way numeric prefix arguments affect g f.

If you store an empty vector in the variable `PlotRejects` (i.e., [ ] s t PlotRejects), Calc will append information to this vector for every data point which was rejected because its "x" or "y" values were not real numbers. The result will be a matrix where each row holds the curve number, data point number, "x" value, and "y" value for a rejected data point. See section The Evaluates-To Operator, for a handy way to keep tabs on the current value of `PlotRejects`. See section Other Operations on Variables, for the s R command which is another easy way to examine `PlotRejects`.

To clear the graphics display, type g c (`calc-graph-clear`). If the GNUPLOT output device is an X window, the window will go away. Effects on other kinds of output devices will vary. You don't need to use g c if you don't want to--if you give another g f or g p command later on, it will reuse the existing graphics window if there is one.

Three-Dimensional Graphics

The g F (`calc-graph-fast-3d`) command makes a three-dimensional graph. It works only if you have GNUPLOT 3.0 or later; with GNUPLOT 2.0, you will see a GNUPLOT error message if you try this command.

The g F command takes three values from the stack, called "x", "y", and "z", respectively. As was the case for 2D graphs, there are several options for these values.

In the first case, "x" and "y" are each vectors (not necessarily of the same length); either or both may instead be interval forms. The "z" value must be a matrix with the same number of rows as elements in "x", and the same number of columns as elements in "y". The result is a surface plot where @c{z_{ij}} z_ij is the height of the point at coordinate (x_i, y_j) on the surface. The 3D graph will be displayed from a certain default viewpoint; you can change this viewpoint by adding a `set view' to the `*Gnuplot Commands*' buffer as described later. See the GNUPLOT 3.0 documentation for a description of the `set view' command.

Each point in the matrix will be displayed as a dot in the graph, and these points will be connected by a grid of lines (isolines).

In the second case, "x", "y", and "z" are all vectors of equal length. The resulting graph displays a 3D line instead of a surface, where the coordinates of points along the line are successive triplets of values from the input vectors.

In the third case, "x" and "y" are vectors or interval forms, and "z" is any formula involving two variables (not counting variables with assigned values). These variables are sorted into alphabetical order; the first takes on values from "x" and the second takes on values from "y" to form a matrix of results that are graphed as a 3D surface.

If the "z" formula evaluates to a call to the fictitious function `xyz(x, y, z)', then the result is a "parametric surface." In this case, the axes of the graph are taken from the x and y values in these calls, and the "x" and "y" values from the input vectors or intervals are used only to specify the range of inputs to the formula. For example, plotting `[0..360], [0..180], xyz(sin(x)*sin(y), cos(x)*sin(y), cos(y))' will draw a sphere. (Since the default resolution for 3D plots is 5 steps in each of "x" and "y", this will draw a very crude sphere. You could use the g N command, described below, to increase this resolution, or specify the "x" and "y" values as vectors with more than 5 elements.

It is also possible to have a function in a regular g f plot evaluate to an `xyz` call. Since g f plots a line, not a surface, the result will be a 3D parametric line. For example, `[[0..720], xyz(sin(x), cos(x), x)]' will plot two turns of a helix (a three-dimensional spiral).

As for g f, each of "x", "y", and "z" may instead be variables containing the relevant data.

Managing Curves

The g f command is really shorthand for the following commands: C-u g d g a g p. Likewise, g F is shorthand for C-u g d g A g p. You can gain more control over your graph by using these commands directly.

The g a (`calc-graph-add`) command adds the "curve" represented by the two values on the top of the stack to the current graph. You can have any number of curves in the same graph. When you give the g p command, all the curves will be drawn superimposed on the same axes.

The g a command (and many others that affect the current graph) will cause a special buffer, `*Gnuplot Commands*', to be displayed in another window. This buffer is a template of the commands that will be sent to GNUPLOT when it is time to draw the graph. The first g a command adds a `plot` command to this buffer. Succeeding g a commands add extra curves onto that `plot` command. Other graph-related commands put other GNUPLOT commands into this buffer. In normal usage you never need to work with this buffer directly, but you can if you wish. The only constraint is that there must be only one `plot` command, and it must be the last command in the buffer. If you want to save and later restore a complete graph configuration, you can use regular Emacs commands to save and restore the contents of the `*Gnuplot Commands*' buffer.

If the values on the stack are not variable names, g a will invent variable names for them (of the form `PlotDatan') and store the values in those variables. The "x" and "y" variables are what go into the `plot` command in the template. If you add a curve that uses a certain variable and then later change that variable, you can replot the graph without having to delete and re-add the curve. That's because the variable name, not the vector, interval or formula itself, is what was added by g a.

A numeric prefix argument on g a or g f changes the way stack entries are interpreted as curves. With a positive prefix argument n, the top n stack entries are "y" values for n different curves which share a common "x" value in the n+1st stack entry. (Thus g a with no prefix argument is equivalent to C-u 1 g a.)

A prefix of zero or plain C-u means to take two stack entries, "x" and "y" as usual, but to interpret "y" as a vector of "y" values for several curves that share a common "x".

A negative prefix argument tells Calc to read n vectors from the stack; each vector [x, y] describes an

independent curve. This is the only form of g a that creates several curves at once that don't have common "x" values. (Of course, the range of "x" values covered by all the curves ought to be roughly same if they are to look nice on the same graph.)

For example, to plot @c{$\sin n x$} sin(n x) for integers n from 1 to 5, you could use v x to create a vector of integers (n), then V M ' or V M $ to map `sin(n x)' across this vector. The resulting vector of formulas is suitable for use as the "y" argument to a C-u g a or C-u g f command.

The g A (`calc-graph-add-3d`) command adds a 3D curve to the graph. It is not legal to intermix 2D and 3D curves in a single graph. This command takes three arguments, "x", "y", and "z", from the stack. With a positive prefix n, it takes n+2 arguments (common "x" and "y", plus n separate "z"s). With a zero prefix, it takes three stack entries but the "z" entry is a vector of curve values. With a negative prefix -n, it takes n vectors of the form [x, y, z]. The g A command works by adding a `splot` (surface-plot) command to the `*Gnuplot Commands*' buffer.

(Although g a adds a 2D `plot` command to the `*Gnuplot Commands*' buffer, Calc changes this to `splot` before sending it to GNUPLOT if it notices that the data points are evaluating to `xyz` calls. It will not work to mix 2D and 3D g a curves in a single graph, although Calc does not currently check for this.)

The g d (`calc-graph-delete`) command deletes the most recently added curve from the graph. It has no effect if there are no curves in the graph. With a numeric prefix argument of any kind, it deletes all of the curves from the graph.

The g H (`calc-graph-hide`) command "hides" or "unhides" the most recently added curve. A hidden curve will not appear in the actual plot, but information about it such as its name and line and point styles will be retained.

The g j (`calc-graph-juggle`) command moves the curve at the end of the list (the "most recently added curve") to the front of the list. The next-most-recent curve is thus exposed for g d or similar commands to use. With g j you can work with any curve in the graph even though curve-related commands only affect the last curve in the list.

The g p (`calc-graph-plot`) command uses GNUPLOT to draw the graph described in the `*Gnuplot Commands*' buffer. Any GNUPLOT parameters which are not defined by commands in this buffer are reset to their default values. The variables named in the `plot` command are written to a temporary data file and the variable names are then replaced by the file name in the template. The resulting plotting commands are fed to the GNUPLOT program. See the documentation for the GNUPLOT program for more specific information. All temporary files are removed when Emacs or GNUPLOT exits.

If you give a formula for "y", Calc will remember all the values that it calculates for the formula so that later plots can reuse these values. Calc throws out these saved values when you change any circumstances that may affect the data, such as switching from Degrees to Radians mode, or changing the value of a parameter in the formula. You can force Calc to recompute the data from scratch by giving a negative numeric prefix argument to g p.

Calc uses a fairly rough step size when graphing formulas over intervals. This is to ensure quick response. You can "refine" a plot by giving a positive numeric prefix argument to g p. Calc goes through the data points it has computed and saved from previous plots of the function, and computes and inserts a

new data point midway between each of the existing points. You can refine a plot any number of tim but beware that the amount of calculation involved doubles each time.

Calc does not remember computed values for 3D graphs. This means the numerix prefix argument, if any, to g p is effectively ignored if the current graph is three-dimensional.

The g P (`calc-graph-print`) command is like g p, except that it sends the output to a printer instead of to the screen. More precisely, g p looks for `set terminal' or `set output' commands in the `*Gnuplot Commands*' buffer; lacking these it uses the default settings. However, g P ignores `set terminal' and `set output' commands and uses a different set of default values. All of these values are controlled by the g D and g O commands discussed below. Provided everything is set up properly, g p will plot to the screen unless you have specified otherwise and g P will always plot to the printer.

Graphics Options

The g g (`calc-graph-grid`) command turns the "grid" on and off. It is off by default; tick marks appear only at the edges of the graph. With the grid turned on, dotted lines appear across the graph at each tick mark. Note that this command only changes the setting in `*Gnuplot Commands*'; to see the effects of the change you must give another g p command.

The g b (`calc-graph-border`) command turns the border (the box that surrounds the graph) on and off. It is on by default. This command will only work with GNUPLOT 3.0 and later versions.

The g k (`calc-graph-key`) command turns the "key" on and off. The key is a chart in the corner of the graph that shows the correspondence between curves and line styles. It is off by default, and is only really useful if you have several curves on the same graph.

The g N (`calc-graph-num-points`) command allows you to select the number of data points in the graph. This only affects curves where neither "x" nor "y" is specified as a vector. Enter a blank line to revert to the default value (initially 15). With no prefix argument, this command affects only the current graph. With a positive prefix argument this command changes or, if you enter a blank line, displays the default number of points used for all graphs created by g a that don't specify the resolution explicitly. With a negative prefix argument, this command changes or displays the default value (initially 5) used for 3D graphs created by g A. Note that a 3D setting of 5 means that a total of 5^2 = 25 points will be computed for the surface.

Data values in the graph of a function are normally computed to a precision of five digits, regardless of the current precision at the time. This is usually more than adequate, but there are cases where it will not be. For example, plotting 1 + x with x in the interval `[0 .. 1e-6]' will round all the data points down to 1.0! Putting the command `set precision n' in the `*Gnuplot Commands*' buffer will cause the data to be computed at precision n instead of 5. Since this is such a rare case, there is no keystroke-based command to set the precision.

The g h (`calc-graph-header`) command sets the title for the graph. This will show up centered above the graph. The default title is blank (no title).

The g n (`calc-graph-name`) command sets the title of an individual curve. Like the other curve-manipulating commands, it affects the most recently added curve, i.e., the last curve on the list in

the `*Gnuplot Commands*' buffer. To set the title of the other curves you must first juggle them to t
end of the list with g j, or edit the `*Gnuplot Commands*' buffer by hand. Curve titles appear in the l
if the key is turned off they are not used.

The g t (`calc-graph-title-x`) and g T (`calc-graph-title-y`) commands set the titles on the "x" and "y" axes, respectively. These titles appear next to the tick marks on the left and bottom edges of the graph, respectively. Calc does not have commands to control the tick marks themselves, but you can edit them into the `*Gnuplot Commands*' buffer if you wish. See the GNUPLOT documentation for details.

The g r (`calc-graph-range-x`) and g R (`calc-graph-range-y`) commands set the range of values on the "x" and "y" axes, respectively. You are prompted to enter a suitable range. This should be either a pair of numbers of the form, `min:max', or a blank line to revert to the default behavior of setting the range based on the range of values in the data, or `$' to take the range from the top of the stack. Ranges on the stack can be represented as either interval forms or vectors: `[min .. max]' or `[min, max]'.

The g l (`calc-graph-log-x`) and g L (`calc-graph-log-y`) commands allow you to set either or both of the axes of the graph to be logarithmic instead of linear.

For 3D plots, g C-t, g C-r, and g C-l (those are letters with the Control key held down) are the corresponding commands for the "z" axis.

The g z (`calc-graph-zero-x`) and g Z (`calc-graph-zero-y`) commands control whether a dotted line is drawn to indicate the "x" and/or "y" zero axes. (These are the same dotted lines that would be drawn there anyway if you used g g to turn the "grid" feature on.) Zero-axis lines are on by default, and may be turned off only in GNUPLOT 3.0 and later versions. They are not available for 3D plots.

The g s (`calc-graph-line-style`) command turns the connecting lines on or off for the most recently added curve, and optionally selects the style of lines to be used for that curve. Plain g s simply toggles the lines on and off. With a numeric prefix argument, g s turns lines on and sets a particular line style. Line style numbers start at one and their meanings vary depending on the output device. GNUPLOT guarantees that there will be at least six different line styles available for any device.

The g S (`calc-graph-point-style`) command similarly turns the symbols at the data points on or off, or sets the point style. If you turn both lines and points off, the data points will show as tiny dots.

Another way to specify curve styles is with the `LineStyles` and `PointStyles` variables. These variables initially have no stored values, but if you store a vector of integers in one of these variables, the g a and g f commands will use those style numbers instead of the defaults for new curves that are added to the graph. An entry should be a positive integer for a specific style, or 0 to let the style be chosen automatically, or *-1* to turn off lines or points altogether. If there are more curves than elements in the vector, the last few curves will continue to have the default styles. Of course, you can later use g s and g S to change any of these styles.

For example, '[2 -1 3] RET s t LineStyles causes the first curve to have lines in style number 2, the second curve to have no connecting lines, and the third curve to have lines in style 3. Point styles will still be assigned automatically, but you could store another vector in `PointStyles` to define them, too.

Graphical Devices

The g D (`calc-graph-device`) command sets the device name (or "terminal name" in GNUPLOT lingo) to be used by g p commands on this graph. It does not affect the permanent default device name. If you enter a blank name, the device name reverts to the default. Enter `?' to see a list of supported devices.

With a positive numeric prefix argument, g D instead sets the default device name, used by all plots in the future which do not override it with a plain g D command. If you enter a blank line this command shows you the current default. The special name `default` signifies that Calc should choose `x11` if the X window system is in use (as indicated by the presence of a `DISPLAY` environment variable), or otherwise `dumb` under GNUPLOT 3.0 and later, or `postscript` under GNUPLOT 2.0. This is the initial default value.

The `dumb` device is an interface to "dumb terminals," i.e., terminals with no special graphics facilities. It writes a crude picture of the graph composed of characters like `-` and `|` to a buffer called `*Gnuplot Trail*', which Calc then displays. The graph is made the same size as the Emacs screen, which on most dumb terminals will be @c{80×24} 80x24 characters. The graph is displayed in an Emacs "recursive edit"; type q or M-# M-# to exit the recursive edit and return to Calc. Note that the `dumb` device is present only in GNUPLOT 3.0 and later versions.

The word `dumb` may be followed by two numbers separated by spaces. These are the desired width and height of the graph in characters. Also, the device name `big` is like `dumb` but creates a graph four times the width and height of the Emacs screen. You will then have to scroll around to view the entire graph. In the `*Gnuplot Trail*' buffer, SPC, DEL, <, and > are defined to scroll by one screenful in each of the four directions.

With a negative numeric prefix argument, g D sets or displays the device name used by g P (`calc-graph-print`). This is initially `postscript`. If you don't have a PostScript printer, you may decide once again to use `dumb` to create a plot on any text-only printer.

The g O (`calc-graph-output`) command sets the name of the output file used by GNUPLOT. For some devices, notably `x11`, there is no output file and this information is not used. Many other "devices" are really file formats like `postscript`; in these cases the output in the desired format goes into the file you name with g O. Type g O stdout RET to set GNUPLOT to write to its standard output stream, i.e., to `*Gnuplot Trail*'. This is the default setting.

Another special output name is `tty`, which means that GNUPLOT is going to write graphics commands directly to its standard output, which you wish Emacs to pass through to your terminal. Tektronix graphics terminals, among other devices, operate this way. Calc does this by telling GNUPLOT to write to a temporary file, then running a sub-shell executing the command `cat tempfile >/dev/tty'. On typical Unix systems, this will copy the temporary file directly to the terminal, bypassing Emacs entirely. You will have to type C-l to Emacs afterwards to refresh the screen.

Once again, g O with a positive or negative prefix argument sets the default or printer output file names, respectively. In each case you can specify `auto`, which causes Calc to invent a temporary file name for each g p (or g P) command. This temporary file will be deleted once it has been displayed or printed. If the output file name is not `auto`, the file is not automatically deleted.

The default and printer devices and output files can be saved permanently by the m m (`calc-save-modes`) command. The default number of data points (see g N) and the X geometry (see g X) are also saved. Other graph information is *not* saved; you can save a graph's configuration simply by saving the contents of the `*Gnuplot Commands*' buffer.

If you are installing Calc you may wish to configure the default and printer devices and output files for the whole system. The relevant Lisp variables are `calc-gnuplot-default-device` and `-output`, and `calc-gnuplot-print-device` and `-output`. The output file names must be either strings as described above, or Lisp expressions which are evaluated on the fly to get the output file names.

Other important Lisp variables are `calc-gnuplot-plot-command` and `calc-gnuplot-print-command`, which give the system commands to display or print the output of GNUPLOT, respectively. These may be `nil` if no command is necessary, or strings which can include `%s' to signify the name of the file to be displayed or printed. Or, these variables may contain Lisp expressions which are evaluated to display or print the output.

The g x (`calc-graph-display`) command lets you specify on which X window system display your graphs should be drawn. Enter a blank line to see the current display name. This command has no effect unless the current device is `x11`.

The g X (`calc-graph-geometry`) command is a similar command for specifying the position and size of the X window. The normal value is `default`, which generally means your window manager will let you place the window interactively. Entering `800x500+0+0' would create an 800-by-500 pixel window in the upper-left corner of the screen.

The buffer called `*Gnuplot Trail*' holds a transcript of the session with GNUPLOT. This shows the commands Calc has "typed" to GNUPLOT and the responses it has received. Calc tries to notice when an error message has appeared here and display the buffer for you when this happens. You can check this buffer yourself if you suspect something has gone wrong.

The g C (`calc-graph-command`) command prompts you to enter any line of text, then simply sends that line to the current GNUPLOT process. The `*Gnuplot Trail*' buffer looks deceptively like a Shell buffer but you can't type commands in it yourself. Instead, you must use g C for this purpose.

The g v (`calc-graph-view-commands`) and g V (`calc-graph-view-trail`) commands display the `*Gnuplot Commands*' and `*Gnuplot Trail*' buffers, respectively, in another window. This happens automatically when Calc thinks there is something you will want to see in either of these buffers. If you type g v or g V when the relevant buffer is already displayed, the buffer is hidden again.

One reason to use g v is to add your own commands to the `*Gnuplot Commands*' buffer. Press g v, then use C-x o to switch into that window. For example, GNUPLOT has `set label' and `set arrow' commands that allow you to annotate your plots. Since Calc doesn't understand these commands, you have to add them to the `*Gnuplot Commands*' buffer yourself, then use g p to replot using these new commands. Note that your commands must appear *before* the `plot` command. To get help on any GNUPLOT feature, type, e.g., g C help set label. You may have to type g C RET a few times to clear the "press return for more" or "subtopic of ..." requests. Note that Calc always sends commands (like `set nolabel') to reset all plotting parameters to the defaults before each plot, so to delete a label all you need

to do is delete the `set label' line you added (or comment it out with `#') and then replot with g p.

You can use g q (`calc-graph-quit`) to kill the GNUPLOT process that is running. The next graphing command you give will start a fresh GNUPLOT process. The word `Graph' appears in the Calc window's mode line whenever a GNUPLOT process is currently running. The GNUPLOT process is automatically killed when you exit Emacs if you haven't killed it manually by then.

The g K (`calc-graph-kill`) command is like g q except that it also views the `*Gnuplot Trail*' buffer so that you can see the process being killed. This is better if you are killing GNUPLOT because you think it has gotten stuck.

Go to the previous, next section.

Go to the previous, next section.

Kill and Yank Functions

The commands in this chapter move information between the Calculator and other Emacs editing buffers.

In many cases Embedded Mode is an easier and more natural way to work with Calc from a regular editing buffer. See section Embedded Mode.

Killing from the Stack

Kill commands are Emacs commands that insert text into the "kill ring," from which it can later be "yanked" by a C-y command. Three common kill commands in normal Emacs are C-k, which kills one line, C-w, which kills the region between mark and point, and M-w, which puts the region into the kill ring without actually deleting it. All of these commands work in the Calculator, too. Also, M-k has been provided to complete the set; it puts the current line into the kill ring without deleting anything.

The kill commands are unusual in that they pay attention to the location of the cursor in the Calculator buffer. If the cursor is on or below the bottom line, the kill commands operate on the top of the stack. Otherwise, they operate on whatever stack element the cursor is on. Calc's kill commands always operate on whole stack entries. (They act the same as their standard Emacs cousins except they "round up" the specified region to encompass full lines.) The text is copied into the kill ring exactly as it appears on the screen, including line numbers if they are enabled.

A numeric prefix argument to C-k or M-k affects the number of lines killed. A positive argument kills the current line and n-1 lines below it. A negative argument kills the -n lines above the current line. Again this mirrors the behavior of the standard Emacs C-k command. Although a whole line is always deleted, C-k with no argument copies only the number itself into the kill ring, whereas C-k with a prefix argument of 1 copies the number with its trailing newline.

Yanking into the Stack

The C-y command yanks the most recently killed text back into the Calculator. It pushes this value onto the top of the stack regardless of the cursor position. In general it re-parses the killed text as a number or formula (or a list of these separated by commas or newlines). However if the thing being yanked is something that was just killed from the Calculator itself, its full internal structure is yanked. For example, if you have set the floating-point display mode to show only four significant digits, then killing and re-yanking 3.14159 (which displays as 3.142) will yank the full 3.14159, even though yanking it into any other buffer would yank the number in its displayed form, 3.142. (Since the default display modes show all objects to their full precision, this feature normally makes no difference.)

Grabbing from Other Buffers

The M-# g (`calc-grab-region`) command takes the text between point and mark in the current buffer and attempts to parse it as a vector of values. Basically, it wraps the text in vector brackets `[ ]' unless the text already is enclosed in vector brackets, then reads the text as if it were an algebraic entry. The contents of the vector may be numbers, formulas, or any other Calc objects. If the M-# g command works successfully, it does an automatic M-# c to enter the Calculator buffer.

A numeric prefix argument grabs the specified number of lines around point, ignoring the mark. A positive prefix grabs from point to the nth following newline (so that M-1 M-# g grabs from point to the end of the current line); a negative prefix grabs from point back to the n+1st preceding newline. In these cases the text that is grabbed is exactly the same as the text that C-k would delete given that prefix argument.

A prefix of zero grabs the current line; point may be anywhere on the line.

A plain C-u prefix interprets the region between point and mark as a single number or formula rather than a vector. For example, M-# g on the text `2 a b' produces the vector of three values `[2, a, b]', but C-u M-# g on the same region reads a formula which is a product of three things: `2 a b'. (The text `a + b', on the other hand, will be grabbed as a vector of one element by plain M-# g because the interpretation `[a, +, b]' would be a syntax error.)

If a different language has been specified (see section Language Modes), the grabbed text will be interpreted according to that language.

The M-# r (`calc-grab-rectangle`) command takes the text between point and mark and attempts to parse it as a matrix. If point and mark are both in the leftmost column, the lines in between are parsed in their entirety. Otherwise, point and mark define the corners of a rectangle whose contents are parsed.

Each line of the grabbed area becomes a row of the matrix. The result will actually be a vector of vectors, which Calc will treat as a matrix only if every row contains the same number of values.

If a line contains a portion surrounded by square brackets (or curly braces), that portion is interpreted as a vector which becomes a row of the matrix. Any text surrounding the bracketed portion on the line is ignored.

Otherwise, the entire line is interpreted as a row vector as if it were surrounded by square brackets. Leading line numbers (in the format used in the Calc stack buffer) are ignored. If you wish to force this interpretation (even if the line contains bracketed portions), give a negative numeric prefix argument to the M-# r command.

If you give a numeric prefix argument of zero or plain C-u, each line is instead interpreted as a single formula which is converted into a one-element vector. Thus the result of C-u M-# r will be a one-column matrix. For example, suppose one line of the data is the expression `2 a'. A plain M-# r will interpret this as `[2 a]', which in turn is read as a two-element vector that forms one row of the matrix. But a C-u M-# r will interpret this row as `[2*a]'.

If you give a positive numeric prefix argument n, then each line will be split up into columns of width n;

each column is parsed separately as a matrix element. If a line contained `2 +/- 3 4 +/- 5', then grabbi with a prefix argument of 8 would correctly split the line into two error forms.

See section Vector/Matrix Functions, to see how to pull the matrix apart into its constituent rows and columns. (If it is a @c{1×1} 1x1 matrix, just hit v u (`calc-unpack`) twice.)

The M-# : (`calc-grab-sum-down`) command is a handy way to grab a rectangle of data and sum its columns. It is equivalent to typing M-# r, followed by V R : + (the vector reduction command that sums the columns of a matrix; see section Reducing). The result of the command will be a vector of numbers, one for each column in the input data. The M-# _ (`calc-grab-sum-across`) command similarly grabs a rectangle and sums its rows by executing V R _ +.

As well as being more convenient, M-# : and M-# _ are also much faster because they don't actually place the grabbed vector on the stack. In a M-# r V R : + sequence, formatting the vector for display on the stack takes a large fraction of the total time (unless you have planned ahead and used v . and t . modes).

For example, suppose we have a column of numbers in a file which we wish to sum. Go to one corner of the column and press C-@ to set the mark; go to the other corner and type M-# :. Since there is only one column, the result will be a vector of one number, the sum. (You can type v u to unpack this vector into a plain number if you want to do further arithmetic with it.)

To compute the product of the column of numbers, we would have to do it "by hand" since there's no special grab-and-multiply command. Use M-# r to grab the column of numbers into the calculator in the form of a column matrix. The statistics command u * is a handy way to find the product of a vector or matrix of numbers. See section Statistical Operations on Vectors. Another approach would be to use an explicit column reduction command, V R : *.

Yanking into Other Buffers

The plain y (`calc-copy-to-buffer`) command inserts the number at the top of the stack into the most recently used normal editing buffer. (More specifically, this is the most recently used buffer which is displayed in a window and whose name does not begin with `*'. If there is no such buffer, this is the most recently used buffer except for Calculator and Calc Trail buffers.) The number is inserted exactly as it appears and without a newline. (If line-numbering is enabled, the line number is normally not included.) The number is *not* removed from the stack.

With a prefix argument, y inserts several numbers, one per line. A positive argument inserts the specified number of values from the top of the stack. A negative argument inserts the nth value from the top of the stack. An argument of zero inserts the entire stack. Note that y with an argument of 1 is slightly different from y with no argument; the former always copies full lines, whereas the latter strips off the trailing newline.

With a lone C-u as a prefix argument, y *replaces* the region in the other buffer with the yanked text, then quits the Calculator, leaving you in that buffer. A typical use would be to use M-# g to read a region of data into the Calculator, operate on the data to produce a new matrix, then type C-u y to replace the original data with the new data. One might wish to alter the matrix display style (see section Vector and

Matrix Display Formats) or change the current display language (see section Language Modes) befo doing this. Also, note that this command replaces a linear region of text (as grabbed by M-# g), not a rectangle (as grabbed by M-# r).

If the editing buffer is in overwrite (as opposed to insert) mode, and the C-u prefix was not used, then the yanked number will overwrite the characters following point rather than being inserted before those characters. The usual conventions of overwrite mode are observed; for example, characters will be inserted at the end of a line rather than overflowing onto the next line. Yanking a multi-line object such as a matrix in overwrite mode overwrites the next n lines in the buffer, lengthening or shortening each line as necessary. Finally, if the thing being yanked is a simple integer or floating-point number (like `-1.2345e-3') and the characters following point also make up such a number, then Calc will replace that number with the new number, lengthening or shortening as necessary. The concept of "overwrite mode" has thus been generalized from overwriting characters to overwriting one complete number with another.

The M-# y key sequence is equivalent to y except that it can be typed anywhere, not just in Calc. This provides an easy way to guarantee that Calc knows which editing buffer you want to use!

X Cut and Paste

If you are using Emacs with the X window system, there is an easier way to move small amounts of data into and out of the calculator: Use the mouse-oriented cut and paste facilities of X.

The default bindings for a three-button mouse cause the left button to move the Emacs cursor to the given place, the right button to select the text between the cursor and the clicked location, and the middle button to yank the selection into the buffer at the clicked location. So, if you have a Calc window and an editing window on your Emacs screen, you can use left-click/right-click to select a number, vector, or formula from one window, then middle-click to paste that value into the other window. When you paste text into the Calc window, Calc interprets it as an algebraic entry. It doesn't matter where you click in the Calc window; the new value is always pushed onto the top of the stack.

The `xterm` program that is typically used for general-purpose shell windows in X interprets the mouse buttons in the same way. So you can use the mouse to move data between Calc and any other Unix program. One nice feature of `xterm` is that a double left-click selects one word, and a triple left-click selects a whole line. So you can usually transfer a single number into Calc just by double-clicking on it in the shell, then middle-clicking in the Calc window.

Go to the previous, next section.

Go to the previous, next section.

"Keypad" Mode

The M-# k (`calc-keypad`) command starts the Calculator and displays a picture of a calculator-style keypad. If you are using the X window system, you can click on any of the "keys" in the keypad using the left mouse button to operate the calculator. The original window remains the selected window; in keypad mode you can type in your file while simultaneously performing calculations with the mouse.

If you have used M-# b first, M-# k instead invokes the `full-calc-keypad` command, which takes over the whole Emacs screen and displays the keypad, the Calc stack, and the Calc trail all at once. This mode would normally be used when running Calc standalone (see section Standalone Operation).

If you aren't using the X window system, you must switch into the `*Calc Keypad*' window, place the cursor on the desired "key," and type SPC or RET. If you think this is easier than using Calc normally, go right ahead.

Calc commands are more or less the same in keypad mode. Certain keypad keys differ slightly from the corresponding normal Calc keystrokes; all such deviations are described below.

Keypad Mode includes many more commands than will fit on the keypad at once. Click the right mouse button [`calc-keypad-menu`] to switch to the next menu. The bottom five rows of the keypad stay the same; the top three rows change to a new set of commands. To return to earlier menus, click the middle mouse button [`calc-keypad-menu-back`] or simply advance through the menus until you wrap around. Typing TAB inside the keypad window is equivalent to clicking the right mouse button there.

You can always click the EXEC button and type any normal Calc key sequence. This is equivalent to switching into the Calc buffer, typing the keys, then switching back to your original buffer.

Main Menu

```
|----+-----Calc 2.00-----+----1
|FLR |CEIL|RND |TRNC|CLN2|FLT |
|----+----+----+----+----+----|
| LN |EXP |    |ABS |IDIV|MOD |
|----+----+----+----+----+----|
|SIN |COS |TAN |SQRT|y^x |1/x |
|----+----+----+----+----+----|
|  ENTER  |+/- |EEX |UNDO| <- |
|-----+---+-+--+--+-+---++----|
| INV |  7  |  8  |  9  |  /  |
|-----+-----+-----+-----+-----|
| HYP |  4  |  5  |  6  |  *  |
|-----+-----+-----+-----+-----|
```


```
|EXEC |  1  |  2  |  3  |  -  |
|-----+-----+-----+-----+-----|
| OFF |  0  |  .  | PI  |  +  |
|-----+-----+-----+-----+-----+
```

This is the menu that appears the first time you start Keypad Mode. It will show up in a vertical window on the right side of your screen. Above this menu is the traditional Calc stack display. On a 24-line screen you will be able to see the top three stack entries.

The ten digit keys, decimal point, and EEX key are used for entering numbers in the obvious way. EEX begins entry of an exponent in scientific notation. Just as with regular Calc, the number is pushed onto the stack as soon as you press ENTER or any other function key.

The +/- key corresponds to normal Calc's n key. During numeric entry it changes the sign of the number or of the exponent. At other times it changes the sign of the number on the top of the stack.

The INV and HYP keys modify other keys. As well as having the effects described elsewhere in this manual, Keypad Mode defines several other "inverse" operations. These are described below and in the following sections.

The ENTER key finishes the current numeric entry, or otherwise duplicates the top entry on the stack.

The UNDO key undoes the most recent Calc operation. INV UNDO is the "redo" command, and HYP UNDO is "last arguments" (M-RET).

The <- key acts as a "backspace" during numeric entry. At other times it removes the top stack entry. INV <- clears the entire stack. HYP <- takes an integer from the stack, then removes that many additional stack elements.

The EXEC key prompts you to enter any keystroke sequence that would normally work in Calc mode. This can include a numeric prefix if you wish. It is also possible simply to switch into the Calc window and type commands in it; there is nothing "magic" about this window when Keypad Mode is active.

The other keys in this display perform their obvious calculator functions. CLN2 rounds the top-of-stack by temporarily reducing the precision by 2 digits. FLT converts an integer or fraction on the top of the stack to floating-point.

The INV and HYP keys combined with several of these keys give you access to some common functions even if the appropriate menu is not displayed. Obviously you don't need to learn these keys unless you find yourself wasting time switching among the menus.

INV +/-

is the same as 1/x.

INV +

is the same as SQRT.

INV -

is the same as CONJ.

INV *

is the same as y^x.

INV /

is the same as INV y^x (the xth root of y).

HYP/INV 1

are the same as SIN / INV SIN.

HYP/INV 2

are the same as COS / INV COS.

HYP/INV 3

are the same as TAN / INV TAN.

INV/HYP 4

are the same as LN / HYP LN.

INV/HYP 5

are the same as EXP / HYP EXP.

INV 6

is the same as ABS.

INV 7

is the same as RND (`calc-round`).

INV 8

is the same as CLN2.

INV 9

is the same as FLT (`calc-float`).

INV 0

is the same as IMAG.

INV .

is the same as PREC.

INV ENTER

is the same as SWAP.

HYP ENTER

is the same as RLL3.

INV HYP ENTER

is the same as OVER.

HYP +/-

packs the top two stack entries as an error form.

HYP EEX

packs the top two stack entries as a modulo form.

INV EEX

creates an interval form; this removes an integer which is one of 0 `[]', 1 `[)', 2 `(]' or 3 `()', followed by the two limits of the interval.

The OFF key turns Calc off; typing M-# k or M-# M-# again has the same effect. This is analogous to typing q or hitting M-# c again in the normal calculator. If Calc is running standalone (the `full-calc-keypad` command appeared in the command line that started Emacs), then OFF is replaced with EXIT; clicking on this actually exits Emacs itself.

Functions Menu

```
|----+----+----+----+----+----2
|IGAM|BETA|IBET|ERF |BESJ|BESY|
|----+----+----+----+----+----|
|IMAG|CONJ| RE |ATN2|RAND|RAGN|
|----+----+----+----+----+----|
|GCD |FACT|DFCT|BNOM|PERM|NXTP|
|----+----+----+----+----+----|
```

This menu provides various operations from the f and k prefix keys.

IMAG multiplies the number on the stack by the imaginary number i = (0, 1).

RE extracts the real part a complex number. INV RE extracts the imaginary part.

RAND takes a number from the top of the stack and computes a random number greater than or equal to zero but less than that number. (See section Random Numbers.) RAGN is the "random again" command; it computes another random number using the same limit as last time.

INV GCD computes the LCM (least common multiple) function.

INV FACT is the gamma function. @c{$\Gamma(x) = (x-1)!$} gamma(x) = (x-1)!.

PERM is the number-of-permutations function, which is on the H k c key in normal Calc.

NXTP finds the next prime after a number. INV NXTP finds the previous prime.

Binary Menu

```
|----+----+----+----+----+----3
|AND | OR |XOR |NOT |LSH |RSH |
|----+----+----+----+----+----|
|DEC |HEX |OCT |BIN |WSIZ|ARSH|
|----+----+----+----+----+----|
| A  | B  | C  | D  | E  | F  |
|----+----+----+----+----+----|
```

The keys in this menu perform operations on binary integers. Note that both logical and arithmetic right-shifts are provided. INV LSH rotates one bit to the left.

The "difference" function (normally on b d) is on INV AND. The "clip" function (normally on b c) is on INV NOT.

The DEC, HEX, OCT, and BIN keys select the current radix for display and entry of numbers: Decimal, hexadecimal, octal, or binary. The six letter keys A through F are used for entering hexadecimal numbers.

The WSIZ key displays the current word size for binary operations and allows you to enter a new word size. You can respond to the prompt using either the keyboard or the digits and ENTER from the keypad. The initial word size is 32 bits.

Vectors Menu

```
|----+----+----+----+----+----4
|SUM |PROD|MAX |MAP*|MAP^|MAP$|
|----+----+----+----+----+----|
|MINV|MDET|MTRN|IDNT|CRSS|"x" |
|----+----+----+----+----+----|
|PACK|UNPK|INDX|BLD |LEN |... |
|----+----+----+----+----+----|
```

The keys in this menu operate on vectors and matrices.

PACK removes an integer n from the top of the stack; the next n stack elements are removed and packed into a vector, which is replaced onto the stack. Thus the sequence 1 ENTER 3 ENTER 5 ENTER 3 PACK enters the vector `[1, 3, 5]' onto the stack. To enter a matrix, build each row on the stack as a vector, then use a final PACK to collect the rows into a matrix.

UNPK unpacks the vector on the stack, pushing each of its components separately.

INDX removes an integer n, then builds a vector of integers from 1 to n. INV INDX takes three numbers from the stack: The vector size n, the starting number, and the increment. BLD takes an integer n and any value x and builds a vector of n copies of x.

IDNT removes an integer n, then builds an n-by-n identity matrix.

LEN replaces a vector by its length, an integer.

... turns on or off "abbreviated" display mode for large vectors.

MINV, MDET, MTRN, and CROSS are the matrix inverse, determinant, and transpose, and vector cross product.

SUM replaces a vector by the sum of its elements. It is equivalent to u + in normal Calc (see section Statistical Operations on Vectors). PROD computes the product of the elements of a vector, and MAX

computes the maximum of all the elements of a vector.

INV SUM computes the alternating sum of the first element minus the second, plus the third, minus the fourth, and so on. INV MAX computes the minimum of the vector elements.

HYP SUM computes the mean of the vector elements. HYP PROD computes the sample standard deviation. HYP MAX computes the median.

MAP* multiplies two vectors elementwise. It is equivalent to the V M * command. MAP^ computes powers elementwise. The arguments must be vectors of equal length, or one must be a vector and the other must be a plain number. For example, 2 MAP^ squares all the elements of a vector.

MAP$ maps the formula on the top of the stack across the vector in the second-to-top position. If the formula contains several variables, Calc takes that many vectors starting at the second-to-top position and matches them to the variables in alphabetical order. The result is a vector of the same size as the input vectors, whose elements are the formula evaluated with the variables set to the various sets of numbers in those vectors. For example, you could simulate MAP^ using MAP$ with the formula `x^y'.

The "x" key pushes the variable name x onto the stack. To build the formula x^2 + 6, you would use the key sequence "x" 2 y^x 6 +. This formula would then be suitable for use with the MAP$ key described above. With INV, HYP, or INV and HYP, the "x" key pushes the variable names y, z, and t, respectively.

Modes Menu

```
|----+----+----+----+----+----5
|FLT |FIX |SCI |ENG |GRP |    |
|----+----+----+----+----+----|
|RAD |DEG |FRAC|POLR|SYMB|PREC|
|----+----+----+----+----+----|
|SWAP|RLL3|RLL4|OVER|STO |RCL |
|----+----+----+----+----+----|
```

The keys in this menu manipulate modes, variables, and the stack.

The FLT, FIX, SCI, and ENG keys select floating-point, fixed-point, scientific, or engineering notation. FIX displays two digits after the decimal by default; the others display full precision. With the INV prefix, these keys pop a number-of-digits argument from the stack.

The GRP key turns grouping of digits with commas on or off. INV GRP enables grouping to the right of the decimal point as well as to the left.

The RAD and DEG keys switch between radians and degrees for trigonometric functions.

The FRAC key turns Fraction mode on or off. This affects whether commands like / with integer arguments produce fractional or floating-point results.

The POLR key turns Polar mode on or off, determining whether polar or rectangular complex numbers are used by default.

The SYMB key turns Symbolic mode on or off, in which operations that would produce inexact floating-point results are left unevaluated as algebraic formulas.

The PREC key selects the current precision. Answer with the keyboard or with the keypad digit and ENTER keys.

The SWAP key exchanges the top two stack elements. The RLL3 key rotates the top three stack elements upwards. The RLL4 key rotates the top four stack elements upwards. The OVER key duplicates the second-to-top stack element.

The STO and RCL keys are analogous to s t and s r in regular Calc. See section Storing and Recalling. Click the STO or RCL key, then one of the ten digits. (Named variables are not available in Keypad Mode.) You can also use, for example, STO + 3 to add to register 3.

Go to the previous, next section.

Go to the previous, next section.

Embedded Mode

Embedded Mode in Calc provides an alternative to copying numbers and formulas back and forth between editing buffers and the Calc stack. In Embedded Mode, your editing buffer becomes temporarily linked to the stack and this copying is taken care of automatically.

Basic Embedded Mode

To enter Embedded mode, position the Emacs point (cursor) on a formula in any buffer and press M-# e (`calc-embedded`). Note that M-# e is not to be used in the Calc stack buffer like most Calc commands, but rather in regular editing buffers that are visiting your own files.

Calc normally scans backward and forward in the buffer for the nearest opening and closing formula delimiters. The simplest delimiters are blank lines. Other delimiters that Embedded Mode understands are:

1. The TeX and LaTeX math delimiters `$ $', `$$ $$', `\[ \]', and `\(\)';
2. Lines beginning with `\begin' and `\end';
3. Lines beginning with `@' (Texinfo delimiters).
4. Lines beginning with `.EQ' and `.EN' (eqn delimiters);
5. Lines containing a single `%' or `.\"' symbol and nothing else.

See section Customizing Embedded Mode, to see how to make Calc recognize your own favorite delimiters. Delimiters like `$ $' can appear on their own separate lines or in-line with the formula.

If you give a positive or negative numeric prefix argument, Calc instead uses the current point as one end of the formula, and moves forward or backward (respectively) by that many lines to find the other end. Explicit delimiters are not necessary in this case.

With a prefix argument of zero, Calc uses the current region (delimited by point and mark) instead of formula delimiters.

With a prefix argument of C-u only, Calc scans for the first non-numeric character (i.e., the first character that is not a digit, sign, decimal point, or upper- or lower-case `e') forward and backward to delimit the formula. M-# w (`calc-embedded-word`) is equivalent to C-u M-# e.

When you enable Embedded mode for a formula, Calc reads the text between the delimiters and tries to interpret it as a Calc formula. It's best if the current Calc language mode is correct for the formula, but Calc can generally identify TeX formulas and Big-style formulas even if the language mode is wrong. If Calc can't make sense of the formula, it beeps and refuses to enter Embedded mode. But if the current language is wrong, Calc can sometimes parse the formula successfully (but incorrectly); for example, the C expression `atan(a[1])' can be parsed in Normal language mode, but the `atan` won't correspond to the built-in `arctan` function, and the `a[1]' will be interpreted as `a' times the vector `[1]'!

If you press M-# e or M-# w to activate an embedded formula which is blank, say with the cursor on space between the two delimiters `$ $', Calc will immediately prompt for an algebraic entry.

Only one formula in one buffer can be enabled at a time. If you move to another area of the current buffer and give Calc commands, Calc turns Embedded mode off for the old formula and then tries to restart Embedded mode at the new position. Other buffers are not affected by Embedded mode.

When Embedded mode begins, Calc pushes the current formula onto the stack. No Calc stack window is created; however, Calc copies the top-of-stack position into the original buffer at all times. You can create a Calc window by hand with M-# o if you find you need to see the entire stack.

For example, typing M-# e while somewhere in the formula `n>2' in the following line enables Embedded mode on that inequality:

```
We define $F_n = F_(n-1)+F_(n-2)$ for all $n>2$.
```

The formula n>2 will be pushed onto the Calc stack, and the top of stack will be copied back into the editing buffer. This means that spaces will appear around the `>' symbol to match Calc's usual display style:

```
We define $F_n = F_(n-1)+F_(n-2)$ for all $n > 2$.
```

No spaces have appeared around the `+' sign because it's in a different formula, one which we have not yet touched with Embedded mode.

Now that Embedded mode is enabled, keys you type in this buffer are interpreted as Calc commands. At this point we might use the "commute" command j C to reverse the inequality. This is a selection-based command for which we first need to move the cursor onto the operator (`>' in this case) that needs to be commuted.

```
We define $F_n = F_(n-1)+F_(n-2)$ for all $2 < n$.
```

The M-# o command is a useful way to open a Calc window without actually selecting that window. Giving this command verifies that `2 < n' is also on the Calc stack. Typing 17 RET would produce:

```
We define $F_n = F_(n-1)+F_(n-2)$ for all $17$.
```

with `2 < n' and `17' on the stack; typing TAB at this point will exchange the two stack values and restore `2 < n' to the embedded formula. Even though you can't normally see the stack in Embedded mode, it is still there and it still operates in the same way. But, as with old-fashioned RPN calculators, you can only see the value at the top of the stack at any given time (unless you use M-# o).

Typing M-# e again turns Embedded mode off. The Calc window reveals that the formula `2 < n' is automatically removed from the stack, but the `17' is not. Entering Embedded mode always pushes one thing onto the stack, and leaving Embedded mode always removes one thing. Anything else that happens on the stack is entirely your business as far as Embedded mode is concerned.

If you press M-# e in the wrong place by accident, it is possible that Calc will be able to parse the nearby

text as a formula and will mangle that text in an attempt to redisplay it "properly" in the current langu mode. If this happens, press M-# e again to exit Embedded mode, then give the regular Emacs "undo command (C-_ or C-x u) to put the text back the way it was before Calc edited it. Note that Calc's own Undo command (typed before you turn Embedded mode back off) will not do you any good, because as far as Calc is concerned you haven't done anything with this formula yet.

More About Embedded Mode

When Embedded mode "activates" a formula, i.e., when it examines the formula for the first time since the buffer was created or loaded, Calc tries to sense the language in which the formula was written. If the formula contains any TeX-like `\' sequences, it is parsed (i.e., read) in TeX mode. If the formula appears to be written in multi-line Big mode, it is parsed in Big mode. Otherwise, it is parsed according to the current language mode.

Note that Calc does not change the current language mode according to what it finds. Even though it can read a TeX formula when not in TeX mode, it will immediately rewrite this formula using whatever language mode is in effect. You must then type d T to switch Calc permanently into TeX mode if that is what you desire.

Calc's parser is unable to read certain kinds of formulas. For example, with v] (`calc-matrix-brackets`) you can specify matrix display styles which the parser is unable to recognize as matrices. The d p (`calc-show-plain`) command turns on a mode in which a "plain" version of a formula is placed in front of the fully-formatted version. When Calc reads a formula that has such a plain version in front, it reads the plain version and ignores the formatted version.

Plain formulas are preceded and followed by `%%%' signs by default. This notation has the advantage that the `%' character begins a comment in TeX, so if your formula is embedded in a TeX document its plain version will be invisible in the final printed copy. See section Customizing Embedded Mode, to see how to change the "plain" formula delimiters, say to something that eqn or some other formatter will treat as a comment.

There are several notations which Calc's parser for "big" formatted formulas can't yet recognize. In particular, it can't read the large symbols for `sum`, `prod`, and `integ`, and it can't handle `=>' with the righthand argument omitted. Also, Calc won't recognize special formats you have defined with the Z C command (see section User-Defined Compositions). In these cases it is important to use "plain" mode to make sure Calc will be able to read your formula later.

Another example where "plain" mode is important is if you have specified a float mode with few digits of precision. Normally any digits that are computed but not displayed will simply be lost when you save and re-load your embedded buffer, but "plain" mode allows you to make sure that the complete number is present in the file as well as the rounded-down number.

Embedded buffers remember active formulas for as long as they exist in Emacs memory. Suppose you have an embedded formula which is @c{π} pi to the normal 12 decimal places, and then type C-u 5 d n to display only five decimal places. If you then type d n, all 12 places reappear because the full number is still there on the Calc stack. More surprisingly, even if you exit Embedded mode and later re-enter it for that formula, typing d n will restore all 12 places because each buffer remembers all its active

formulas. However, if you save the buffer in a file and reload it in a new Emacs session, all non-displayed digits will have been lost unless you used "plain" mode.

In some applications of Embedded mode, you will want to have a sequence of copies of a formula that show its evolution as you work on it. For example, you might want to have a sequence like this in your file (elaborating here on the example from the "Getting Started" chapter):

```
The derivative of

                                ln(ln(x))

is

                    (the derivative of ln(ln(x)))

whose value at x = 2 is

                               (the value)

and at x = 3 is

                               (the value)
```

The M-# d (`calc-embedded-duplicate`) command is a handy way to make sequences like this. If you type M-# d, the formula under the cursor (which may or may not have Embedded mode enabled for it at the time) is copied immediately below and Embedded mode is then enabled for that copy.

For this example, you would start with just

```
The derivative of

                                ln(ln(x))
```

and press M-# d with the cursor on this formula. The result is

```
The derivative of

                                ln(ln(x))


                                ln(ln(x))
```

with the second copy of the formula enabled in Embedded mode. You can now press a d x RET to take the derivative, and M-# d M-# d to make two more copies of the derivative. To complete the computations, type 3 s l x RET to evaluate the last formula, then move up to the second-to-last formula and type 2 s l x RET.

Finally, you would want to press M-# e to exit Embedded mode, then go up and insert the necessary in between the various formulas and numbers.

The M-# f (`calc-embedded-new-formula`) command creates a new embedded formula at the current point. It inserts some default delimiters, which are usually just blank lines, and then does an algebraic entry to get the formula (which is then enabled for Embedded mode). This is just shorthand for typing the delimiters yourself, positioning the cursor between the new delimiters, and pressing M-# e. The key sequence M-# ' is equivalent to M-# f.

The M-# n (`calc-embedded-next`) and M-# p (`calc-embedded-previous`) commands move the cursor to the next or previous active embedded formula in the buffer. They can take positive or negative prefix arguments to move by several formulas. Note that these commands do not actually examine the text of the buffer looking for formulas; they only see formulas which have previously been activated in Embedded mode. In fact, M-# n and M-# p are a useful way to tell which embedded formulas are currently active. Also, note that these commands do not enable Embedded mode on the next or previous formula, they just move the cursor. (By the way, M-# n is not as awkward to type as it may seem, because M-# ignores Shift and Meta on the second keystroke: M-# M-N can be typed by holding down Shift and Meta and alternately typing two keys.)

The M-# ` (`calc-embedded-edit`) command edits the embedded formula at the current point as if by ` (`calc-edit`). Embedded mode does not have to be enabled for this to work. Press M-# M-# to finish the edit, or M-# x to cancel.

Assignments in Embedded Mode

The `:=' (assignment) and `=>' ("evaluates-to") operators are especially useful in Embedded mode. They allow you to make a definition in one formula, then refer to that definition in other formulas embedded in the same buffer.

An embedded formula which is an assignment to a variable, as in

```
foo := 5
```

records 5 as the stored value of `foo` for the purposes of Embedded mode operations in the current buffer. It does *not* actually store 5 as the "global" value of `foo`, however. Regular Calc operations, and Embedded formulas in other buffers, will not see this assignment.

One way to use this assigned value is simply to create an Embedded formula elsewhere that refers to `foo`, and to press = in that formula. However, this permanently replaces the `foo` in the formula with its current value. More interesting is to use `=>' elsewhere:

```
foo + 7 => 12
```

See section The Evaluates-To Operator, for a general discussion of `=>'.

If you move back and change the assignment to `foo`, any `=>' formulas which refer to it are automatically updated.


```
foo := 17

foo + 7 => 24
```

The obvious question then is, *how* can one easily change the assignment to `foo`? If you simply select the formula in Embedded mode and type 17, the assignment itself will be replaced by the 17. The effect on the other formula will be that the variable `foo` becomes unassigned:

```
17

foo + 7 => foo + 7
```

The right thing to do is first to use a selection command (j 2 will do the trick) to select the righthand side of the assignment. Then, 17 TAB DEL will swap the 17 into place (see section Selecting Sub-Formulas, to see how this works).

The M-# j (`calc-embedded-select`) command provides an easy way to operate on assigments. It is just like M-# e, except that if the enabled formula is an assignment, it uses j 2 to select the righthand side. If the enabled formula is an evaluates-to, it uses j 1 to select the lefthand side. A formula can also be a combination of both:

```
bar := foo + 3 => 20
```

in which case M-# j will select the middle part (`foo + 3').

The formula is automatically deselected when you leave Embedded mode.

Another way to change the assignment to `foo` would simply be to edit the number using regular Emacs editing rather than Embedded mode. Then, we have to find a way to get Embedded mode to notice the change. The M-# u or M-# = (`calc-embedded-update-formula`) command is a convenient way to do this.

```
foo := 6

foo + 7 => 13
```

Pressing M-# u is much like pressing M-# e = M-# e, that is, temporarily enabling Embedded mode for the formula under the cursor and then evaluating it with =. But M-# u does not actually use M-# e, and in fact another formula somewhere else can be enabled in Embedded mode while you use M-# u and that formula will not be disturbed.

With a numeric prefix argument, M-# u updates all active `=>' formulas in the buffer. Formulas which have not yet been activated in Embedded mode, and formulas which do not have `=>' as their top-level operator, are not affected by this. (This is useful only if you have used m C; see below.)

With a plain C-u prefix, C-u M-# u updates only in the region between mark and point rather than in the whole buffer.

M-# u is also a handy way to activate a formula, such as an `=>' formula that has freshly been typed in or loaded from a file.

The M-# a (`calc-embedded-activate`) command scans through the current buffer and activates all embedded formulas that contain `:=' or `=>' symbols. This does not mean that Embedded mode is actually turned on, but only that the formulas' positions are registered with Embedded mode so that the `=>' values can be properly updated as assignments are changed.

It is a good idea to type M-# a right after loading a file that uses embedded `=>' operators. Emacs includes a nifty "buffer-local variables" feature that you can use to do this automatically. The idea is to place near the end of your file a few lines that look like this:

```
--- Local Variables: ---
--- eval:(calc-embedded-activate) ---
--- End: ---
```

where the leading and trailing `---' can be replaced by any suitable strings (which must be the same on all three lines) or omitted altogether; in a TeX file, `%' would be a good leading string and no trailing string would be necessary. In a C program, `/*' and `*/' would be good leading and trailing strings.

When Emacs loads a file into memory, it checks for a Local Variables section like this one at the end of the file. If it finds this section, it does the specified things (in this case, running M-# a automatically) before editing of the file begins. The Local Variables section must be within 3000 characters of the end of the file for Emacs to find it, and it must be in the last page of the file if the file has any page separators. See section 'Local Variables in Files' in the Emacs manual.

Note that M-# a does not update the formulas it finds. To do this, type, say, M-1 M-# u after M-# a. Generally this should not be a problem, though, because the formulas will have been up-to-date already when the file was saved.

Normally, M-# a activates all the formulas it finds, but any previous active formulas remain active as well. With a positive numeric prefix argument, M-# a first deactivates all current active formulas, then actives the ones it finds in its scan of the buffer. With a negative prefix argument, M-# a simply deactivates all formulas.

Embedded mode has two symbols, `Active' and `~Active', which it puts next to the major mode name in a buffer's mode line. It puts `Active' if it has reason to believe that all formulas in the buffer are active, because you have typed M-# a and Calc has not since had to deactivate any formulas (which can happen if Calc goes to update an `=>' formula somewhere because a variable changed, and finds that the formula is no longer there due to some kind of editing outside of Embedded mode). Calc puts `~Active' in the mode line if some, but probably not all, formulas in the buffer are active. This happens if you activate a few formulas one at a time but never use M-# a, or if you used M-# a but then Calc had to deactivate a formula because it lost track of it. If neither of these symbols appears in the mode line, no embedded formulas are active in the buffer (e.g., before Embedded mode has been used, or after a M-- M-# a).

Embedded formulas can refer to assignments both before and after them in the buffer. If there are several assignments to a variable, the nearest preceding assignment is used if there is one, otherwise the following assignment is used.


```
x => 1

x := 1

x => 1

x := 2

x => 2
```

As well as simple variables, you can also assign to subscript expressions of the form `var_number' (as in `x_0`), or `var_var' (as in `x_max`). Assignments to other kinds of objects can be represented by Calc, but the automatic linkage between assignments and references works only for plain variables and these two kinds of subscript expressions.

If there are no assignments to a given variable, the global stored value for the variable is used (see section Storing Variables), or, if no value is stored, the variable is left in symbolic form. Note that global stored values will be lost when the file is saved and loaded in a later Emacs session, unless you have used the s p (`calc-permanent-variable`) command to save them; see section Other Operations on Variables.

The m C (`calc-auto-recompute`) command turns automatic recomputation of `=>' forms on and off. If you turn automatic recomputation off, you will have to use M-# u to update these formulas manually after an assignment has been changed. If you plan to change several assignments at once, it may be more efficient to type m C, change all the assignments, then use M-1 M-# u to update the entire buffer afterwards. The m C command also controls `=>' formulas on the stack; see section The Evaluates-To Operator. When you turn automatic recomputation back on, the stack will be updated but the Embedded buffer will not; you must use M-# u to update the buffer by hand.

Mode Settings in Embedded Mode

Embedded Mode has a rather complicated mechanism for handling mode settings in Embedded formulas. It is possible to put annotations in the file that specify mode settings either global to the entire file or local to a particular formula or formulas. In the latter case, different modes can be specified for use when a formula is the enabled Embedded Mode formula.

When you give any mode-setting command, like m f (for fraction mode) or d s (for scientific notation), Embedded Mode adds a line like the following one to the file just before the opening delimiter of the formula.

```
% [calc-mode: fractions: t]
% [calc-mode: float-format: (sci 0)]
```

When Calc interprets an embedded formula, it scans the text before the formula for mode-setting annotations like these and sets the Calc buffer to match these modes. Modes not explicitly described in

the file are not changed. Calc scans all the way to the top of the file, or up to a line of the form

```
% [calc-defaults]
```

which you can insert at strategic places in the file if this backward scan is getting too slow, or just to provide a barrier between one "zone" of mode settings and another.

If the file contains several annotations for the same mode, the closest one before the formula is used. Annotations after the formula are never used (except for global annotations, described below).

The scan does not look for the leading `% ', only for the square brackets and the text they enclose. You can edit the mode annotations to a style that works better in context if you wish. See section Customizing Embedded Mode, to see how to change the style that Calc uses when it generates the annotations. You can write mode annotations into the file yourself if you know the syntax; the easiest way to find the syntax for a given mode is to let Calc write the annotation for it once and see what it does.

If you give a mode-changing command for a mode that already has a suitable annotation just above the current formula, Calc will modify that annotation rather than generating a new, conflicting one.

Mode annotations have three parts, separated by colons. (Spaces after the colons are optional.) The first identifies the kind of mode setting, the second is a name for the mode itself, and the third is the value in the form of a Lisp symbol, number, or list. Annotations with unrecognizable text in the first or second parts are ignored. The third part is not checked to make sure the value is of a legal type or range; if you write an annotation by hand, be sure to give a proper value or results will be unpredictable. Mode-setting annotations are case-sensitive.

While Embedded Mode is enabled, the word `Local` appears in the mode line. This is to show that mode setting commands generate annotations that are "local" to the current formula or set of formulas. The m R (`calc-mode-record-mode`) command causes Calc to generate different kinds of annotations. Pressing m R repeatedly cycles through the possible modes.

`LocEdit` and `LocPerm` modes generate annotations that look like this, respectively:

```
% [calc-edit-mode: float-format: (sci 0)]
% [calc-perm-mode: float-format: (sci 5)]
```

The first kind of annotation will be used only while a formula is enabled in Embedded Mode. The second kind will be used only when the formula is *not* enabled. (Whether the formula is "active" or not, i.e., whether Calc has seen this formula yet, is not relevant here.)

`Global` mode generates an annotation like this at the end of the file:

```
% [calc-global-mode: fractions t]
```

Global mode annotations affect all formulas throughout the file, and may appear anywhere in the file. This allows you to tuck your mode annotations somewhere out of the way, say, on a new page of the file, as long as those mode settings are suitable for all formulas in the file.

Enabling a formula with M-# e causes a fresh scan for local mode annotations; you will have to use this

after adding annotations above a formula by hand to get the formula to notice them. Updating a form with M-# u will also re-scan the local modes, but global modes are only re-scanned by M-# a.

Another way that modes can get out of date is if you add a local mode annotation to a formula that has another formula after it. In this example, we have used the d s command while the first of the two embedded formulas is active. But the second formula has not changed its style to match, even though by the rules of reading annotations the `(sci 0)' applies to it, too.

```
% [calc-mode: float-format: (sci 0)]
1.23e2

456.
```

We would have to go down to the other formula and press M-# u on it in order to get it to notice the new annotation.

Two more mode-recording modes selectable by m R are `Save` (which works even outside of Embedded Mode), in which mode settings are recorded permanently in your Emacs startup file `` `~/.emacs' `` rather than by annotating the current document, and no-recording mode (where there is no symbol like `Save` or `Local` in the mode line), in which mode-changing commands do not leave any annotations at all.

When Embedded Mode is not enabled, mode-recording modes except for `Save` have no effect.

Customizing Embedded Mode

You can modify Embedded Mode's behavior by setting various Lisp variables described here. Use M-x set-variable or M-x edit-options to adjust a variable on the fly, or put a suitable `setq` statement in your `` `~/.emacs' `` file to set a variable permanently. (Another possibility would be to use a file-local variable annotation at the end of the file; see section 'Local Variables in Files' in the Emacs manual.)

While none of these variables will be buffer-local by default, you can make any of them local to any embedded-mode buffer. (Their values in the `*Calculator*' buffer are never used.)

The `calc-embedded-open-formula` variable holds a regular expression for the opening delimiter of a formula. See section 'Regular Expression Search' in the Emacs manual, to see how regular expressions work. Basically, a regular expression is a pattern that Calc can search for. A regular expression that considers blank lines, `$', and `$$' to be opening delimiters is `"\\`\\|^\n\\|\\$\\$?"`. Just in case the meaning of this regular expression is not completely plain, let's go through it in detail.

The surrounding `" "' marks quote the text between them as a Lisp string. If you left them off, `set-variable` or `edit-options` would try to read the regular expression as a Lisp program.

The most obvious property of this regular expression is that it contains indecently many backslashes. There are actually two levels of backslash usage going on here. First, when Lisp reads a quoted string, all pairs of characters beginning with a backslash are interpreted as special characters. Here, `\n` changes to a

new-line character, and \\ changes to a single backslash. So the actual regular expression seen by C is `\`\|^ (newline) \|\$\$?'.

Regular expressions also consider pairs beginning with backslash to have special meanings. Sometimes the backslash is used to quote a character that otherwise would have a special meaning in a regular expression, like `$', which normally means "end-of-line," or `?', which means that the preceding item is optional. So `\$\$?' matches either one or two dollar signs.

The other codes in this regular expression are `^', which matches "beginning-of-line," `\|', which means "or," and `\`', which matches "beginning-of-buffer." So the whole pattern means that a formula begins at the beginning of the buffer, or on a newline that occurs at the beginning of a line (i.e., a blank line), or at one or two dollar signs.

The default value of `calc-embedded-open-formula` looks just like this example, with several more alternatives added on to recognize various other common kinds of delimiters.

By the way, the reason to use `^\n' rather than `^$' or `\n\n', which also would appear to match blank lines, is that the former expression actually "consumes" only one newline character as *part of* the delimiter, whereas the latter expressions consume zero or two newlines, respectively. The former choice gives the most natural behavior when Calc must operate on a whole formula including its delimiters.

See the Emacs manual for complete details on regular expressions. But just for your convenience, here is a list of all characters which must be quoted with backslash (like `\$') to avoid some special interpretation: `. * + ? [ ] ^ $ \'. (Note the backslash in this list; for example, to match `\[' you must use `"\\\\\\["`. An exercise for the reader is to account for each of these six backslashes!)

The `calc-embedded-close-formula` variable holds a regular expression for the closing delimiter of a formula. A closing regular expression to match the above example would be `"\\'\\|\n$\\|\\$\\$?"`. This is almost the same as the other one, except it now uses `\'' ("end-of-buffer") and `\n$' (newline occurring at end of line, yet another way of describing a blank line that is more appropriate for this case).

The `calc-embedded-open-word` and `calc-embedded-close-word` variables are similar expressions used when you type M-# w instead of M-# e to enable Embedded mode.

The `calc-embedded-open-plain` variable is a string which begins a "plain" formula written in front of the formatted formula when d p mode is turned on. Note that this is an actual string, not a regular expression, because Calc must be able to write this string into a buffer as well as to recognize it. The default string is `"%%% "` (note the trailing space).

The `calc-embedded-close-plain` variable is a string which ends a "plain" formula. The default is `" %%%\n"`. Without the trailing newline here, the first line of a "big" mode formula that followed might be shifted over with respect to the other lines.

The `calc-embedded-open-new-formula` variable is a string which is inserted at the front of a new formula when you type M-# f. Its default value is `"\n\n"`. If this string begins with a newline character and the M-# f is typed at the beginning of a line, M-# f will skip this first newline to avoid introducing unnecessary blank lines in the file.

The `calc-embedded-close-new-formula` variable is the corresponding string which is inserted

at the end of a new formula. Its default value is also `"\n\n"`. The final newline is omitted by M-# f typed at the end of a line. (It follows that if M-# f is typed on a blank line, both a leading opening newline and a trailing closing newline are omitted.)

The `calc-embedded-announce-formula` variable is a regular expression which is sure to be followed by an embedded formula. The M-# a command searches for this pattern as well as for `=>' and `:=' operators. Note that M-# a will not activate just anything surrounded by formula delimiters; after all, blank lines are considered formula delimiters by default! But if your language includes a delimiter which can only occur actually in front of a formula, you can take advantage of it here. The default pattern is `"%Embed\n\\(% .*\n\\)*"`, which checks for `%Embed' followed by any number of lines beginning with `%' and a space. This last is important to make Calc consider mode annotations part of the pattern, so that the formula's opening delimiter really is sure to follow the pattern.

The `calc-embedded-open-mode` variable is a string (not a regular expression) which should precede a mode annotation. Calc never scans for this string; Calc always looks for the annotation itself. But this is the string that is inserted before the opening bracket when Calc adds an annotation on its own. The default is `"% "`.

The `calc-embedded-close-mode` variable is a string which follows a mode annotation written by Calc. Its default value is simply a newline, `"\n"`. If you change this, it is a good idea still to end with a newline so that mode annotations will appear on lines by themselves.

Go to the previous, next section.

Go to the previous, next section.

Programming

There are several ways to "program" the Emacs Calculator, depending on the nature of the problem you need to solve.

1. Keyboard macros allow you to record a sequence of keystrokes and play them back at a later time. This is just the standard Emacs keyboard macro mechanism, dressed up with a few more features such as loops and conditionals.
2. Algebraic definitions allow you to use any formula to define a new function. This function can then be used in algebraic formulas or as an interactive command.
3. Rewrite rules are discussed in the section on algebra commands. See section Rewrite Rules. If you put your rewrite rules in the variable `EvalRules`, they will be applied automatically to all Calc results in just the same way as an internal "rule" is applied to evaluate `sqrt(9)' to 3 and so on. See section Automatic Rewrites.
4. Lisp is the programming language that Calc (and most of Emacs) is written in. If the above techniques aren't powerful enough, you can write Lisp functions to do anything that built-in Calc commands can do. Lisp code is also somewhat faster than keyboard macros or rewrite rules.

Programming features are available through the z and Z prefix keys. New commands that you define are two-key sequences beginning with z. Commands for managing these definitions use the shift-Z prefix. (The Z T (`calc-timing`) command is described elsewhere; see section Troubleshooting Commands. The Z C (`calc-user-define-composition`) command is also described elsewhere; see section User-Defined Compositions.)

Creating User Keys

Any Calculator command may be bound to a key using the Z D (`calc-user-define`) command. Actually, it is bound to a two-key sequence beginning with the lower-case z prefix.

The Z D command first prompts for the key to define. For example, press Z D a to define the new key sequence z a. You are then prompted for the name of the Calculator command that this key should run. For example, the `calc-sincos` command is not normally available on a key. Typing Z D s sincos RET programs the z s key sequence to run `calc-sincos`. This definition will remain in effect for the rest of this Emacs session, or until you redefine z s to be something else.

You can actually bind any Emacs command to a z key sequence by backspacing over the `calc-' when you are prompted for the command name.

As with any other prefix key, you can type z ? to see a list of all the two-key sequences you have defined that start with z. Initially, no z sequences (except z ? itself) are defined.

User keys are typically letters, but may in fact be any key. (META-keys are not permitted, nor are a terminal's special function keys which generate multi-character sequences when pressed.) You can define different commands on the shifted and unshifted versions of a letter if you wish.

The Z U (`calc-user-undefine`) command unbinds a user key. For example, the key sequence Z U s will

undefine the `sincos` key we defined above.

The Z P (`calc-user-define-permanent`) command makes a key binding permanent so that it will remain in effect even in future Emacs sessions. (It does this by adding a suitable bit of Lisp code into your `` `.emacs' `` file.) For example, Z P s would register our `sincos` command permanently. If you later wish to unregister this command you must edit your `` `.emacs' `` file by hand. (See section General Mode Commands, for a way to tell Calc to use a different file instead of `` `.emacs' ``.)

The Z P command also saves the user definition, if any, for the command bound to the key. After Z F and Z C, a given user key could invoke a command, which in turn calls an algebraic function, which might have one or more special display formats. A single Z P command will save all of these definitions.

To save a command or function without its key binding (or if there is no key binding for the command or function), type ' (the apostrophe) when prompted for a key. Then, type the function name, or backspace to change the `calcFunc-' prefix to `calc-' and enter a command name. (If the command you give implies a function, the function will be saved, and if the function has any display formats, those will be saved, but not the other way around: Saving a function will not save any commands or key bindings associated with the function.)

The Z E (`calc-user-define-edit`) command edits the definition of a user key. This works for keys that have been defined by either keyboard macros or formulas; further details are contained in the relevant following sections.

Programming with Keyboard Macros

The easiest way to "program" the Emacs Calculator is to use standard keyboard macros. Press C-x (to begin recording a macro. From this point on, keystrokes you type will be saved away as well as performing their usual functions. Press C-x) to end recording. Press shift-X (or the standard Emacs key sequence C-x e) to execute your keyboard macro by replaying the recorded keystrokes. See section 'Keyboard Macros' in the Emacs Manual, for further information.

When you use X to invoke a keyboard macro, the entire macro is treated as a single command by the undo and trail features. The stack display buffer is not updated during macro execution, but is instead fixed up once the macro completes. Thus, commands defined with keyboard macros are convenient and efficient. The C-x e command, on the other hand, invokes the keyboard macro with no special treatment: Each command in the macro will record its own undo information and trail entry, and update the stack buffer accordingly. If your macro uses features outside of Calc's control to operate on the contents of the Calc stack buffer, or if it includes Undo, Redo, or last-arguments commands, you must use C-x e to make sure the buffer and undo list are up-to-date at all times. You could also consider using K (`calc-keep-args`) instead of M-RET (`calc-last-args`).

Calc extends the standard Emacs keyboard macros in several ways. Keyboard macros can be used to create user-defined commands. Keyboard macros can include conditional and iteration structures, somewhat analogous to those provided by a traditional programmable calculator.

Naming Keyboard Macros

Once you have defined a keyboard macro, you can bind it to a z key sequence with the Z K (`calc-user-define-kbd-macro`) command. This command prompts first for a key, then for a command name. For example, if you type C-x (n TAB n TAB C-x) you will define a keyboard macro which negates the top two numbers on the stack (TAB swaps the top two stack elements). Now you can type Z K n RET to define

this keyboard macro onto the z n key sequence. The default command name (if you answer the second pror with just the RET key as in this example) will be something like `calc-User-n'. The keyboard macro will nc available as both z n and M-x calc-User-n. You can backspace and enter a more descriptive command name if you wish.

Macros defined by Z K act like single commands; they are executed in the same way as by the X key. If you wish to define the macro as a standard no-frills Emacs macro (to be executed as if by C-x e), give a negative prefix argument to Z K.

Once you have bound your keyboard macro to a key, you can use Z P to register it permanently with Emacs. See section Creating User Keys.

The Z E (`calc-user-define-edit`) command on a key that has been defined by a keyboard macro tries to use the `edit-kbd-macro` command to edit the macro. This command may be found in the `` `macedit' `` package, a copy of which comes with Calc. It decomposes the macro definition into full Emacs command names, like `calc-pop` and `calc-add`. Type M-# M-# to finish editing and update the definition stored on the key, or, to cancel the edit, type M-# x.

If you give a negative numeric prefix argument to Z E, the keyboard macro is edited in spelled-out keystroke form. For example, the editing buffer might contain the nine characters `1 RET 2 +'. When you press M-# M-#, the `read-kbd-macro` feature of the `` `macedit' `` package is used to reinterpret these key names. The notations `RET`, `LFD`, `TAB`, `SPC`, `DEL`, and `NUL` must be written in all uppercase, as must the prefixes `C-` and `M-`. Spaces and line breaks are ignored. Other characters are copied verbatim into the keyboard macro. Basically, the notation is the same as is used in all of this manual's examples, except that the manual takes some liberties with spaces: When we say ' [1 2 3] RET, we take it for granted that it is clear we really mean ' [1 SPC 2 SPC 3] RET, which is what `read-kbd-macro` wants to see.

If `` `macedit' `` is not available, Z E edits the keyboard macro in "raw" form; the editing buffer simply contains characters like `1^M2+' (here `^M' represents the carriage-return character). Editing in this mode, you will have to use C-q to enter new control characters into the buffer.

The M-# m (`read-kbd-macro`) command reads an Emacs "region" of spelled-out keystrokes and defines it as the current keyboard macro. It is a convenient way to define a keyboard macro that has been stored in a file, or to define a macro without executing it at the same time. The M-# m command works only if `` `macedit' `` is present.

Conditionals in Keyboard Macros

The Z [(`calc-kbd-if`) and Z] (`calc-kbd-end-if`) commands allow you to put simple tests in a keyboard macro. When Calc sees the Z [, it pops an object from the stack and, if the object is a non-zero value, continues executing keystrokes. But if the object is zero, or if it is not provably nonzero, Calc skips ahead to the matching Z] keystroke. See section Logical Operations, for a set of commands for performing tests which conveniently produce 1 for true and 0 for false.

For example, RET 0 a < Z [n Z] implements an absolute-value function in the form of a keyboard macro. This macro duplicates the number on the top of the stack, pushes zero and compares using a < (`calc-less-than`), then, if the number was less than zero, executes n (`calc-change-sign`). Otherwise, the change-sign command is skipped.

To program this macro, type C-x (, type the above sequence of keystrokes, then type C-x). Note that the keystrokes will be executed while you are making the definition as well as when you later re-execute the macro

by typing X. Thus you should make sure a suitable number is on the stack before defining the macro so that you don't get a stack-underflow error during the definition process.

Conditionals can be nested arbitrarily. However, there should be exactly one Z] for each Z [in a keyboard macro.

The Z : (`calc-kbd-else`) command allows you to choose between two keystroke sequences. The general format is cond Z [then-part Z : else-part Z]. If cond is true (i.e., if the top of stack contains a non-zero number after cond has been executed), the then-part will be executed and the else-part will be skipped. Otherwise, the then-part will be skipped and the else-part will be executed.

The Z | (`calc-kbd-else-if`) command allows you to choose between any number of alternatives. For example, cond1 Z [part1 Z : cond2 Z | part2 Z : part3 Z] will execute part1 if cond1 is true, otherwise it will execute part2 if cond2 is true, otherwise it will execute part3.

More precisely, Z [pops a number and conditionally skips to the next matching Z : or Z] key. Z] has no effect when actually executed. Z : skips to the next matching Z]. Z | pops a number and conditionally skips to the next matching Z : or Z]; thus, Z [and Z | are functionally equivalent except that Z [participates in nesting but Z | does not.

Calc's conditional and looping constructs work by scanning the keyboard macro for occurrences of character sequences like `Z:' and `Z]'. One side-effect of this is that if you use these constructs you must be careful that these character pairs do not occur by accident in other parts of the macros. Since Calc rarely uses shift-Z for any purpose except as a prefix character, this is not likely to be a problem. Another side-effect is that it will not work to define your own custom key bindings for these commands. Only the standard shift-Z bindings will work correctly.

If Calc gets stuck while skipping characters during the definition of a macro, type Z C-g to cancel the definition. (Typing plain C-g actually adds a C-g keystroke to the macro.)

Loops in Keyboard Macros

The Z < (`calc-kbd-repeat`) and Z > (`calc-kbd-end-repeat`) commands pop a number from the stack, which must be an integer, then repeat the keystrokes between the brackets the specified number of times. If the integer is zero or negative, the body is skipped altogether. For example, 1 TAB Z < 2 * Z > computes two to a nonnegative integer power. First, we push 1 on the stack and then swap the integer argument back to the top. The Z < pops that argument leaving the 1 back on top of the stack. Then, we repeat a multiply-by-two step however many times.

Once again, the keyboard macro is executed as it is being entered. In this case it is especially important to set up reasonable initial conditions before making the definition: Suppose the integer 1000 just happened to be sitting on the stack before we typed the above definition! Another approach is to enter a harmless dummy definition for the macro, then go back and edit in the real one with a Z E command. Yet another approach is to type the macro as written-out keystroke names in a buffer, then use M-# m (`read-kbd-macro`) to read the macro.

The Z / (`calc-kbd-break`) command allows you to break out of a keyboard macro loop prematurely. It pops an object from the stack; if that object is true (a non-zero number), control jumps out of the innermost enclosing Z < ... Z > loop and continues after the Z >. If the object is false, the Z / has no effect. Thus cond Z / is similar to `if (cond) break;' in the C language.

The Z ((`calc-kbd-for`) and Z) (`calc-kbd-end-for`) commands are similar to Z < and Z >, except that they make the value of the counter available inside the loop. The general layout is init final Z (body step Z).

The Z (command pops initial and final values from the stack. It then creates a temporary internal counter a initializes it with the value init. The Z (command then repeatedly pushes the counter value onto the stack a executes body and step, adding step to the counter each time until the loop finishes.

By default, the loop finishes when the counter becomes greater than (or less than) final, assuming initial is less than (greater than) final. If initial is equal to final, the body executes exactly once. The body of the loop always executes at least once. For example, 0 1 10 Z (2 ^ + 1 Z) computes the sum of the squares of the integers from 1 to 10, in steps of 1.

If you give a numeric prefix argument of 1 to Z (, the loop is forced to use upward-counting conventions. In this case, if initial is greater than final the body will not be executed at all. Note that step may still be negative in this loop; the prefix argument merely constrains the loop-finished test. Likewise, a prefix argument of *-1* forces downward-counting conventions.

The Z { (`calc-kbd-loop`) and Z } (`calc-kbd-end-loop`) commands are similar to Z < and Z >, except that they do not pop a count from the stack--they effectively create an infinite loop. Every Z { ... Z } loop ought to include at least one Z / to make sure the loop doesn't run forever. (If any error message occurs which causes Emacs to beep, the keyboard macro will also be halted; this is a standard feature of Emacs. You can also generally press C-g to halt a running keyboard macro, although not all versions of Unix support this feature.)

The conditional and looping constructs are not actually tied to keyboard macros, but they are most often used in that context. For example, the keystrokes 10 Z < 23 RET Z > push ten copies of 23 onto the stack. This can be typed "live" just as easily as in a macro definition.

See section Conditionals in Keyboard Macros, for some additional notes about conditional and looping commands.

Local Values in Macros

Keyboard macros sometimes want to operate under known conditions without affecting surrounding conditions. For example, a keyboard macro may wish to turn on Fraction Mode, or set a particular precision, independent of the user's normal setting for those modes.

Macros also sometimes need to use local variables. Assignments to local variables inside the macro should not affect any variables outside the macro. The Z ` (`calc-kbd-push`) and Z ' (`calc-kbd-pop`) commands give you both of these capabilities.

When you type Z ` (with a backquote or accent grave character), the values of various mode settings are saved away. The ten "quick" variables `q0` through `q9` are also saved. When you type Z ' (with an apostrophe), these values are restored. Pairs of Z ` and Z ' commands may be nested.

If a keyboard macro halts due to an error in between a Z ` and a Z ', the saved values will be restored correctly even though the macro never reaches the Z ' command. Thus you can use Z ` and Z ' without having to worry about what happens in exceptional conditions.

If you type Z ` "live" (not in a keyboard macro), Calc puts you into a "recursive edit." You can tell you are in a recursive edit because there will be extra square brackets in the mode line, as in `[(Calculator)]'. These brackets will go away when you type the matching Z ' command. The modes and quick variables will be saved and restored in just the same way as if actual keyboard macros were involved.

The modes saved by Z ` and Z ' are the current precision and binary word size, the angular mode (Deg, Rad, or HMS), the simplification mode, Algebraic mode, Symbolic mode, Infinite mode, Matrix or Scalar mode,

Fraction mode, and the current complex mode (Polar or Rectangular). The ten "quick" variables' values (or thereof) are also saved.

Most mode-setting commands act as toggles, but with a numeric prefix they force the mode either on (positive prefix) or off (negative or zero prefix). Since you don't know what the environment might be when you invoke your macro, it's best to use prefix arguments for all mode-setting commands inside the macro.

In fact, C-u Z ` is like Z ` except that it sets the modes listed above to their default values. As usual, the matching Z ' will restore the modes to their settings from before the C-u Z `. Also, Z ` with a negative prefix argument resets algebraic mode to its default (off) but leaves the other modes the same as they were outside the construct.

The contents of the stack and trail, values of non-quick variables, and other settings such as the language mode and the various display modes, are *not* affected by Z ` and Z '.

Queries in Keyboard Macros

The Z = (`calc-kbd-report`) command displays an informative message including the value on the top of the stack. You are prompted to enter a string. That string, along with the top-of-stack value, is displayed unless m w (`calc-working`) has been used to turn such messages off.

The Z # (`calc-kbd-query`) command displays a prompt message (which you enter during macro definition), then does an algebraic entry which takes its input from the keyboard, even during macro execution. This command allows your keyboard macros to accept numbers or formulas as interactive input. All the normal conventions of algebraic input, including the use of $ characters, are supported.

See section 'Kbd Macro Query' in the Emacs Manual, for a description of C-x q (`kbd-macro-query`), the standard Emacs way to accept keyboard input during a keyboard macro. In particular, you can use C-x q to enter a recursive edit, which allows the user to perform any Calculator operations interactively before pressing C-M-c to return control to the keyboard macro.

Invocation Macros

Calc provides one special keyboard macro, called up by M-# z (`calc-user-invocation`), that is intended to allow you to define your own special way of starting Calc. To define this "invocation macro," create the macro in the usual way with C-x (and C-x), then type Z I (`calc-user-define-invocation`). There is only one invocation macro, so you don't need to type any additional letters after Z I. From now on, you can type M-# z at any time to execute your invocation macro.

For example, suppose you find yourself often grabbing rectangles of numbers into Calc and multiplying their columns. You can do this by typing M-# r to grab, and V R : * to multiply columns. To make this into an invocation macro, just type C-x (M-# r V R : * C-x), then Z I. Then, to multiply a rectangle of data, just mark the data in its buffer in the usual way and type M-# z.

Invocation macros are treated like regular Emacs keyboard macros; all the special features described above for Z K-style macros do not apply. M-# z is just like C-x e, except that it uses the macro that was last stored by Z I. (In fact, the macro does not even have to have anything to do with Calc!)

The m m command saves the last invocation macro defined by Z I along with all the other Calc mode settings. See section General Mode Commands.

Programming with Formulas

Another way to create a new Calculator command uses algebraic formulas. The Z F (`calc-user-define-formula`) command stores the formula at the top of the stack as the definition for a key. This command prompts for five things: The key, the command name, the function name, the argument list, and the behavior of the command when given non-numeric arguments.

For example, suppose we type ' a+2b RET to push the formula `a + 2*b' onto the stack. We now type Z F m to define this formula on the z m key sequence. The next prompt is for a command name, beginning with `calc-', which should be the long (M-x) form for the new command. If you simply press RET, a default name like `calc-User-m` will be constructed. In our example, suppose we enter spam RET to define the new command as `calc-spam`.

If you want to give the formula a long-style name only, you can press SPC or RET when asked which single key to use. For example Z F RET spam RET defines the new command as M-x calc-spam, with no keyboard equivalent.

The third prompt is for a function name. The default is to use the same name as the command name but with `calcFunc-' in place of `calc-'. This is the name you will use if you want to enter your new function in an algebraic formula. Suppose we enter yow RET. Then the new function can be invoked by pushing two numbers on the stack and typing z m or x spam, or by entering the algebraic formula `yow(x,y)'.

The fourth prompt is for the function's argument list. This is used to associate values on the stack with the variables that appear in the formula. The default is a list of all variables which appear in the formula, sorted into alphabetical order. In our case, the default would be `(a b)'. This means that, when the user types z m, the Calculator will remove two numbers from the stack, substitute these numbers for `a' and `b' (respectively) in the formula, then simplify the formula and push the result on the stack. In other words, 10 RET 100 z m would replace the 10 and 100 on the stack with the number 210, which is a + 2 b with a=10 and b=100. Likewise, the formula `yow(10, 100)' will be evaluated by substituting a=10 and b=100 in the definition.

You can rearrange the order of the names before pressing RET to control which stack positions go to which variables in the formula. If you remove a variable from the argument list, that variable will be left in symbolic form by the command. Thus using an argument list of `(b)' for our function would cause 10 z m to replace the 10 on the stack with the formula `a + 20'. If we had used an argument list of `(b a)', the result with inputs 10 and 100 would have been 120.

You can also put a nameless function on the stack instead of just a formula, as in `<a, b : a + 2 b>'. See section Specifying Operators. In this example, the command will be defined by the formula `a + 2 b' using the argument list `(a b)'.

The final prompt is a y-or-n question concerning what to do if symbolic arguments are given to your function. If you answer y, then executing z m (using the original argument list `(a b)') with arguments 10 and x will leave the function in symbolic form, i.e., `yow(10,x)'. On the other hand, if you answer n, then the formula will always be expanded, even for non-constant arguments: `10 + 2 x'. If you never plan to feed algebraic formulas to your new function, it doesn't matter how you answer this question.

If you answered y to this question you can still cause a function call to be expanded by typing a " (`calc-expand-formula`). Also, Calc will expand the function if necessary when you take a derivative or integral or solve an equation involving the function.

Once you have defined a formula on a key, you can retrieve this formula with the Z G

TOC

(`calc-user-define-get-defn`) command. Press a key, and this command pushes the formula that was used to define that key onto the stack. Actually, it pushes a nameless function that specifies both the argument list and the defining formula. You will get an error message if the key is undefined, or if the key was not defined by a Z F command.

The Z E (`calc-user-define-edit`) command on a key that has been defined by a formula uses a variant of the `calc-edit` command to edit the defining formula. Press M-# M-# to finish editing and store the new formula back in the definition, or M-# x to cancel the edit. (The argument list and other properties of the definition are unchanged; to adjust the argument list, you can use Z G to grab the function onto the stack, edit with `, and then re-execute the Z F command.)

As usual, the Z P command records your definition permanently. In this case it will permanently record all three of the relevant definitions: the key, the command, and the function.

You may find it useful to turn off the default simplifications with m O (`calc-no-simplify-mode`) when entering a formula to be used as a function definition. For example, the formula `deriv(a^2,v)' which might be used to define a new function `dsqr(a,v)' will be "simplified" to 0 immediately upon entry since `deriv` considers a to be constant with respect to v. Turning off default simplifications cures this problem: The definition will be stored in symbolic form without ever activating the `deriv` function. Press m D to turn the default simplifications back on afterwards.

Programming with Lisp

The Calculator can be programmed quite extensively in Lisp. All you do is write a normal Lisp function definition, but with `defmath` in place of `defun`. This has the same form as `defun`, but it automagically replaces calls to standard Lisp functions like `+` and `zerop` with calls to the corresponding functions in Calc's own library. Thus you can write natural-looking Lisp code which operates on all of the standard Calculator data types. You can then use Z D if you wish to bind your new command to a z-prefix key sequence. The Z E command will not edit a Lisp-based definition.

Emacs Lisp is described in the GNU Emacs Lisp Reference Manual. This section assumes a familiarity with Lisp programming concepts; if you do not know Lisp, you may find keyboard macros or rewrite rules to be an easier way to program the Calculator.

This section first discusses ways to write commands, functions, or small programs to be executed inside of Calc. Then it discusses how your own separate programs are able to call Calc from the outside. Finally, there is a list of internal Calc functions and data structures for the true Lisp enthusiast.

Defining New Functions

The `defmath` function (actually a Lisp macro) is like `defun` except that code in the body of the definition can make use of the full range of Calculator data types. The prefix `calcFunc-' is added to the specified name to get the actual Lisp function name. As a simple example,

```
(defmath myfact (n)
  (if (> n 0)
      (* n (myfact (1- n)))
    1))
```

This actually expands to the code,


```
(defun calcFunc-myfact (n)
  (if (math-posp n)
      (math-mul n (calcFunc-myfact (math-add n -1)))
    1))
```

This function can be used in algebraic expressions, e.g., `myfact(5)'.

The `myfact' function as it is defined above has the bug that an expression `myfact(a+b)' will be simplified to 1 because the formula `a+b' is not considered to be `posp'. A robust factorial function would be written along the following lines:

```
(defmath myfact (n)
  (if (> n 0)
      (* n (myfact (1- n)))
    (if (= n 0)
        1
      nil)))    ; this could be simplified as: (and (= n 0) 1)
```

If a function returns `nil', it is left unsimplified by the Calculator (except that its arguments will be simplified). Thus, `myfact(a+1+2)' will be simplified to `myfact(a+3)' but no further. Beware that every time the Calculator reexamines this formula it will attempt to resimplify it, so your function ought to detect the returning-`nil' case as efficiently as possible.

The following standard Lisp functions are treated by `defmath': `+', `-', `*', `/', `%', `^' or `expt', `=', `<', `>', `<=', `>=', `/=', `1+', `1-', `logand', `logior', `logxor', `logandc2', `lognot'. Also, `~=' is an abbreviation for `math-nearly-equal', which is useful in implementing Taylor series.

For other functions func, if a function by the name `calcFunc-func' exists it is used, otherwise if a function by the name `math-func' exists it is used, otherwise if func itself is defined as a function it is used, otherwise `calcFunc-func' is used on the assumption that this is a to-be-defined math function. Also, if the function name is quoted as in `('integerp a)' the function name is always used exactly as written (but not quoted).

Variable names have `var-' prepended to them unless they appear in the function's argument list or in an enclosing `let', `let*', `for', or `foreach' form, or their names already contain a `-' character. Thus a reference to `foo' is the same as a reference to `var-foo'.

A few other Lisp extensions are available in `defmath' definitions:

- The `elt' function accepts any number of index variables. Note that Calc vectors are stored as Lisp lists whose first element is the symbol `vec'; thus, `(elt v 2)' yields the second element of vector `v', and `(elt m i j)' yields one element of a Calc matrix.
- The `setq' function has been extended to act like the Common Lisp `setf' function. (The name `setf' is recognized as a synonym of `setq'.) Specifically, the first argument of `setq' can be an `nth', `elt', `car', or `cdr' form, in which case the effect is to store into the specified element of a list. Thus, `(setq (elt m i j) x)' stores x into one element of a matrix.
- A `for' looping construct is available. For example, `(for ((i 0 10)) body)' executes `body' once for each binding of i from zero to 10. This is like a `let' form in that i is temporarily bound to the loop count without disturbing its value outside the `for' construct. Nested loops, as in `(for ((i 0 10) (j 0 (1- i) 2)) body)', are also available. For each value of i from zero to 10, j counts from 0 to i-1 in steps of two. Note that `for' has the same general outline as `let*', except that each element of the header is a list of three or

four things, not just two.

- The `foreach` construct loops over elements of a list. For example, `(foreach ((x (cdr v))) body)' executes `body` with x bound to each element of Calc vector v in turn. The purpose of `cdr` here is to skip over the initial `vec` symbol in the vector.
- The `break` function breaks out of the innermost enclosing `while`, `for`, or `foreach` loop. If given a value, as in `(break x)', this value is returned by the loop. (Lisp loops otherwise always return `nil`.)
- The `return` function prematurely returns from the enclosing function. For example, `(return (+ x y))' returns x+y as the value of a function. You can use `return` anywhere inside the body of the function.

Non-integer numbers (and extremely large integers) cannot be included directly into a `defmath` definition. This is because the Lisp reader will fail to parse them long before `defmath` ever gets control. Instead, use the notation, `:"3.1415"'. In fact, any algebraic formula can go between the quotes. For example,

```
(defmath sqexp (x)     ; sqexp(x) == sqrt(exp(x)) == exp(x*0.5)
  (and (numberp x)
       (exp :"x * 0.5")))
```

expands to

```
(defun calcFunc-sqexp (x)
  (and (math-numberp x)
       (calcFunc-exp (math-mul x '(float 5 -1)))))
```

Note the use of `numberp` as a guard to ensure that the argument is a number first, returning `nil` if not. The exponential function could itself have been included in the expression, if we had preferred: `:"exp(x * 0.5)"'. As another example, the multiplication-and-recursion step of `myfact` could have been written

```
:"n * myfact(n-1)"
```

If a file named `.emacs' exists in your home directory, Emacs reads and executes the Lisp forms in this file as it starts up. While it may seem like a good idea to put your favorite `defmath` commands here, this has the unfortunate side-effect that parts of the Calculator must be loaded in to process the `defmath` commands whether or not you will actually use the Calculator! A better effect can be had by writing

```
(put 'calc-define 'thing '(progn
 (defmath ... )
 (defmath ... )
))
```

The `put` function adds a property to a symbol. Each Lisp symbol has a list of properties associated with it. Here we add a property with a name of `thing` and a `(progn ...)' form as its value. When Calc starts up, and at the start of every Calc command, the property list for the symbol `calc-define` is checked and the values of any properties found are evaluated as Lisp forms. The properties are removed as they are evaluated. The property names (like `thing`) are not used; you should choose something like the name of your project so as not to conflict with other properties.

The net effect is that you can put the above code in your `.emacs' file and it will not be executed until Calc is loaded. Or, you can put that same code in another file which you load by hand either before or after Calc itself is loaded.

The properties of `calc-define` are evaluated in the same order that they were added. They can assume that the Calc modules `` `calc.el' ``, `` `calc-ext.el' ``, and `` `calc-macs.el' `` have been fully loaded, and that the `*Calculator*' buffer will be the current buffer.

If your `calc-define` property only defines algebraic functions, you can be sure that it will have been evaluated before Calc tries to call your function, even if the file defining the property is loaded after Calc is loaded. But if the property defines commands or key sequences, it may not be evaluated soon enough. (Suppose it defines the new command `tweak-calc`; the user can load your file, then type M-x tweak-calc before Calc has had chance to do anything.) To protect against this situation, you can put

```
(run-hooks 'calc-check-defines)
```

at the end of your file. The `calc-check-defines` function is what looks for and evaluates properties on `calc-define`; `run-hooks` has the advantage that it is quietly ignored if `calc-check-defines` is not yet defined because Calc has not yet been loaded.

Examples of things that ought to be enclosed in a `calc-define` property are `defmath` calls, `define-key` calls that modify the Calc key map, and any calls that redefine things defined inside Calc. Ordinary `defun`s need not be enclosed with `calc-define`.

Defining New Simple Commands

If a `defmath` form contains an `interactive` clause, it defines a Calculator command. Actually such a `defmath` results in *two* function definitions: One, a `calcFunc-' function as was just described, with the `interactive` clause removed. Two, a `calc-' function with a suitable `interactive` clause and some sort of wrapper to make the command work in the Calc environment.

In the simple case, the `interactive` clause has the same form as for normal Emacs Lisp commands:

```
(defmath increase-precision (delta)
  "Increase precision by DELTA."     ; This is the "documentation string"
  (interactive "p")                  ; Register this as a M-x-able command
  (setq calc-internal-prec (+ calc-internal-prec delta)))
```

This expands to the pair of definitions,

```
(defun calc-increase-precision (delta)
  "Increase precision by DELTA."
  (interactive "p")
  (calc-wrapper
   (setq calc-internal-prec (math-add calc-internal-prec delta))))

(defun calcFunc-increase-precision (delta)
  "Increase precision by DELTA."
  (setq calc-internal-prec (math-add calc-internal-prec delta)))
```

where in this case the latter function would never really be used! Note that since the Calculator stores small integers as plain Lisp integers, the `math-add` function will work just as well as the native `+` even when the intent is to operate on native Lisp integers.

The `calc-wrapper' call invokes a macro which surrounds the body of the function with code that looks roughly like this:

```
(let ((calc-command-flags nil))
  (unwind-protect
      (save-excursion
        (calc-select-buffer)
        body of function
        renumber stack
        clear Working message)
    realign cursor and window
    clear Inverse, Hyperbolic, and Keep Args flags
    update Emacs mode line))
```

The `calc-select-buffer` function selects the `*Calculator*' buffer if necessary, say, because the command was invoked from inside the `*Calc Trail*' window.

You can call, for example, `(calc-set-command-flag 'no-align)` to set the above-mentioned command flags. The following command flags are recognized by Calc routines:

`renum-stack`

Stack line numbers `1:', `2:', and so on must be renumbered after this command completes. This is set by routines like `calc-push`.

`clear-message`

Calc should call `(message "")' if this command completes normally (to clear a "Working..." message out of the echo area).

`no-align`

Do not move the cursor back to the `.' top-of-stack marker.

`position-point`

Use the variables `calc-position-point-line` and `calc-position-point-column` to position the cursor after this command finishes.

`keep-flags`

Do not clear `calc-inverse-flag`, `calc-hyperbolic-flag`, and `calc-keep-args-flag` at the end of this command.

`do-edit`

Switch to buffer `*Calc Edit*' after this command.

`hold-trail`

Do not move trail pointer to end of trail when something is recorded there.

Calc reserves a special prefix key, shift-Y, for user-written extensions to Calc. There are no built-in commands that work with this prefix key; you must call `define-key` from Lisp (probably from inside a `calc-define` property) to add to it. Initially only Y ? is defined; it takes help messages from a list of strings (initially `nil`) in the variable `calc-Y-help-msgs`. All other undefined keys except for Y are reserved for use by future versions of Calc.

If you are writing a Calc enhancement which you expect to give to others, it is best to minimize the number of Y-key sequences you use. In fact, if you have more than one key sequence you should consider defining

three-key sequences with a Y, then a key that stands for your package, then a third key for the particular command within your package.

Users may wish to install several Calc enhancements, and it is possible that several enhancements will choose to use the same key. In the example below, a variable `inc-prec-base-key` has been defined to contain the key that identifies the `inc-prec` package. Its value is initially `"P"`, but a user can change this variable if necessary without having to modify the file.

Here is a complete file, `` `inc-prec.el' ``, which makes a Y P I command that increases the precision, and a Y P D command that decreases the precision.

```
;;; Increase and decrease Calc precision.  Dave Gillespie, 5/31/91.
;;; (Include copyright or copyleft stuff here.)

(defvar inc-prec-base-key "P"
  "Base key for inc-prec.el commands.")

(put 'calc-define 'inc-prec '(progn

(define-key calc-mode-map (format "Y%sI" inc-prec-base-key)
            'increase-precision)
(define-key calc-mode-map (format "Y%sD" inc-prec-base-key)
            'decrease-precision)

(setq calc-Y-help-msgs
      (cons (format "%s + Inc-prec, Dec-prec" inc-prec-base-key)
            calc-Y-help-msgs))

(defmath increase-precision (delta)
  "Increase precision by DELTA."
  (interactive "p")
  (setq calc-internal-prec (+ calc-internal-prec delta)))

(defmath decrease-precision (delta)
  "Decrease precision by DELTA."
  (interactive "p")
  (setq calc-internal-prec (- calc-internal-prec delta)))

))  ; end of calc-define property

(run-hooks 'calc-check-defines)
```

Defining New Stack-Based Commands

To define a new computational command which takes and/or leaves arguments on the stack, a special form of `interactive` clause is used.

```
(interactive num tag)
```

where num is an integer, and tag is a string. The effect is to pop num values off the stack, resimplify them by calling `calc-normalize`, and hand them to your function according to the function's argument list. Your function may include `&optional` and `&rest` parameters, so long as calling the function with num parameters is legal.

Your function must return either a number or a formula in a form acceptable to Calc, or a list of such numbers or formulas. These value(s) are pushed onto the stack when the function completes. They are also recorded in the Calc Trail buffer on a line beginning with tag, a string of (normally) four characters or less. If you omit tag or use `nil` as a tag, the result is not recorded in the trail.

As an example, the definition

```
(defmath myfact (n)
  "Compute the factorial of the integer at the top of the stack."
  (interactive 1 "fact")
  (if (> n 0)
      (* n (myfact (1- n)))
    (and (= n 0) 1)))
```

is a version of the factorial function shown previously which can be used as a command as well as an algebraic function. It expands to

```
(defun calc-myfact ()
  "Compute the factorial of the integer at the top of the stack."
  (interactive)
  (calc-slow-wrapper
   (calc-enter-result 1 "fact"
     (cons 'calcFunc-myfact (calc-top-list-n 1)))))

(defun calcFunc-myfact (n)
  "Compute the factorial of the integer at the top of the stack."
  (if (math-posp n)
      (math-mul n (calcFunc-myfact (math-add n -1)))
    (and (math-zerop n) 1)))
```

The `calc-slow-wrapper` function is a version of `calc-wrapper` that automatically puts up a `Working...' message before the computation begins. (This message can be turned off by the user with an m w (`calc-working`) command.)

The `calc-top-list-n` function returns a list of the specified number of values from the top of the stack. It resimplifies each value by calling `calc-normalize`. If its argument is zero it returns an empty list. It does not actually remove these values from the stack.

The `calc-enter-result` function takes an integer num and string tag as described above, plus a third argument which is either a Calculator data object or a list of such objects. These objects are resimplified and pushed onto the stack after popping the specified number of values from the stack. If tag is non-`nil`, the values being pushed are also recorded in the trail.

Note that if `calcFunc-myfact` returns `nil` this represents "leave the function in symbolic form." To return an actual empty list, in the sense that `calc-enter-result` will push zero elements back onto the stack, you should return the special value `'(nil)', a list containing the single symbol `nil`.

The `interactive` declaration can actually contain a limited Emacs-style code string as well which comes just before num and tag. Currently the only Emacs code supported is `'"p"', as in

```
(defmath foo (a b &optional c)
  (interactive "p" 2 "foo")
  body)
```

In this example, the command `calc-foo` will evaluate the expression `foo(a,b)' if executed with no argument, or `foo(a,b,n)' if executed with a numeric prefix argument of n.

The other code string allowed is `'"m"' (unrelated to the usual `'"m"' code as used with `defun`). It uses the numeric prefix argument as the number of objects to remove from the stack and pass to the function. In this case, the integer num serves as a default number of arguments to be used when no prefix is supplied.

Argument Qualifiers

Anywhere a parameter name can appear in the parameter list you can also use an argument qualifier. Thus the general form of a definition is:

```
(defmath name (param param...
               &optional param param...
               &rest param)
  body)
```

where each param is either a symbol or a list of the form

```
(qual param)
```

The following qualifiers are recognized:

`complete'

The argument must not be an incomplete vector, interval, or complex number. (This is rarely needed since the Calculator itself will never call your function with an incomplete argument. But there is nothing stopping your own Lisp code from calling your function with an incomplete argument.)

`integer'

The argument must be an integer. If it is an integer-valued float it will be accepted but converted to integer form. Non-integers and formulas are rejected.

`natnum'

Like `integer', but the argument must be non-negative.

`fixnum'

Like `integer', but the argument must fit into a native Lisp integer, which on most systems means less than 2^23 in absolute value. The argument is converted into Lisp-integer form if necessary.

`float'

The argument is converted to floating-point format if it is a number or vector. If it is a formula it is left alone. (The argument is never actually rejected by this qualifier.)

`pred'

The argument must satisfy predicate pred, which is one of the standard Calculator predicates. See se
Predicates.

`not-pred'

The argument must *not* satisfy predicate pred.

For example,

```
(defmath foo (a (constp (not-matrixp b)) &optional (float c)
              &rest (integer d))
  body)
```

expands to

```
(defun calcFunc-foo (a b &optional c &rest d)
  (and (math-matrixp b)
       (math-reject-arg b 'not-matrixp))
  (or (math-constp b)
      (math-reject-arg b 'constp))
  (and c (setq c (math-check-float c)))
  (setq d (mapcar 'math-check-integer d))
  body)
```

which performs the necessary checks and conversions before executing the body of the function.

Example Definitions

This section includes some Lisp programming examples on a larger scale. These programs make use of some of the Calculator's internal functions; see section Calculator Internals.

Bit-Counting

Calc does not include a built-in function for counting the number of "one" bits in a binary integer. It's easy to invent one using b u to convert the integer to a set, and V # to count the elements of that set; let's write a function that counts the bits without having to create an intermediate set.

```
(defmath bcount ((natnum n))
  (interactive 1 "bcnt")
  (let ((count 0))
    (while (> n 0)
      (if (oddp n)
          (setq count (1+ count)))
      (setq n (lsh n -1)))
    count))
```

When this is expanded by `defmath`, it will become the following Emacs Lisp function:

```
(defun calcFunc-bcount (n)
  (setq n (math-check-natnum n))
```


```
  (let ((count 0))
    (while (math-posp n)
      (if (math-oddp n)
          (setq count (math-add count 1)))
      (setq n (calcFunc-lsh n -1)))
    count))
```

If the input numbers are large, this function involves a fair amount of arithmetic. A binary right shift is essentially a division by two; recall that Calc stores integers in decimal form so bit shifts must involve actual division.

To gain a bit more efficiency, we could divide the integer into *n*-bit chunks, each of which can be handled quickly because they fit into Lisp integers. It turns out that Calc's arithmetic routines are especially fast when dividing by an integer less than 1000, so we can set *n = 9* bits and use repeated division by 512:

```
(defmath bcount ((natnum n))
  (interactive 1 "bcnt")
  (let ((count 0))
    (while (not (fixnump n))
      (let ((qr (idivmod n 512)))
        (setq count (+ count (bcount-fixnum (cdr qr)))
              n (car qr))))
    (+ count (bcount-fixnum n))))

(defun bcount-fixnum (n)
  (let ((count 0))
    (while (> n 0)
      (setq count (+ count (logand n 1))
            n (lsh n -1)))
    count))
```

Note that the second function uses `defun`, not `defmath`. Because this function deals only with native Lisp integers ("fixnums"), it can use the actual Emacs `+` and related functions rather than the slower but more general Calc equivalents which `defmath` uses.

The `idivmod` function does an integer division, returning both the quotient and the remainder at once. Again, note that while it might seem that `(logand n 511)' and `(lsh n -9)' are more efficient ways to split off the bottom nine bits of `n`, actually they are less efficient because each operation is really a division by 512 in disguise; `idivmod` allows us to do the same thing with a single division by 512.

The Sine Function

A somewhat limited sine function could be defined as follows, using the well-known Taylor series expansion for @c{$\sin x$} `sin(x)':

```
(defmath mysin ((float (anglep x)))
  (interactive 1 "mysn")
  (setq x (to-radians x))    ; Convert from current angular mode.
  (let ((sum x)              ; Initial term of Taylor expansion of sin.
        newsum
```


```
      (nfact 1)             ; "nfact" equals "n" factorial at all times
      (xnegsqr :"-(x^2)")) ; "xnegsqr" equals -x^2.
  (for ((n 3 100 2))        ; Upper limit of 100 is a good precaution.
    (working "mysin" sum)   ; Display "Working" message, if enabled.
    (setq nfact (* nfact (1- n) n)
          x (* x xnegsqr)
          newsum (+ sum (/ x nfact)))
    (if (~= newsum sum)     ; If newsum is "nearly equal to" sum,
        (break))            ;  then we are done.
    (setq sum newsum))
  sum))
```

The actual `sin` function in Calc works by first reducing the problem to a sine or cosine of a nonnegative number less than @c{$\pi \over 4$} pi/4. This ensures that the Taylor series will converge quickly. Also, the calculation is carried out with two extra digits of precision to guard against cumulative round-off in `sum'. Finally, complex arguments are allowed and handled by a separate algorithm.

```
(defmath mysin ((float (scalarp x)))
  (interactive 1 "mysn")
  (setq x (to-radians x))     ; Convert from current angular mode.
  (with-extra-prec 2          ; Evaluate with extra precision.
    (cond ((complexp x)
           (mysin-complex x))
          ((< x 0)
           (- (mysin-raw (- x)))    ; Always call mysin-raw with x >= 0.
          (t (mysin-raw x))))))

(defmath mysin-raw (x)
  (cond ((>= x 7)
         (mysin-raw (% x (two-pi))))     ; Now x < 7.
        ((> x (pi-over-2))
         (- (mysin-raw (- x (pi)))))     ; Now -pi/2 <= x <= pi/2.
        ((> x (pi-over-4))
         (mycos-raw (- x (pi-over-2))))  ; Now -pi/2 <= x <= pi/4.
        ((< x (- (pi-over-4)))
         (- (mycos-raw (+ x (pi-over-2)))))  ; Now -pi/4 <= x <= pi/4,
        (t (mysin-series x))))           ; so the series will be efficient.
```

where `mysin-complex` is an appropriate function to handle complex numbers, `mysin-series` is the routine to compute the sine Taylor series as before, and `mycos-raw` is a function analogous to `mysin-raw` for cosines.

The strategy is to ensure that x is nonnegative before calling `mysin-raw`. This function then recursively reduces its argument to a suitable range, namely, plus-or-minus @c{$\pi \over 4$} pi/4. Note that each test, and particularly the first comparison against 7, is designed so that small roundoff errors cannnot produce an infinite loop. (Suppose we compared with `(two-pi)' instead; if due to roundoff problems the modulo operator ever returned `(two-pi)' exactly, an infinite recursion could result!) We use modulo only for arguments that will clearly get reduced, knowing that the next rule will catch any reductions that this rule misses.

If a program is being written for general use, it is important to code it carefully as shown in this second

example. For quick-and-dirty programs, when you know that your own use of the sine function will never encounter a large argument, a simpler program like the first one shown is fine.

Calling Calc from Your Lisp Programs

A later section (see section Calculator Internals) gives a full description of Calc's internal Lisp functions. It's not hard to call Calc from inside your programs, but the number of these functions can be daunting. So Calc provides one special "programmer-friendly" function called `calc-eval` that can be made to do just about everything you need. It's not as fast as the low-level Calc functions, but it's much simpler to use!

It may seem that `calc-eval` itself has a daunting number of options, but they all stem from one simple operation.

In its simplest manifestation, `(calc-eval "1+2")' parses the string `"1+2"` as if it were a Calc algebraic entry and returns the result formatted as a string: `"3"`.

Since `calc-eval` is on the list of recommended `autoload` functions, you don't need to make any special preparations to load Calc before calling `calc-eval` the first time. Calc will be loaded and initialized for you.

All the Calc modes that are currently in effect will be used when evaluating the expression and formatting the result.

Additional Arguments to `calc-eval`

If the input string parses to a list of expressions, Calc returns the results separated by `", "`. You can specify a different separator by giving a second string argument to `calc-eval`: `(calc-eval "1+2,3+4" ";")' returns `"3;7"`.

The "separator" can also be any of several Lisp symbols which request other behaviors from `calc-eval`. These are discussed one by one below.

You can give additional arguments to be substituted for `$', `$$', and so on in the main expression. For example, `(calc-eval "$/$$" nil "7" "1+1")' evaluates the expression `"7/(1+1)"` to yield the result `"3.5"` (assuming Fraction mode is not in effect). Note the `nil` used as a placeholder for the item-separator argument.

Error Handling

If `calc-eval` encounters an error, it returns a list containing the character position of the error, plus a suitable message as a string. Note that `1 / 0' is *not* an error by Calc's standards; it simply returns the string `"1 / 0"` which is the division left in symbolic form. But `(calc-eval "1/")' will return the list `(2 "Expected a number")'.

If you bind the variable `calc-eval-error` to `t` using a `let` form surrounding the call to `calc-eval`, errors instead call the Emacs `error` function which aborts to the Emacs command loop with a beep and an error message.

If you bind this variable to the symbol `string`, error messages are returned as strings instead of lists. The character position is ignored.

As a courtesy to other Lisp code which may be using Calc, be sure to bind `calc-eval-error` using `let` rather than changing it permanently with `setq`.

Numbers Only

Sometimes it is preferable to treat `1 / 0' as an error rather than returning a symbolic result. If you pass the symbol `num` as the second argument to `calc-eval`, results that are not constants are treated as errors. The error message reported is the first `calc-why` message if there is one, or otherwise "Number expected."

A result is "constant" if it is a number, vector, or other object that does not include variables or function calls. If it is a vector, the components must themselves be constants.

Default Modes

If the first argument to `calc-eval` is a list whose first element is a formula string, then `calc-eval` sets all the various Calc modes to their default values while the formula is evaluated and formatted. For example, the precision is set to 12 digits, digit grouping is turned off, and the normal language mode is used.

This same principle applies to the other options discussed below. If the first argument would normally be x, then it can also be the list `(x)' to use the default mode settings.

If there are other elements in the list, they are taken as variable-name/value pairs which override the default mode settings. Look at the documentation at the front of the `` `calc.el' `` file to find the names of the Lisp variables for the various modes. The mode settings are restored to their original values when `calc-eval` is done.

For example, `(calc-eval '("$+$$" calc-internal-prec 8) 'num a b)' computes the sum of two numbers, requiring a numeric result, and using default mode settings except that the precision is 8 instead of the default of 12.

It's usually best to use this form of `calc-eval` unless your program actually considers the interaction with Calc's mode settings to be a feature. This will avoid all sorts of potential "gotchas"; consider what happens with `(calc-eval "sqrt(2)" 'num)' when the user has left Calc in symbolic mode or no-simplify mode.

As another example, `(equal (calc-eval '("$<$$") nil a b) "1")' checks if the number in string a is less than the one in string b. Without using a list, the integer 1 might come out in a variety of formats which would be hard to test for conveniently: `"1"`, `"8#1"`, `"00001"`. (But see "Predicates" mode, below.)

Raw Numbers

Normally all input and output for `calc-eval` is done with strings. You can do arithmetic with, say, `(calc-eval "$+$$" nil a b)' in place of `(+ a b)', but this is very inefficient since the numbers must be converted to and from string format as they are passed from one `calc-eval` to the next.

If the separator is the symbol `raw`, the result will be returned as a raw Calc data structure rather than a string. You can read about how these objects look in the following sections, but usually you can treat them as "black box" objects with no important internal structure.

There is also a `rawnum` symbol, which is a combination of `raw` (returning a raw Calc object) and `num` (signalling an error if that object is not a constant).

You can pass a raw Calc object to `calc-eval` in place of a string, either as the formula itself or as one of the `$' arguments. Thus `(calc-eval "$+$$" 'raw a b)' is an addition function that operates on raw Calc objects. Of course in this case it would be easier to call the low-level `math-add` function in Calc, if you can remember its name.

In particular, note that a plain Lisp integer is acceptable to Calc as a raw object. (All Lisp integers are accepted on input, but integers of more than six decimal digits are converted to "big-integer" form for output. See section

Data Type Formats.)

When it comes time to display the object, just use `(calc-eval a)' to format it as a string.

It is an error if the input expression evaluates to a list of values. The separator symbol `list` is like `raw` except that it returns a list of one or more raw Calc objects.

Note that a Lisp string is not a valid Calc object, nor is a list containing a string. Thus you can still safely distinguish all the various kinds of error returns discussed above.

Predicates

If the separator symbol is `pred`, the result of the formula is treated as a true/false value; `calc-eval` returns `t` or `nil`, respectively. A value is considered "true" if it is a non-zero number, or false if it is zero or if it is not a number.

For example, `(calc-eval "$<$$" 'pred a b)' tests whether one value is less than another.

As usual, it is also possible for `calc-eval` to return one of the error indicators described above. Lisp will interpret such an indicator as "true" if you don't check for it explicitly. If you wish to have an error register as "false", use something like `(eq (calc-eval ...) t)'.

Variable Values

Variables in the formula passed to `calc-eval` are not normally replaced by their values. If you wish this, you can use the `evalv` function (see section Algebraic Manipulation). For example, if 4 is stored in Calc variable `a` (i.e., in Lisp variable `var-a`), then `(calc-eval "a+pi")' will return the formula `"a + pi"`, but `(calc-eval "evalv(a+pi)")' will return `"7.14159265359"`.

To store in a Calc variable, just use `setq` to store in the corresponding Lisp variable. (This is obtained by prepending `var-' to the Calc variable name.) Calc routines will understand either string or raw form values stored in variables, although raw data objects are much more efficient. For example, to increment the Calc variable `a`:

```
(setq var-a (calc-eval "evalv(a+1)" 'raw))
```

Stack Access

If the separator symbol is `push`, the formula argument is evaluated (with possible `$' expansions, as usual). The result is pushed onto the Calc stack. The return value is `nil` (unless there is an error from evaluating the formula, in which case the return value depends on `calc-eval-error` in the usual way).

If the separator symbol is `pop`, the first argument to `calc-eval` must be an integer instead of a string. That many values are popped from the stack and thrown away. A negative argument deletes the entry at that stack level. The return value is the number of elements remaining in the stack after popping; `(calc-eval 0 'pop)' is a good way to measure the size of the stack.

If the separator symbol is `top`, the first argument to `calc-eval` must again be an integer. The value at that stack level is formatted as a string and returned. Thus `(calc-eval 1 'top)' returns the top-of-stack value. If the integer is out of range, `nil` is returned.

The separator symbol `rawtop` is just like `top` except that the stack entry is returned as a raw Calc object

instead of as a string.

In all of these cases the first argument can be made a list in order to force the default mode settings, as described above. Thus `(calc-eval '(2 calc-number-radix 16) 'top)' returns the second-to-top stack entry, formatted as a string using the default instead of current display modes, except that the radix is hexadecimal instead of decimal.

It is, of course, polite to put the Calc stack back the way you found it when you are done, unless the user of your program is actually expecting it to affect the stack.

Note that you do not actually have to switch into the `*Calculator*' buffer in order to use `calc-eval`; it temporarily switches into the stack buffer if necessary.

Keyboard Macros

If the separator symbol is `macro`, the first argument must be a string of characters which Calc can execute as a sequence of keystrokes. This switches into the Calc buffer for the duration of the macro. For example, `(calc-eval "vx5\rVR+" 'macro)' pushes the vector `[1,2,3,4,5]' on the stack and then replaces it with the sum of those numbers. Note that `\r' is the Lisp notation for the carriage-return, RET, character.

If your keyboard macro wishes to pop the stack, `\C-d' is safer than `\177' (the DEL character) because some installations may have switched the meanings of DEL and C-h. Calc always interprets C-d as a synonym for "pop-stack" regardless of key mapping.

If you provide a third argument to `calc-eval`, evaluation of the keyboard macro will leave a record in the Trail using that argument as a tag string. Normally the Trail is unaffected.

The return value in this case is always `nil`.

Lisp Evaluation

Finally, if the separator symbol is `eval`, then the Lisp `eval` function is called on the first argument, which must be a Lisp expression rather than a Calc formula. Remember to quote the expression so that it is not evaluated until inside `calc-eval`.

The difference from plain `eval` is that `calc-eval` switches to the Calc buffer before evaluating the expression. For example, `(calc-eval '(setq calc-internal-prec 17) 'eval)' will correctly affect the buffer-local Calc precision variable.

An alternative would be `(calc-eval '(calc-precision 17) 'eval)'. This is evaluating a call to the function that is normally invoked by the p key, giving it 17 as its "numeric prefix argument." Note that this function will leave a message in the echo area as a side effect. Also, all Calc functions switch to the Calc buffer automatically if not invoked from there, so the above call is also equivalent to `(calc-precision 17)' by itself. In all cases, Calc uses `save-excursion` to switch back to your original buffer when it is done.

As usual the first argument can be a list that begins with a Lisp expression to use default instead of current mode settings.

The result of `calc-eval` in this usage is just the result returned by the evaluated Lisp expression.

Example

Here is a sample Emacs command that uses `calc-eval`. Suppose you have a document with lots of refer to temperatures on the Fahrenheit scale, say "98.6 F", and you wish to convert these references to Centigra The following command does this conversion. Place the Emacs cursor right after the letter "F" and invoke th command to change "98.6 F" to "37 C". Or, if the temperature is already in Centigrade form, the command changes it back to Fahrenheit.

```
(defun convert-temp ()
  (interactive)
  (save-excursion
    (re-search-backward "[^-.0-9]\\([-.0-9]+\\) *\\([FC]\\)")
    (let* ((top1 (match-beginning 1))
           (bot1 (match-end 1))
           (number (buffer-substring top1 bot1))
           (top2 (match-beginning 2))
           (bot2 (match-end 2))
           (type (buffer-substring top2 bot2)))
      (if (equal type "F")
          (setq type "C"
                number (calc-eval "($ - 32)*5/9" nil number))
        (setq type "F"
              number (calc-eval "$*9/5 + 32" nil number)))
      (goto-char top2)
      (delete-region top2 bot2)
      (insert-before-markers type)
      (goto-char top1)
      (delete-region top1 bot1)
      (if (string-match "\\.$" number)   ; change "37." to "37"
          (setq number (substring number 0 -1)))
      (insert number))))
```

Note the use of `insert-before-markers` when changing between "F" and "C", so that the character winds up before the cursor instead of after it.

Calculator Internals

This section describes the Lisp functions defined by the Calculator that may be of use to user-written Calculator programs (as described in the rest of this chapter). These functions are shown by their names as they conventionally appear in `defmath`. Their full Lisp names are generally gotten by prepending `calcFunc-' or `math-' to their apparent names. (Names that begin with `calc-' are already in their full Lisp form.) You can use the actual full names instead if you prefer them, or if you are calling these functions from regular Lisp.

The functions described here are scattered throughout the various Calc component files. Note that `` `calc.el' `` includes `autoload`s for only a few component files; when Calc wants to call an advanced function it calls `(calc-extensions)' first; this function autoloads `` `calc-ext.el' ``, which in turn autoloads all the functions in the remaining component files.

Because `defmath` itself uses the extensions, user-written code generally always executes with the extensions already loaded, so normally you can use any Calc function and be confident that it will be autoloaded for you when necessary. If you are doing something special, check carefully to make sure each function you are using is

from `calc.el' or its components, and call `(calc-extensions)' before using any function based in `calc-ext.el' if you can't prove this file will already be loaded.

Data Type Formats

Integers are stored in either of two ways, depending on their magnitude. Integers less than one million in absolute value are stored as standard Lisp integers. This is the only storage format for Calc data objects which is not a Lisp list.

Large integers are stored as lists of the form `(bigpos d0 d1 d2 ...)' for positive integers 1000000 or more, or `(bigneg d0 d1 d2 ...)' for negative integers *-1000000* or less. Each d is a base-1000 "digit," a Lisp integer from 0 to 999. The least significant digit is d0; the last digit, dn, which is always nonzero, is the most significant digit. For example, the integer *-12345678* is stored as `(bigneg 678 345 12)'.

The distinction between small and large integers is entirely hidden from the user. In `defmath` definitions, the Lisp predicate `integerp` returns true for either kind of integer, and in general both big and small integers are accepted anywhere the word "integer" is used in this manual. If the distinction must be made, native Lisp integers are called fixnums and large integers are called bignums.

Fractions are stored as a list of the form, `(frac n d)' where n is an integer (big or small) numerator, d is an integer denominator greater than one, and n and d are relatively prime. Note that fractions where d is one are automatically converted to plain integers by all math routines; fractions where d is negative are normalized by negating the numerator and denominator.

Floating-point numbers are stored in the form, `(float mant exp)', where mant (the "mantissa") is an integer less than `10^p' in absolute value (p represents the current precision), and exp (the "exponent") is a fixnum. The value of the float is `mant * 10^exp'. For example, the number *-3.14* is stored as `(float -314 -2) = -314*10^-2'. Other constraints are that the number 0.0 is always stored as `(float 0 0)', and, except for the 0.0 case, the rightmost base-10 digit of mant is always nonzero. (If the rightmost digit is zero, the number is rearranged by dividing mant by ten and incrementing exp.)

Rectangular complex numbers are stored in the form `(cplx re im)', where re and im are each real numbers, either integers, fractions, or floats. The value is `re + imi'. The im part is nonzero; complex numbers with zero imaginary components are converted to real numbers automatically.

Polar complex numbers are stored in the form `(polar r theta)', where r is a positive real value and theta is a real value or HMS form representing an angle. This angle is usually normalized to lie in the interval `(-180 .. 180)' degrees, or `(-pi .. pi)' radians, according to the current angular mode. If the angle is 0 the value is converted to a real number automatically. (If the angle is 180 degrees, the value is usually also converted to a negative real number.)

Hours-minutes-seconds forms are stored as `(hms h m s)', where h is an integer or an integer-valued float (i.e., a float with `exp >= 0'), m is an integer or integer-valued float in the range `[0 .. 60)', and s is any real number in the range `[0 .. 60)'.

Date forms are stored as `(date n)', where n is a real number that counts days since midnight on the morning of January 1, 1 AD. If n is an integer, this is a pure date form. If n is a fraction or float, this is a date/time form.

Modulo forms are stored as `(mod n m)', where m is a positive real number or HMS form, and n is a real number or HMS form in the range `[0 .. m)'.

Error forms are stored as `(sdev x sigma)', where x is the mean value and sigma is the standard deviation. Each

component is either a number, an HMS form, or a symbolic object (a variable or function call). If sigma is a the value is converted to a plain real number. If sigma is negative or complex, it is automatically normalized to be a positive real.

Interval forms are stored as `(intv mask lo hi)', where mask is one of the integers 0, 1, 2, or 3, and lo and hi are real numbers, HMS forms, or symbolic objects. The mask is a binary integer where 1 represents the fact that the interval is closed on the high end, and 2 represents the fact that it is closed on the low end. (Thus 3 represents a fully closed interval.) The interval `(intv 3 x x)' is converted to the plain number x; intervals `(intv mask x x)' for any other mask represent empty intervals. If hi is less than lo, the interval is converted to a standard empty interval by replacing hi with lo.

Vectors are stored as `(vec v1 v2 ...)', where v1 is the first element of the vector, v2 is the second, and so on. An empty vector is stored as `(vec)'. A matrix is simply a vector where all v's are themselves vectors of equal lengths. Note that Calc vectors are unrelated to the Emacs Lisp "vector" type, which is generally unused by Calc data structures.

Variables are stored as `(var name sym)', where name is a Lisp symbol whose print name is used as the visible name of the variable, and sym is a Lisp symbol in which the variable's value is actually stored. Thus, `(var pi var-pi)' represents the special constant `pi'. Almost always, the form is `(var v var-v)'. If the variable name was entered with # signs (which are converted to hyphens internally), the form is `(var u v)', where u is a symbol whose name contains # characters, and v is a symbol that contains `-' characters instead. The value of a variable is the Calc object stored in its sym symbol's value cell. If the symbol's value cell is void or if it contains `nil', the variable has no value. Special constants have the form `(special-const value)' stored in their value cell, where value is a formula which is evaluated when the constant's value is requested. Variables which represent units are not stored in any special way; they are units only because their names appear in the units table. If the value cell contains a string, it is parsed to get the variable's value when the variable is used.

A Lisp list with any other symbol as the first element is a function call. The symbols `+', `-', `*', `/', `%', `^', and `|' represent special binary operators; these lists are always of the form `(op lhs rhs)' where lhs is the sub-formula on the lefthand side and rhs is the sub-formula on the right. The symbol `neg' represents unary negation; this list is always of the form `(neg arg)'. Any other symbol func represents a function that would be displayed in function-call notation; the symbol func is in general always of the form `calcFunc-name'. The function cell of the symbol func should contain a Lisp function for evaluating a call to func. This function is passed the remaining elements of the list (themselves already evaluated) as arguments; such functions should return `nil' or call `reject-arg' to signify that they should be left in symbolic form, or they should return a Calc object which represents their value, or a list of such objects if they wish to return multiple values. (The latter case is allowed only for functions which are the outer-level call in an expression whose value is about to be pushed on the stack; this feature is considered obsolete and is not used by any built-in Calc functions.)

Interactive Functions

The functions described here are used in implementing interactive Calc commands. Note that this list is not exhaustive! If there is an existing command that behaves similarly to the one you want to define, you may find helpful tricks by checking the source code for that command.

Function: **calc-set-command-flag** *flag*

Set the command flag flag. This is generally a Lisp symbol, but may in fact be anything. The effect is to add flag to the list stored in the variable `calc-command-flags', unless it is already there. See section Defining New Simple Commands.

Function: **calc-clear-command-flag** *flag*

If flag appears among the list of currently-set command flags, remove it from that list.

Function: **calc-record-undo** *rec*

Add the "undo record" rec to the list of steps to take if the current operation should need to be undone. Stack push and pop functions automatically call `calc-record-undo`, so the kinds of undo records you might need to create take the form `(set sym value)', which says that the Lisp variable sym was changed and had previously contained value; `(store var value)' which says that the Calc variable var (a string which is the name of the symbol that contains the variable's value) was stored and its previous value was value (either a Calc data object, or `nil` if the variable was previously void); or `(eval undo redo args ...)', which means that to undo requires calling the function `(undo args ...)' and, if the undo is later redone, calling `(redo args ...)'.

Function: **calc-record-why** *msg args*

Record the error or warning message msg, which is normally a string. This message will be replayed if the user types w (`calc-why`); if the message string begins with a `*', it is considered important enough to display even if the user doesn't type w. If one or more args are present, the displayed message will be of the form, `msg: arg1, arg2, ...', where the arguments are formatted on the assumption that they are either strings or Calc objects of some sort. If msg is a symbol, it is the name of a Calc predicate (such as `integerp` or `numvecp`) which the arguments did not satisfy; it is expanded to a suitable string such as "Expected an integer." The `reject-arg` function calls `calc-record-why` automatically; see section Predicates.

Function: **calc-is-inverse**

This predicate returns true if the current command is inverse, i.e., if the Inverse (I key) flag was set.

Function: **calc-is-hyperbolic**

This predicate is the analogous function for the H key.

Stack-Oriented Functions

The functions described here perform various operations on the Calc stack and trail. They are to be used in interactive Calc commands.

Function: **calc-push-list** *vals n*

Push the Calc objects in list vals onto the stack at stack level n. If n is omitted it defaults to 1, so that the elements are pushed at the top of the stack. If n is greater than 1, the elements will be inserted into the stack so that the last element will end up at level n, the next-to-last at level n+1, etc. The elements of vals are assumed to be valid Calc objects, and are not evaluated, rounded, or renormalized in any way. If vals is an empty list, nothing happens.

The stack elements are pushed without any sub-formula selections. You can give an optional third argument to this function, which must be a list the same size as vals of selections. Each selection must be `eq` to some sub-formula of the corresponding formula in vals, or `nil` if that formula should have no selection.

Function: **calc-top-list** *n m*

Return a list of the n objects starting at level m of the stack. If m is omitted it defaults to 1, so that the elements are taken from the top of the stack. If n is omitted, it also defaults to 1, so that the top stack element (in the form of a one-element list) is returned. If m is greater than 1, the mth stack element will be at the end of the list, the

m+1st element will be next-to-last, etc. If n or m are out of range, the command is aborted with a suitable error message. If n is zero, the function returns an empty list. The stack elements are not evaluated, rounded, or renormalized.

If any stack elements contain selections, and selections have not been disabled by the j e (`calc-enable-selections`) command, this function returns the selected portions rather than the entire stack elements. It can be given a third "selection-mode" argument which selects other behaviors. If it is the symbol `t`, then a selection in any of the requested stack elements produces an "illegal operation on selections" error. If it is the symbol `full`, the whole stack entry is always returned regardless of selections. If it is the symbol `sel`, the selected portion is always returned, or `nil` if there is no selection. (This mode ignores the j e command.) If the symbol is `entry`, the complete stack entry in list form is returned; the first element of this list will be the whole formula, and the third element will be the selection (or `nil`).

Function: **calc-pop-stack** *n m*

Remove the specified elements from the stack. The parameters n and m are defined the same as for `calc-top-list`. The return value of `calc-pop-stack` is uninteresting.

If there are any selected sub-formulas among the popped elements, and j e has not been used to disable selections, this produces an error without changing the stack. If you supply an optional third argument of `t`, the stack elements are popped even if they contain selections.

Function: **calc-record-list** *vals tag*

This function records one or more results in the trail. The vals are a list of strings or Calc objects. The tag is the four-character tag string to identify the values. If tag is omitted, a blank tag will be used.

Function: **calc-normalize** *n*

This function takes a Calc object and "normalizes" it. At the very least this involves re-rounding floating-point values according to the current precision and other similar jobs. Also, unless the user has selected no-simplify mode (see section Simplification Modes), this involves actually evaluating a formula object by executing the function calls it contains, and possibly also doing algebraic simplification, etc.

Function: **calc-top-list-n** *n m*

This function is identical to `calc-top-list`, except that it calls `calc-normalize` on the values that it takes from the stack. They are also passed through `check-complete`, so that incomplete objects will be rejected with an error message. All computational commands should use this in preference to `calc-top-list`; the only standard Calc commands that operate on the stack without normalizing are stack management commands like `calc-enter` and `calc-roll-up`. This function accepts the same optional selection-mode argument as `calc-top-list`.

Function: **calc-top-n** *m*

This function is a convenient form of `calc-top-list-n` in which only a single element of the stack is taken and returned, rather than a list of elements. This also accepts an optional selection-mode argument.

Function: **calc-enter-result** *n tag vals*

This function is a convenient interface to most of the above functions. The vals argument should be either a single Calc object, or a list of Calc objects; the object or objects are normalized, and the top n stack entries are replaced by the normalized objects. If tag is non-`nil`, the normalized objects are also recorded in the trail. A

typical stack-based computational command would take the form,

```
(calc-enter-result n tag (cons 'calcFunc-func
                               (calc-top-list-n n)))
```

If any of the n stack elements replaced contain sub-formula selections, and selections have not been disabled by j e, this function takes one of two courses of action. If n is equal to the number of elements in vals, then each element of vals is spliced into the corresponding selection; this is what happens when you use the TAB key, or when you use a unary arithmetic operation like `sqrt`. If vals has only one element but n is greater than one, there must be only one selection among the top n stack elements; the element from vals is spliced into that selection. This is what happens when you use a binary arithmetic operation like +. Any other combination of n and vals is an error when selections are present.

Function: **calc-unary-op** *tag func arg*

This function implements a unary operator that allows a numeric prefix argument to apply the operator over many stack entries. If the prefix argument arg is `nil`, this uses `calc-enter-result` as outlined above. Otherwise, it maps the function over several stack elements; see section Numeric Prefix Arguments. For example,

```
(defun calc-zeta (arg)
  (interactive "P")
  (calc-unary-op "zeta" 'calcFunc-zeta arg))
```

Function: **calc-binary-op** *tag func arg ident unary*

This function implements a binary operator, analogously to `calc-unary-op`. The optional ident and unary arguments specify the behavior when the prefix argument is zero or one, respectively. If the prefix is zero, the value ident is pushed onto the stack, if specified, otherwise an error message is displayed. If the prefix is one, the unary function unary is applied to the top stack element, or, if unary is not specified, nothing happens. When the argument is two or more, the binary function func is reduced across the top arg stack elements; when the argument is negative, the function is mapped between the next-to-top *-arg* stack elements and the top element.

Function: **calc-stack-size**

Return the number of elements on the stack as an integer. This count does not include elements that have been temporarily hidden by stack truncation; see section Truncating the Stack.

Function: **calc-cursor-stack-index** *n*

Move the point to the nth stack entry. If n is zero, this will be the `.' line. If n is from 1 to the current stack size, this will be the beginning of the first line of that stack entry's display. If line numbers are enabled, this will move to the first character of the line number, not the stack entry itself.

Function: **calc-substack-height** *n*

Return the number of lines between the beginning of the nth stack entry and the bottom of the buffer. If n is zero, this will be one (assuming no stack truncation). If all stack entries are one line long (i.e., no matrices are displayed), the return value will be equal n+1 as long as n is in range. (Note that in Big mode, the return value includes the blank lines that separate stack entries.)

Function: **calc-refresh**

Erase the `*Calculator*` buffer and reformat its contents from memory. This must be called after chang any parameter, such as the current display radix, which might change the appearance of existing stack entri (During a keyboard macro invoked by the X key, refreshing is suppressed, but a flag is set so that the entire stack will be refreshed rather than just the top few elements when the macro finishes.)

Predicates

The functions described here are predicates, that is, they return a true/false value where `nil` means false and anything else means true. These predicates are expanded by `defmath`, for example, from `zerop` to `math-zerop`. In many cases they correspond to native Lisp functions by the same name, but are extended to cover the full range of Calc data types.

Function: **zerop** *x*

Returns true if x is numerically zero, in any of the Calc data types. (Note that for some types, such as error forms and intervals, it never makes sense to return true.) In `defmath`, the expression `(= x 0)' will automatically be converted to `(math-zerop x)', and `(/= x 0)' will be converted to `(not (math-zerop x))'.

Function: **negp** *x*

Returns true if x is negative. This accepts negative real numbers of various types, negative HMS and date forms, and intervals in which all included values are negative. In `defmath`, the expression `(< x 0)' will automatically be converted to `(math-negp x)', and `(>= x 0)' will be converted to `(not (math-negp x))'.

Function: **posp** *x*

Returns true if x is positive (and non-zero). For complex numbers, none of these three predicates will return true.

Function: **looks-negp** *x*

Returns true if x is "negative-looking." This returns true if x is a negative number, or a formula with a leading minus sign such as `-a/b'. In other words, this is an object which can be made simpler by calling `(- x)`.

Function: **integerp** *x*

Returns true if x is an integer of any size.

Function: **fixnump** *x*

Returns true if x is a native Lisp integer.

Function: **natnump** *x*

Returns true if x is a nonnegative integer of any size.

Function: **fixnatnump** *x*

Returns true if x is a nonnegative Lisp integer.

Function: **num-integerp** *x*

Returns true if x is numerically an integer, i.e., either a true integer or a float with no significant digits to the right of the decimal point.

Function: **messy-integerp** *x*

Returns true if x is numerically, but not literally, an integer. A value is `num-integerp` if it is `integerp` | `messy-integerp` (but it is never both at once).

Function: **num-natnump** *x*

Returns true if x is numerically a nonnegative integer.

Function: **evenp** *x*

Returns true if x is an even integer.

Function: **looks-evenp** *x*

Returns true if x is an even integer, or a formula with a leading multiplicative coefficient which is an even integer.

Function: **oddp** *x*

Returns true if x is an odd integer.

Function: **ratp** *x*

Returns true if x is a rational number, i.e., an integer or a fraction.

Function: **realp** *x*

Returns true if x is a real number, i.e., an integer, fraction, or floating-point number.

Function: **anglep** *x*

Returns true if x is a real number or HMS form.

Function: **floatp** *x*

Returns true if x is a float, or a complex number, error form, interval, date form, or modulo form in which at least one component is a float.

Function: **complexp** *x*

Returns true if x is a rectangular or polar complex number (but not a real number).

Function: **rect-complexp** *x*

Returns true if x is a rectangular complex number.

Function: **polar-complexp** *x*

Returns true if x is a polar complex number.

Function: **numberp** *x*

Returns true if x is a real number or a complex number.

Function: **scalarp** *x*

Returns true if x is a real or complex number or an HMS form.

Function: **vectorp** *x*

Returns true if x is a vector (this simply checks if its argument is a list whose first element is the symbol `ve

Function: **numvecp** *x*

Returns true if x is a number or vector.

Function: **matrixp** *x*

Returns true if x is a matrix, i.e., a vector of one or more vectors, all of the same size.

Function: **square-matrixp** *x*

Returns true if x is a square matrix.

Function: **objectp** *x*

Returns true if x is any numeric Calc object, including real and complex numbers, HMS forms, date forms, error forms, intervals, and modulo forms. (Note that error forms and intervals may include formulas as their components; see `constp` below.)

Function: **objvecp** *x*

Returns true if x is an object or a vector. This also accepts incomplete objects, but it rejects variables and formulas (except as mentioned above for `objectp`).

Function: **primp** *x*

Returns true if x is a "primitive" or "atomic" Calc object, i.e., one whose components cannot be regarded as sub-formulas. This includes variables, and all `objectp` types except error forms and intervals.

Function: **constp** *x*

Returns true if x is constant, i.e., a real or complex number, HMS form, date form, or error form, interval, or vector all of whose components are `constp`.

Function: **lessp** *x y*

Returns true if x is numerically less than y. Returns false if x is greater than or equal to y, or if the order is undefined or cannot be determined. Generally speaking, this works by checking whether `x - y' is `negp`. In `defmath`, the expression `(< x y)' will automatically be converted to `(lessp x y)'; expressions involving `>`, `<=`, and `>=` are similarly converted in terms of `lessp`.

Function: **beforep** *x y*

Returns true if x comes before y in a canonical ordering of Calc objects. If x and y are both real numbers, this will be the same as `lessp`. But whereas `lessp` considers other types of objects to be unordered, `beforep` puts any two objects into a definite, consistent order. The `beforep` function is used by the V S vector-sorting command, and also by a s to put the terms of a product into canonical order: This allows `x y + y x' to be simplified easily to `2 x y'.

Function: **equal** *x y*

This is the standard Lisp `equal` predicate; it returns true if x and y are structurally identical. This is the usual way to compare numbers for equality, but note that `equal` will treat 0 and 0.0 as different.

Function: **math-equal** *x y*

Returns true if x and y are numerically equal, either because they are `equal`, or because their difference is `zerop`. In `defmath`, the expression `(= x y)' will automatically be converted to `(math-equal x y)'.

Function: **equal-int** *x n*

Returns true if x and n are numerically equal, where n is a fixnum which is not a multiple of 10. This will automatically be used by `defmath` in place of the more general `math-equal` whenever possible.

Function: **nearly-equal** *x y*

Returns true if x and y, as floating-point numbers, are equal except possibly in the last decimal place. For example, 314.159 and 314.166 are considered nearly equal if the current precision is 6 (since they differ by 7 units), but not if the current precision is 7 (since they differ by 70 units). Most functions which use series expansions use `with-extra-prec` to evaluate the series with 2 extra digits of precision, then use `nearly-equal` to decide when the series has converged; this guards against cumulative error in the series evaluation without doing extra work which would be lost when the result is rounded back down to the current precision. In `defmath`, this can be written `(~= x y)'. The x and y can be numbers of any kind, including complex.

Function: **nearly-zerop** *x y*

Returns true if x is nearly zero, compared to y. This checks whether x plus y would by be `nearly-equal` to y itself, to within the current precision, in other words, if adding x to y would have a negligible effect on y due to roundoff error. X may be a real or complex number, but y must be real.

Function: **is-true** *x*

Return true if the formula x represents a true value in Calc, not Lisp, terms. It tests if x is a non-zero number or a provably non-zero formula.

Function: **reject-arg** *val pred*

Abort the current function evaluation due to unacceptable argument values. This calls `(calc-record-why pred val)', then signals a Lisp error which `normalize` will trap. The net effect is that the function call which led here will be left in symbolic form.

Function: **inexact-value**

If Symbolic Mode is enabled, this will signal an error that causes `normalize` to leave the formula in symbolic form, with the message "Inexact result." (This function has no effect when not in Symbolic Mode.) Note that if your function calls `(sin 5)' in Symbolic Mode, the `sin` function will call `inexact-value`, which will cause your function to be left unsimplified. You may instead wish to call `(normalize (list 'calcFunc-sin 5))', which in Symbolic Mode will return the formula `sin(5)' to your function.

Function: **overflow**

This signals an error that will be reported as a floating-point overflow.

Function: **underflow**

This signals a floating-point underflow.

Computational Functions

The functions described here do the actual computational work of the Calculator. In addition to these, note any function described in the main body of this manual may be called from Lisp; for example, if the documentation refers to the `calc-sqrt` [`sqrt`] command, this means `calc-sqrt` is an interactive stack-based square-root command and `sqrt` (which `defmath` expands to `calcFunc-sqrt`) is the actual Lisp function for taking square roots.

The functions `math-add`, `math-sub`, `math-mul`, `math-div`, `math-mod`, and `math-neg` are not included in this list, since `defmath` allows you to write native Lisp `+`, `-`, `*`, `/`, `%`, and unary `-`, respectively, instead.

Function: **normalize** *val*

(Full form: `math-normalize`.) Reduce the value val to standard form. For example, if val is a fixnum, it will be converted to a bignum if it is too large, and if val is a bignum it will be normalized by clipping off trailing (i.e., most-significant) zero digits and converting to a fixnum if it is small. All the various data types are similarly converted to their standard forms. Variables are left alone, but function calls are actually evaluated in formulas. For example, normalizing `(+ 2 (calcFunc-abs -4))' will return 6.

If a function call fails, because the function is void or has the wrong number of parameters, or because it returns `nil` or calls `reject-arg` or `inexact-result`, `normalize` returns the formula still in symbolic form.

If the current Simplification Mode is "none" or "numeric arguments only," `normalize` will act appropriately. However, the more powerful simplification modes (like algebraic simplification) are not handled by `normalize`. They are handled by `calc-normalize`, which calls `normalize` and possibly some other routines, such as `simplify` or `simplify-units`. Programs generally will never call `calc-normalize` except when popping or pushing values on the stack.

Function: **evaluate-expr** *expr*

Replace all variables in expr that have values with their values, then use `normalize` to simplify the result. This is what happens when you press the = key interactively.

Macro: **with-extra-prec** *n body*

Evaluate the Lisp forms in body with precision increased by n digits. This is a macro which expands to

```
(math-normalize
  (let ((calc-internal-prec (+ calc-internal-prec n)))
    body))
```

The surrounding call to `math-normalize` causes a floating-point result to be rounded down to the original precision afterwards. This is important because some arithmetic operations assume a number's mantissa contains no more digits than the current precision allows.

Function: **make-frac** *n d*

Build a fraction `n:d'. This is equivalent to calling `(normalize (list 'frac n d))', but more efficient.

Function: **make-float** *mant exp*

Build a floating-point value out of mant and exp, both of which are arbitrary integers. This function will return a properly normalized float value, or signal an overflow or underflow if exp is out of range.

Function: **make-sdev** *x sigma*

Build an error form out of x and the absolute value of sigma. If sigma is zero, the result is the number x di
If sigma is negative or complex, its absolute value is used. If x or sigma is not a valid type of object for use in
error forms, this calls `reject-arg`.

Function: **make-intv** *mask lo hi*

Build an interval form out of mask (which is assumed to be an integer from 0 to 3), and the limits lo and hi. If lo is greater than hi, an empty interval form is returned. This calls `reject-arg` if lo or hi is unsuitable.

Function: **sort-intv** *mask lo hi*

Build an interval form, similar to `make-intv`, except that if lo is less than hi they are simply exchanged, and the bits of mask are swapped accordingly.

Function: **make-mod** *n m*

Build a modulo form out of n and the modulus m. Since modulo forms do not allow formulas as their components, if n or m is not a real number or HMS form the result will be a formula which is a call to `makemod`, the algebraic version of this function.

Function: **float** *x*

Convert x to floating-point form. Integers and fractions are converted to numerically equivalent floats; components of complex numbers, vectors, HMS forms, date forms, error forms, intervals, and modulo forms are recursively floated. If the argument is a variable or formula, this calls `reject-arg`.

Function: **compare** *x y*

Compare the numbers x and y, and return *-1* if `(lessp x y)', 1 if `(lessp y x)', 0 if `(math-equal x y)', or 2 if the order is undefined or cannot be determined.

Function: **numdigs** *n*

Return the number of digits of integer n, effectively `ceil(log10(n))', but much more efficient. Zero is considered to have zero digits.

Function: **scale-int** *x n*

Shift integer x left n decimal digits, or right *-n* digits with truncation toward zero.

Function: **scale-rounding** *x n*

Like `scale-int`, except that a right shift rounds to the nearest integer rather than truncating.

Function: **fixnum** *n*

Return the integer n as a fixnum, i.e., a native Lisp integer. If n is outside the permissible range for Lisp integers (usually 24 binary bits) the result is undefined.

Function: **sqr** *x*

Compute the square of x; short for `(* x x)'.

Function: **quotient** *x y*

Divide integer x by integer y; return an integer quotient and discard the remainder. If x or y is negative, the

direction of rounding is undefined.

Function: **idiv** *x y*

Perform an integer division; if x and y are both nonnegative integers, this uses the `quotient` function, otherwise it computes `floor(x/y)'. Thus the result is well-defined but slower than for `quotient`.

Function: **imod** *x y*

Divide integer x by integer y; return the integer remainder and discard the quotient. Like `quotient`, this works only for integer arguments and is not well-defined for negative arguments. For a more well-defined result, use `(% x y)'.

Function: **idivmod** *x y*

Divide integer x by integer y; return a cons cell whose `car` is `(quotient x y)' and whose `cdr` is `(imod x y)'.

Function: **pow** *x y*

Compute x to the power y. In `defmath` code, this can also be written `(^ x y)' or `(expt x y)'.

Function: **abs-approx** *x*

Compute a fast approximation to the absolute value of x. For example, for a rectangular complex number the result is the sum of the absolute values of the components.

Function: **pi**

The function `(pi)' computes `pi' to the current precision. Other related constant-generating functions are `two-pi`, `pi-over-2`, `pi-over-4`, `pi-over-180`, `sqrt-two-pi`, `e`, `sqrt-e`, `ln-2`, and `ln-10`. Each function returns a floating-point value in the current precision, and each uses caching so that all calls after the first are essentially free.

Macro: **math-defcache** *func initial form*

This macro, usually used as a top-level call like `defun` or `defvar`, defines a new cached constant analogous to `pi`, etc. It defines a function `func` which returns the requested value; if initial is non-`nil` it must be a `(float ...)' form which serves as an initial value for the cache. If func is called when the cache is empty or does not have enough digits to satisfy the current precision, the Lisp expression form is evaluated with the current precision increased by four, and the result minus its two least significant digits is stored in the cache. For example, calling `(pi)' with a precision of 30 computes `pi' to 34 digits, rounds it down to 32 digits for future use, then rounds it again to 30 digits for use in the present request.

Function: **full-circle** *symb*

If the current angular mode is Degrees or HMS, this function returns the integer 360. In Radians mode, this function returns either the corresponding value in radians to the current precision, or the formula `2*pi', depending on the Symbolic Mode. There are also similar function `half-circle` and `quarter-circle`.

Function: **power-of-2** *n*

Compute two to the integer power n, as a (potentially very large) integer. Powers of two are cached, so only the first call for a particular n is expensive.

Function: **integer-log2** *n*

Compute the base-2 logarithm of n, which must be an integer which is a power of two. If n is not a power of two, this function will return `nil`.

Function: **div-mod** *a b m*

Divide a by b, modulo m. This returns `nil` if there is no solution, or if any of the arguments are not integers.

Function: **pow-mod** *a b m*

Compute a to the power b, modulo m. If a, b, and m are integers, this uses an especially efficient algorithm. Otherwise, it simply computes `(% (^ a b) m)'.

Function: **isqrt** *n*

Compute the integer square root of n. This is the square root of n rounded down toward zero, i.e., `floor(sqrt(n))'. If n is itself an integer, the computation is especially efficient.

Function: **to-hms** *a ang*

Convert the argument a into an HMS form. If ang is specified, it is the angular mode in which to interpret a, either `deg` or `rad`. Otherwise, the current angular mode is used. If a is already an HMS form it is returned as-is.

Function: **from-hms** *a ang*

Convert the HMS form a into a real number. If ang is specified, it is the angular mode in which to express the result, otherwise the current angular mode is used. If a is already a real number, it is returned as-is.

Function: **to-radians** *a*

Convert the number or HMS form a to radians from the current angular mode.

Function: **from-radians** *a*

Convert the number a from radians to the current angular mode. If a is a formula, this returns the formula `deg(a)'.

Function: **to-radians-2** *a*

Like `to-radians`, except that in Symbolic Mode a degrees to radians conversion yields a formula like `a*pi/180'.

Function: **from-radians-2** *a*

Like `from-radians`, except that in Symbolic Mode a radians to degrees conversion yields a formula like `a*180/pi'.

Function: **random-digit**

Produce a random base-1000 digit in the range 0 to 999.

Function: **random-digits** *n*

Produce a random n-digit integer; this will be an integer in the interval `[0, 10^n)'.

Function: **random-float**

Produce a random float in the interval `[0, 1)'.

Function: **prime-test** *n iters*

Determine whether the integer n is prime. Return a list which has one of these forms: `(nil f)' means the number is non-prime because it was found to be divisible by f; `(nil)' means it was found to be non-prime by table look-up (so no factors are known); `(nil unknown)' means it is definitely non-prime but no factors are known because n was large enough that Fermat's probabilistic test had to be used; `(t)' means the number is definitely prime; and `(maybe i p)' means that Fermat's test, after i iterations, is p percent sure that the number is prime. The iters parameter is the number of Fermat iterations to use, in the case that this is necessary. If `prime-test` returns "maybe," you can call it again with the same n to get a greater certainty; `prime-test` remembers where it left off.

Function: **to-simple-fraction** *f*

If f is a floating-point number which can be represented exactly as a small rational number. return that number, else return f. For example, 0.75 would be converted to 3:4. This function is very fast.

Function: **to-fraction** *f tol*

Find a rational approximation to floating-point number f to within a specified tolerance tol; this corresponds to the algebraic function `frac`, and can be rather slow.

Function: **quarter-integer** *n*

If n is an integer or integer-valued float, this function returns zero. If n is a half-integer (i.e., an integer plus *1:2* or 0.5), it returns 2. If n is a quarter-integer, it returns 1 or 3. If n is anything else, this function returns `nil`.

Vector Functions

The functions described here perform various operations on vectors and matrices.

Function: **math-concat** *x y*

Do a vector concatenation; this operation is written `x | y' in a symbolic formula. See section Building Vectors.

Function: **vec-length** *v*

Return the length of vector v. If v is not a vector, the result is zero. If v is a matrix, this returns the number of rows in the matrix.

Function: **mat-dimens** *m*

Determine the dimensions of vector or matrix m. If m is not a vector, the result is an empty list. If m is a plain vector but not a matrix, the result is a one-element list containing the length of the vector. If m is a matrix with r rows and c columns, the result is the list `(r c)'. Higher-order tensors produce lists of more than two dimensions. Note that the object `[[1, 2, 3], [4, 5]]' is a vector of vectors not all the same size, and is treated by this and other Calc routines as a plain vector of two elements.

Function: **dimension-error**

Abort the current function with a message of "Dimension error." The Calculator will leave the function being evaluated in symbolic form; this is really just a special case of `reject-arg`.

Function: **build-vector** *args*

Return a Calc vector with the zero-or-more args as elements. For example, `(build-vector 1 2 3)' returns the Calc vector `[1, 2, 3]', stored internally as the list `(vec 1 2 3)'.

Function: **make-vec** *obj dims*

Return a Calc vector or matrix all of whose elements are equal to obj. For example, `(make-vec 27 3 4)' returns a 3x4 matrix filled with 27's.

Function: **row-matrix** *v*

If v is a plain vector, convert it into a row matrix, i.e., a matrix whose single row is v. If v is already a matrix, leave it alone.

Function: **col-matrix** *v*

If v is a plain vector, convert it into a column matrix, i.e., a matrix with each element of v as a separate row. If v is already a matrix, leave it alone.

Function: **map-vec** *f v*

Map the Lisp function f over the Calc vector v. For example, `(map-vec 'math-floor v)' returns a vector of the floored components of vector v.

Function: **map-vec-2** *f a b*

Map the Lisp function f over the two vectors a and b. If a and b are vectors of equal length, the result is a vector of the results of calling `(f ai bi)' for each pair of elements ai and bi. If either a or b is a scalar, it is matched with each value of the other vector. For example, `(map-vec-2 'math-add v 1)' returns the vector v with each element increased by one. Note that using `'+' would not work here, since `defmath` does not expand function names everywhere, just where they are in the function position of a Lisp expression.

Function: **reduce-vec** *f v*

Reduce the function f over the vector v. For example, if v is `[10, 20, 30, 40]', this calls `(f (f (f 10 20) 30) 40)'. If v is a matrix, this reduces over the rows of v.

Function: **reduce-cols** *f m*

Reduce the function f over the columns of matrix m. For example, if m is `[[1, 2], [3, 4], [5, 6]]', the result is a vector of the two elements `(f (f 1 3) 5)' and `(f (f 2 4) 6)'.

Function: **mat-row** *m n*

Return the nth row of matrix m. This is equivalent to `(elt m n)'. For a slower but safer version, use `mrow`. (See section Extracting Vector Elements.)

Function: **mat-col** *m n*

Return the nth column of matrix m, in the form of a vector. The arguments are not checked for correctness.

Function: **mat-less-row** *m n*

Return a copy of matrix m with its nth row deleted. The number n must be in range from 1 to the number of rows in m.

Function: **mat-less-col** *m n*

Return a copy of matrix m with its nth column deleted.

Function: **transpose** *m*

Return the transpose of matrix m.

Function: **flatten-vector** *v*

Flatten nested vector v into a vector of scalars. For example, if v is `[[1, 2, 3], [4, 5]]' the result is `[1, 2, 3, 4, 5]'.

Function: **copy-matrix** *m*

If m is a matrix, return a copy of m. This maps `copy-sequence` over the rows of m; in Lisp terms, each element of the result matrix will be `eq` to the corresponding element of m, but none of the `cons` cells that make up the structure of the matrix will be `eq`. If m is a plain vector, this is the same as `copy-sequence`.

Function: **swap-rows** *m r1 r2*

Exchange rows r1 and r2 of matrix m in-place. In other words, unlike most of the other functions described here, this function changes m itself rather than building up a new result matrix. The return value is m, i.e., `(eq (swap-rows m 1 2) m)' is true, with the side effect of exchanging the first two rows of m.

Symbolic Functions

The functions described here operate on symbolic formulas in the Calculator.

Function: **calc-prepare-selection** *num*

Prepare a stack entry for selection operations. If num is omitted, the stack entry containing the cursor is used; otherwise, it is the number of the stack entry to use. This function stores useful information about the current stack entry into a set of variables. `calc-selection-cache-num` contains the number of the stack entry involved (equal to num if you specified it); `calc-selection-cache-entry` contains the stack entry as a list (such as `calc-top-list` would return with `entry` as the selection mode); and `calc-selection-cache-comp` contains a special "tagged" composition (see section I/O and Formatting Functions) which allows Calc to relate cursor positions in the buffer with their corresponding sub-formulas.

A slight complication arises in the selection mechanism because formulas may contain small integers. For example, in the vector `[1, 2, 1]' the first and last elements are `eq` to each other; selections are recorded as the actual Lisp object that appears somewhere in the tree of the whole formula, but storing `1` would falsely select both `1`'s in the vector. So `calc-prepare-selection` also checks the stack entry and replaces any plain integers with "complex number" lists of the form `(cplx n 0)'. This list will be displayed the same as a plain n and the change will be completely invisible to the user, but it will guarantee that no two sub-formulas of the stack entry will be `eq` to each other. Next time the stack entry is involved in a computation, `calc-normalize` will replace these lists with plain numbers again, again invisibly to the user.

Function: **calc-encase-atoms** *x*

This modifies the formula x to ensure that each part of the formula is a unique atom, using the `(cplx n 0)' trick described above. This function may use `setcar` to modify the formula in-place.

Function: **calc-find-selected-part**

Find the smallest sub-formula of the current formula that contains the cursor. This assumes

`calc-prepare-selection` has been called already. If the cursor is not actually on any part of the fo this returns `nil`.

Function: **calc-change-current-selection** *selection*

Change the currently prepared stack element's selection to selection, which should be `eq` to some sub-formula of the stack element, or `nil` to unselect the formula. The stack element's appearance in the Calc buffer is adjusted to reflect the new selection.

Function: **calc-find-nth-part** *expr n*

Return the nth sub-formula of expr. This function is used by the selection commands, and (unless j b has been used) treats sums and products as flat many-element formulas. Thus if expr is `((a + b) - c) + d', calling `calc-find-nth-part` with n equal to four will return `d'.

Function: **calc-find-parent-formula** *expr part*

Return the sub-formula of expr which immediately contains part. If expr is `a*b + (c+1)*d' and part is `eq` to the `c+1' term of expr, then this function will return `(c+1)*d'. If part turns out not to be a sub-formula of expr, the function returns `nil`. If part is `eq` to expr, the function returns `t`. This function does not take associativity into account.

Function: **calc-find-assoc-parent-formula** *expr part*

This is the same as `calc-find-parent-formula`, except that (unless j b has been used) it continues widening the selection to contain a complete level of the formula. Given `a' from `((a + b) - c) + d', `calc-find-parent-formula` will return `a + b' but `calc-find-assoc-parent-formula` will return the whole expression.

Function: **calc-grow-assoc-formula** *expr part*

This expands sub-formula part of expr to encompass a complete level of the formula. If part and its immediate parent are not compatible associative operators, or if j b has been used, this simply returns part.

Function: **calc-find-sub-formula** *expr part*

This finds the immediate sub-formula of expr which contains part. It returns an index n such that `(calc-find-nth-part expr n)' would return part. If part is not a sub-formula of expr, it returns `nil`. If part is `eq` to expr, it returns `t`. This function does not take associativity into account.

Function: **calc-replace-sub-formula** *expr old new*

This function returns a copy of formula expr, with the sub-formula that is `eq` to old replaced by new.

Function: **simplify** *expr*

Simplify the expression expr by applying various algebraic rules. This is what the a s (`calc-simplify`) command uses. This always returns a copy of the expression; the structure expr points to remains unchanged in memory.

More precisely, here is what `simplify` does: The expression is first normalized and evaluated by calling `normalize`. If any `AlgSimpRules` have been defined, they are then applied. Then the expression is traversed in a depth-first, bottom-up fashion; at each level, any simplifications that can be made are made until no further changes are possible. Once the entire formula has been traversed in this way, it is compared with the original formula (from before the call to `normalize`) and, if it has changed, the entire procedure is repeated

(starting with `normalize`) until no further changes occur. Usually only two iterations are needed: one to simplify the formula, and another to verify that no further simplifications were possible.

Function: **simplify-extended** *expr*

Simplify the expression expr, with additional rules enabled that help do a more thorough job, while not being entirely "safe" in all circumstances. (For example, this mode will simplify `sqrt(x^2)' to `x', which is only valid when x is positive.) This is implemented by temporarily binding the variable `math-living-dangerously` to `t` (using a `let` form) and calling `simplify`. Dangerous simplification rules are written to check this variable before taking any action.

Function: **simplify-units** *expr*

Simplify the expression expr, treating variable names as units whenever possible. This works by binding the variable `math-simplifying-units` to `t` while calling `simplify`.

Macro: **math-defsimplify** *funcs body*

Register a new simplification rule; this is normally called as a top-level form, like `defun` or `defmath`. If funcs is a symbol (like `+` or `calcFunc-sqrt`), this simplification rule is applied to the formulas which are calls to the specified function. Or, funcs can be a list of such symbols; the rule applies to all functions on the list. The body is written like the body of a function with a single argument called `expr`. The body will be executed with `expr` bound to a formula which is a call to one of the functions funcs. If the function body returns `nil`, or if it returns a result `equal` to the original `expr`, it is ignored and Calc goes on to try the next simplification rule that applies. If the function body returns something different, that new formula is substituted for expr in the original formula.

At each point in the formula, rules are tried in the order of the original calls to `math-defsimplify`; the search stops after the first rule that makes a change. Thus later rules for that same function will not have a chance to trigger until the next iteration of the main `simplify` loop.

Note that, since `defmath` is not being used here, body must be written in true Lisp code without the conveniences that `defmath` provides. If you prefer, you can have body simply call another function (defined with `defmath`) which does the real work.

The arguments of a function call will already have been simplified before any rules for the call itself are invoked. Since a new argument list is consed up when this happens, this means that the rule's body is allowed to rearrange the function's arguments destructively if that is convenient. Here is a typical example of a simplification rule:

```
(math-defsimplify calcFunc-arcsinh
  (or (and (math-looks-negp (nth 1 expr))
           (math-neg (list 'calcFunc-arcsinh
                           (math-neg (nth 1 expr)))))
      (and (eq (car-safe (nth 1 expr)) 'calcFunc-sinh)
           (or math-living-dangerously
               (math-known-realp (nth 1 (nth 1 expr))))
           (nth 1 (nth 1 expr)))))
```

This is really a pair of rules written with one `math-defsimplify` for convenience; the first replaces `arcsinh(-x)' with `-arcsinh(x)', and the second, which is safe only for real `x', replaces `arcsinh(sinh(x))' with `x'.

Function: **common-constant-factor** *expr*

Check expr to see if it is a sum of terms all multiplied by the same rational value. If so, return this value. If not, return `nil`. For example, if called on `6x + 9y + 12z', it would return 3, since 3 is a common factor of all the terms.

Function: **cancel-common-factor** *expr factor*

Assuming expr is a sum with factor as a common factor, divide each term of the sum by factor. This is done by destructively modifying parts of expr, on the assumption that it is being used by a simplification rule (where such things are allowed; see above). For example, consider this built-in rule for square roots:

```
(math-defsimplify calcFunc-sqrt
  (let ((fac (math-common-constant-factor (nth 1 expr))))
    (and fac (not (eq fac 1))
         (math-mul (math-normalize (list 'calcFunc-sqrt fac))
                   (math-normalize
                    (list 'calcFunc-sqrt
                          (math-cancel-common-factor
                           (nth 1 expr) fac)))))))
```

Function: **frac-gcd** *a b*

Compute a "rational GCD" of a and b, which must both be rational numbers. This is the fraction composed of the GCD of the numerators of a and b, over the GCD of the denominators. It is used by `common-constant-factor`. Note that the standard `gcd` function uses the LCM to combine the denominators.

Function: **map-tree** *func expr many*

Try applying Lisp function func to various sub-expressions of expr. Initially, call func with expr itself as an argument. If this returns an expression which is not `equal` to expr, apply func again until eventually it does return expr with no changes. Then, if expr is a function call, recursively apply func to each of the arguments. This keeps going until no changes occur anywhere in the expression; this final expression is returned by `map-tree`. Note that, unlike simplification rules, func functions may *not* make destructive changes to expr. If a third argument many is provided, it is an integer which says how many times func may be applied; the default, as described above, is infinitely many times.

Function: **compile-rewrites** *rules*

Compile the rewrite rule set specified by rules, which should be a formula that is either a vector or a variable name. If the latter, the compiled rules are saved so that later `compile-rules` calls for that same variable can return immediately. If there are problems with the rules, this function calls `error` with a suitable message.

Function: **apply-rewrites** *expr crules heads*

Apply the compiled rewrite rule set crules to the expression expr. This will make only one rewrite and only checks at the top level of the expression. The result `nil` if no rules matched, or if the only rules that matched did not actually change the expression. The heads argument is optional; if is given, it should be a list of all function names that (may) appear in expr. The rewrite compiler tags each rule with the rarest-looking function name in the rule; if you specify heads, `apply-rewrites` can use this information to narrow its search down to just a few rules in the rule set.

Function: **rewrite-heads** *expr*

Compute a heads list for expr suitable for use with `apply-rewrites`, as discussed above.

Function: **rewrite** *expr rules many*

This is an all-in-one rewrite function. It compiles the rule set specified by rules, then uses `map-tree` to apply the rules throughout expr up to many (default infinity) times.

Function: **match-patterns** *pat vec not-flag*

Given a Calc vector vec and an uncompiled pattern set or pattern set variable pat, this function returns a new vector of all elements of vec which do (or don't, if not-flag is non-`nil`) match any of the patterns in pat.

Function: **deriv** *expr var value symb*

Compute the derivative of expr with respect to variable var (which may actually be any sub-expression). If value is specified, the derivative is evaluated at the value of var; otherwise, the derivative is left in terms of var. If the expression contains functions for which no derivative formula is known, new derivative functions are invented by adding primes to the names; see section Calculus. However, if symb is non-`nil`, the presence of undifferentiable functions in expr instead cancels the whole differentiation, and `deriv` returns `nil` instead.

Derivatives of an n-argument function can be defined by adding a `math-derivative-n` property to the property list of the symbol for the function's derivative, which will be the function name followed by an apostrophe. The value of the property should be a Lisp function; it is called with the same arguments as the original function call that is being differentiated. It should return a formula for the derivative. For example, the derivative of `ln` is defined by

```
(put 'calcFunc-ln\' 'math-derivative-1
     (function (lambda (u) (math-div 1 u))))
```

The two-argument `log` function has two derivatives,

```
(put 'calcFunc-log\' 'math-derivative-2     ; d(log(x,b)) / dx
     (function (lambda (x b) ... )))
(put 'calcFunc-log\'2 'math-derivative-2    ; d(log(x,b)) / db
     (function (lambda (x b) ... )))
```

Function: **tderiv** *expr var value symb*

Compute the total derivative of expr. This is the same as `deriv`, except that variables other than var are not assumed to be constant with respect to var.

Function: **integ** *expr var low high*

Compute the integral of expr with respect to var. See section Calculus, for further details.

Macro: **math-defintegral** *funcs body*

Define a rule for integrating a function or functions of one argument; this macro is very similar in format to `math-defsimplify`. The main difference is that here body is the body of a function with a single argument `u` which is bound to the argument to the function being integrated, not the function call itself. Also, the variable of integration is available as `math-integ-var`. If evaluation of the integral requires doing further integrals,

the body should call `(math-integral x)' to find the integral of x with respect to `math-integ-var'; this function returns `nil' if the integral could not be done. Some examples:

```
(math-defintegral calcFunc-conj
  (let ((int (math-integral u)))
    (and int
         (list 'calcFunc-conj int))))

(math-defintegral calcFunc-cos
  (and (equal u math-integ-var)
       (math-from-radians-2 (list 'calcFunc-sin u))))
```

In the `cos' example, we define only the integral of `cos(x) dx', relying on the general integration-by-substitution facility to handle cosines of more complicated arguments. An integration rule should return `nil' if it can't do the integral; if several rules are defined for the same function, they are tried in order until one returns a non-`nil' result.

Macro: **math-defintegral-2** *funcs body*

Define a rule for integrating a function or functions of two arguments. This is exactly analogous to `math-defintegral', except that body is written as the body of a function with two arguments, u and v.

Function: **solve-for** *lhs rhs var full*

Attempt to solve the equation `lhs = rhs' by isolating the variable var on the lefthand side; return the resulting righthand side, or `nil' if the equation cannot be solved. The variable var must appear at least once in lhs or rhs. Note that the return value is a formula which does not contain var; this is different from the user-level `solve' and `finv' functions, which return a rearranged equation or a functional inverse, respectively. If full is non-`nil', a full solution including dummy signs and dummy integers will be produced. User-defined inverses are provided as properties in a manner similar to derivatives:

```
(put 'calcFunc-ln 'math-inverse
     (function (lambda (x) (list 'calcFunc-exp x))))
```

This function can call `(math-solve-get-sign x)' to create a new arbitrary sign variable, returning x times that sign, and `(math-solve-get-int x)' to create a new arbitrary integer variable multiplied by x. These functions simply return x if the caller requested a non-"full" solution.

Function: **solve-eqn** *expr var full*

This version of `solve-for' takes an expression which will typically be an equation or inequality. (If it is not, it will be interpreted as the equation `expr = 0'.) It returns an equation or inequality, or `nil' if no solution could be found.

Function: **solve-system** *exprs vars full*

This function solves a system of equations. Generally, exprs and vars will be vectors of equal length. See section Solving Systems of Equations, for other options.

Function: **expr-contains** *expr var*

Returns a non-`nil' value if var occurs as a subexpression of expr.

This function might seem at first to be identical to `calc-find-sub-formula`. The key difference is th `expr-contains` uses `equal` to test for matches, whereas `calc-find-sub-formula` uses `eq`. In th formula `f(a, a)', the two `a's will be `equal` but not `eq` to each other.

Function: **expr-contains-count** *expr var*

Returns the number of occurrences of var as a subexpression of expr, or `nil` if there are no occurrences.

Function: **expr-depends** *expr var*

Returns true if expr refers to any variable the occurs in var. In other words, it checks if expr and var have any variables in common.

Function: **expr-contains-vars** *expr*

Return true if expr contains any variables, or `nil` if expr contains only constants and functions with constant arguments.

Function: **expr-subst** *expr old new*

Returns a copy of expr, with all occurrences of old replaced by new. This treats `lambda` forms specially with respect to the dummy argument variables, so that the effect is always to return expr evaluated at old = new.

Function: **multi-subst** *expr old new*

This is like `expr-subst`, except that old and new are lists of expressions to be substituted simultaneously. If one list is shorter than the other, trailing elements of the longer list are ignored.

Function: **expr-weight** *expr*

Returns the "weight" of expr, basically a count of the total number of objects and function calls that appear in expr. For "primitive" objects, this will be one.

Function: **expr-height** *expr*

Returns the "height" of expr, which is the deepest level to which function calls are nested. (Note that `a + b' counts as a function call.) For primitive objects, this returns zero.

Function: **polynomial-p** *expr var*

Check if expr is a polynomial in variable (or sub-expression) var. If so, return the degree of the polynomial, that is, the highest power of var that appears in expr. For example, for `(x^2 + 3)^3 + 4' this would return 6. This function returns `nil` unless expr, when expanded out by a x (`calc-expand`), would consist of a sum of terms in which var appears only raised to nonnegative integer powers. Note that if var does not occur in expr, then expr is considered a polynomial of degree 0.

Function: **is-polynomial** *expr var degree loose*

Check if expr is a polynomial in variable or sub-expression var, and, if so, return a list representation of the polynomial where the elements of the list are coefficients of successive powers of var: `a + b x + c x^3' would produce the list `(a b 0 c)', and `(x + 1)^2' would produce the list `(1 2 1)'. The highest element of the list will be non-zero, with the special exception that if expr is the constant zero, the returned value will be `(0)'. Return `nil` if expr is not a polynomial in var. If degree is specified, this will not consider polynomials of degree higher than that value. This is a good precaution because otherwise an input of `(x+1)^1000' will cause a huge coefficient list to be built. If loose is non-`nil`, then a looser definition of a polynomial is used in which coefficients are no

longer required not to depend on var, but are only required not to take the form of polynomials themselves. For example, `sin(x) x^2 + cos(x)' is a loose polynomial with coefficients `((calcFunc-cos x) 0 (calcFunc-sin x)'. The result will never be `nil` in loose mode, since any expression can be interpreted as a "constant" loose polynomial.

Function: **polynomial-base** *expr pred*

Check if expr is a polynomial in any variable that occurs in it; if so, return that variable. (If expr is a multivariate polynomial, this chooses one variable arbitrarily.) If pred is specified, it should be a Lisp function which is called as `(pred subexpr)', and which should return true if `mpb-top-expr` (a global name for the original expr) is a suitable polynomial in subexpr. The default predicate uses `(polynomial-p mpb-top-expr subexpr)'; you can use pred to specify additional conditions. Or, you could have pred build up a list of every suitable subexpr that is found.

Function: **poly-simplify** *poly*

Simplify polynomial coefficient list poly by (destructively) clipping off trailing zeros.

Function: **poly-mix** *a ac b bc*

Mix two polynomial lists a and b (in the form returned by `is-polynomial`) in a linear combination with coefficient expressions ac and bc. The result is a (not necessarily simplified) polynomial list representing `ac a + bc b'.

Function: **poly-mul** *a b*

Multiply two polynomial coefficient lists a and b. The result will be in simplified form if the inputs were simplified.

Function: **build-polynomial-expr** *poly var*

Construct a Calc formula which represents the polynomial coefficient list poly applied to variable var. The a c (`calc-collect`) command uses `is-polynomial` to turn an expression into a coefficient list, then `build-polynomial-expr` to turn the list back into an expression in regular form.

Function: **check-unit-name** *var*

Check if var is a variable which can be interpreted as a unit name. If so, return the units table entry for that unit. This will be a list whose first element is the unit name (not counting prefix characters) as a symbol and whose second element is the Calc expression which defines the unit. (Refer to the Calc sources for details on the remaining elements of this list.) If var is not a variable or is not a unit name, return `nil`.

Function: **units-in-expr-p** *expr sub-exprs*

Return true if expr contains any variables which can be interpreted as units. If sub-exprs is `t`, the entire expression is searched. If sub-exprs is `nil`, this checks whether expr is directly a units expression.

Function: **single-units-in-expr-p** *expr*

Check whether expr contains exactly one units variable. If so, return the units table entry for the variable. If expr does not contain any units, return `nil`. If expr contains two or more units, return the symbol `wrong`.

Function: **to-standard-units** *expr which*

Convert units expression expr to base units. If which is `nil`, use Calc's native base units. Otherwise, which can

specify a units system, which is a list of two-element lists, where the first element is a Calc base symbol na and the second is an expression to substitute for it.

Function: **remove-units** *expr*

Return a copy of expr with all units variables replaced by ones. This expression is generally normalized before use.

Function: **extract-units** *expr*

Return a copy of expr with everything but units variables replaced by ones.

I/O and Formatting Functions

The functions described here are responsible for parsing and formatting Calc numbers and formulas.

Function: **calc-eval** *str sep arg1 arg2 ...*

This is the simplest interface to the Calculator from another Lisp program. See section Calling Calc from Your Lisp Programs.

Function: **read-number** *str*

If string str contains a valid Calc number, either integer, fraction, float, or HMS form, this function parses and returns that number. Otherwise, it returns `nil`.

Function: **read-expr** *str*

Read an algebraic expression from string str. If str does not have the form of a valid expression, return a list of the form `(error pos msg)' where pos is an integer index into str of the general location of the error, and msg is a string describing the problem.

Function: **read-exprs** *str*

Read a list of expressions separated by commas, and return it as a Lisp list. If an error occurs in any expressions, an error list as shown above is returned instead.

Function: **calc-do-alg-entry** *initial prompt no-norm*

Read an algebraic formula or formulas using the minibuffer. All conventions of regular algebraic entry are observed. The return value is a list of Calc formulas; there will be more than one if the user entered a list of values separated by commas. The result is `nil` if the user presses Return with a blank line. If initial is given, it is a string which the minibuffer will initially contain. If prompt is given, it is the prompt string to use; the default is "Algebraic:". If no-norm is `t`, the formulas will be returned exactly as parsed; otherwise, they will be passed through `calc-normalize` first.

To support the use of $ characters in the algebraic entry, use `let` to bind `calc-dollar-values` to a list of the values to be substituted for $, $$, and so on, and bind `calc-dollar-used` to 0. Upon return, `calc-dollar-used` will have been changed to the highest number of consecutive $s that actually appeared in the input.

Function: **format-number** *a*

Convert the real or complex number or HMS form a to string form.

Function: **format-flat-expr** *a prec*

Convert the arbitrary Calc number or formula a to string form, in the style used by the trail buffer and the `calc-edit` command. This is a simple format designed mostly to guarantee the string is of a form that can be re-parsed by `read-expr`. Most formatting modes, such as digit grouping, complex number format, and point character, are ignored to ensure the result will be re-readable. The prec parameter is normally 0; if you pass a large integer like 1000 instead, the expression will be surrounded by parentheses unless it is a plain number or variable name.

Function: **format-nice-expr** *a width*

This is like `format-flat-expr` (with prec equal to 0), except that newlines will be inserted to keep lines down to the specified width, and vectors that look like matrices or rewrite rules are written in a pseudo-matrix format. The `calc-edit` command uses this when only one stack entry is being edited.

Function: **format-value** *a width*

Convert the Calc number or formula a to string form, using the format seen in the stack buffer. Beware the the string returned may not be re-readable by `read-expr`, for example, because of digit grouping. Multi-line objects like matrices produce strings that contain newline characters to separate the lines. The w parameter, if given, is the target window size for which to format the expressions. If w is omitted, the width of the Calculator window is used.

Function: **compose-expr** *a prec*

Format the Calc number or formula a according to the current language mode, returning a "composition." To learn about the structure of compositions, see the comments in the Calc source code. You can specify the format of a given type of function call by putting a `math-compose-lang` property on the function's symbol, whose value is a Lisp function that takes a and prec as arguments and returns a composition. Here lang is a language mode name, one of `normal`, `big`, `c`, `pascal`, `fortran`, `tex`, `eqn`, `math`, or `maple`. In Big mode, Calc actually tries `math-compose-big` first, then tries `math-compose-normal`. If this property does not exist, or if the function returns `nil`, the function is written in the normal function-call notation for that language.

Function: **composition-to-string** *c w*

Convert a composition structure returned by `compose-expr` into a string. Multi-line compositions convert to strings containing newline characters. The target window size is given by w. The `format-value` function basically calls `compose-expr` followed by `composition-to-string`.

Function: **comp-width** *c*

Compute the width in characters of composition c.

Function: **comp-height** *c*

Compute the height in lines of composition c.

Function: **comp-ascent** *c*

Compute the portion of the height of composition c which is on or above the baseline. For a one-line composition, this will be one.

Function: **comp-descent** *c*

Compute the portion of the height of composition c which is below the baseline. For a one-line composition this will be zero.

Function: **comp-first-char** *c*

If composition c is a "flat" composition, return the first (leftmost) character of the composition as an integer. Otherwise, return `nil`.

Function: **comp-last-char** *c*

If composition c is a "flat" composition, return the last (rightmost) character, otherwise return `nil`.

Hooks

Hooks are variables which contain Lisp functions (or lists of functions) which are called at various times. Calc defines a number of hooks that help you to customize it in various ways. Calc uses the Lisp function `run-hooks` to invoke the hooks shown below. Several other customization-related variables are also described here.

Variable: **calc-load-hook**

This hook is called at the end of `` `calc.el' ``, after the file has been loaded, before any functions in it have been called, but after `calc-mode-map` and similar variables have been set up.

Variable: **calc-ext-load-hook**

This hook is called at the end of `` `calc-ext.el' ``.

Variable: **calc-start-hook**

This hook is called as the last step in a M-x calc command. At this point, the Calc buffer has been created and initialized if necessary, the Calc window and trail window have been created, and the "Welcome to Calc" message has been displayed.

Variable: **calc-mode-hook**

This hook is called when the Calc buffer is being created. Usually this will only happen once per Emacs session. The hook is called after Emacs has switched to the new buffer, the mode-settings file has been read if necessary, and all other buffer-local variables have been set up. After this hook returns, Calc will perform a `calc-refresh` operation, set up the mode line display, then evaluate any deferred `calc-define` properties that have not been evaluated yet.

Variable: **calc-trail-mode-hook**

This hook is called when the Calc Trail buffer is being created. It is called as the very last step of setting up the Trail buffer. Like `calc-mode-hook`, this will normally happen only once per Emacs session.

Variable: **calc-end-hook**

This hook is called by `calc-quit`, generally because the user presses q or M-# c while in Calc. The Calc buffer will be the current buffer. The hook is called as the very first step, before the Calc window is destroyed.

Variable: **calc-window-hook**

If this hook exists, it is called to create the Calc window. Upon return, this new Calc window should be the

current window. (The Calc buffer will already be the current buffer when the hook is called.) If the hook is defined, Calc will generally use `split-window`, `set-window-buffer`, and `select-window` to cr the Calc window.

Variable: **calc-trail-window-hook**

If this hook exists, it is called to create the Calc Trail window. The variable `calc-trail-buffer` will contain the buffer which the window should use. Unlike `calc-window-hook`, this hook must *not* switch into the new window.

Variable: **calc-edit-mode-hook**

This hook is called by `calc-edit` (and the other "edit" commands) when the temporary editing buffer is being created. The buffer will have been selected and set up to be in `calc-edit-mode`, but will not yet have been filled with text. (In fact it may still have leftover text from a previous `calc-edit` command.)

Variable: **calc-mode-save-hook**

This hook is called by the `calc-save-modes` command, after Calc's own mode features have been inserted into the `` `.emacs' `` buffer and just before the "End of mode settings" message is inserted.

Variable: **calc-reset-hook**

This hook is called after M-# 0 (`calc-reset`) has reset all modes. The Calc buffer will be the current buffer.

Variable: **calc-other-modes**

This variable contains a list of strings. The strings are concatenated at the end of the modes portion of the Calc mode line (after standard modes such as "Deg", "Inv" and "Hyp"). Each string should be a short, single word followed by a space. The variable is `nil` by default.

Variable: **calc-mode-map**

This is the keymap that is used by Calc mode. The best time to adjust it is probably in a `calc-mode-hook`. If the Calc extensions package (`` `calc-ext.el' ``) has not yet been loaded, many of these keys will be bound to `calc-missing-key`, which is a command that loads the extensions package and "retypes" the key. If your `calc-mode-hook` rebinds one of these keys, it will probably be overridden when the extensions are loaded.

Variable: **calc-digit-map**

This is the keymap that is used during numeric entry. Numeric entry uses the minibuffer, but this map binds every non-numeric key to `calcDigit-nondigit` which generally calls `exit-minibuffer` and "retypes" the key.

Variable: **calc-alg-ent-map**

This is the keymap that is used during algebraic entry. This is mostly a copy of `minibuffer-local-map`.

Variable: **calc-store-var-map**

This is the keymap that is used during entry of variable names for commands like `calc-store` and `calc-recall`. This is mostly a copy of `minibuffer-local-completion-map`.

Variable: **calc-edit-mode-map**

This is the (sparse) keymap used by `calc-edit` and other temporary editing commands. It binds RET, LFD,

and C-c C-c to `calc-edit-finish`.

Variable: **calc-mode-var-list**

This is a list of variables which are saved by `calc-save-modes`. Each entry is a list of two items, the variable (as a Lisp symbol) and its default value. When modes are being saved, each variable is compared with its default value (using `equal`) and any non-default variables are written out.

Variable: **calc-local-var-list**

This is a list of variables which should be buffer-local to the Calc buffer. Each entry is a variable name (as a Lisp symbol). These variables also have their default values manipulated by the `calc` and `calc-quit` commands; see section Multiple Calculators. Since `calc-mode-hook` is called after this list has been used the first time, your hook should add a variable to the list and also call `make-local-variable` itself.

Go to the previous, next section.

Go to the previous, next section.

Installation

Calc 2.02 comes as a set of GNU Emacs Lisp files, with names like `calc.el' and `calc-ext.el', and also as a `calc.texinfo' file which can be used to generate both on-line and printed documentation.

To install Calc, just follow these simple steps. If you want more information, each step is discussed at length in the sections below.

1. Change (`cd') to the Calc "home" directory. This directory was created when you unbundled the Calc `.tar' or `.shar' file.
2. Type `make' to install Calc privately for your own use, or type `make install' to install Calc system-wide. This will compile all the Calc component files, modify your `.emacs' or the system-wide `lisp/default' file to install Calc as appropriate, and format the on-line Calc manual.

   Both variants are shorthand for the following three steps:

   - `make compile' to run the byte-compiler.
   - `make private' or `make public', corresponding to `make' and `make install', respectively. (If `make public' fails because your system doesn't already have a `default' or `default.el' file, use Emacs or the Unix `touch' command to create a zero-sized one first.)
   - `make info' to format the on-line Calc manual. This first tries to use the `makeinfo' program; if that program is not present, it uses the Emacs `texinfo-format-buffer' command instead.

   The Unix `make' utility looks in the file `Makefile' in the current directory to see what Unix commands correspond to the various "targets" like `install' or `public'. If your system doesn't have `make', you will have to examine the `Makefile' and type in the corresponding commands by hand.
3. If you ever move Calc to a new home directory, just give the `make private' or `make public' command again in the new directory.
4. Test your installation as described at the end of these instructions.
5. (Optional.) To print a hardcopy of the Calc manual (over 500 pages) or just the Calc Summary (about 20 pages), follow the instructions under "Printed Documentation" below.

Calc is now installed and ready to go!

Upgrading from Calc 1.07

If you have Calc version 1.07 or earlier, you will find that Calc 2.00 is organized quite differently. For one, Calc 2.00 is now distributed already split into many parts; formerly this was done as part of the installation procedure. Also, some new functions must be autoloaded and the M-# key must be bound to `calc-dispatch` instead of to `calc`.

The easiest way to upgrade is to delete your old Calc files and then install Calc 2.00 from scratch using the above instructions. You should then go into your `` `.emacs' `` or `` `default' `` file and remove the old `autoload` and `global-set-key` commands for Calc, since `make public'/`make private' has added new, better ones.

See the `` `README' `` and `` `README.prev' `` files in the Calc distribution for more information about what has changed since version 1.07. (`` `README.prev' `` describes changes before 2.00, and is present only in the FTP and tape versions of the distribution.)

The `make public' Command

If you are not the regular Emacs administrator on your system, your account may not be allowed to execute the `make public' command, since the system-wide `` `default' `` file may be write-protected. If this is the case, you will have to ask your Emacs installer to execute this command. (Just `cd` to the Calc home directory and type `make public'.)

The `make private' command adds exactly the same set of commands to your `` `.emacs' `` file as `make public' adds to `` `default' ``. If your Emacs installer is concerned about typing this command out of the blue, you can ask her/him instead to copy the necessary text from your `` `.emacs' `` file. (It will be marked by a comment that says "Commands added by `calc-private-autoloads` on (date and time).")

Compilation

Calc is written in a way that maximizes performance when its code has been byte-compiled; a side effect is that performance is seriously degraded if it *isn't* compiled. Thus, it is essential to compile the Calculator before trying to use it. The function `calc-compile' in the file `` `calc-maint.el' `` runs the Emacs byte-compiler on all the Calc source files. (Specifically, it runs M-x byte-compile-file on all files in the current directory with names of the form `` `calc*.el' ``, and also on the file `` `macedit.el' ``.)

If `calc-compile` finds that certain files have already been compiled and have not been changed since, then it will not bother to recompile those files.

The `calc-compile` command also pre-builds certain tables, such as the units table (see section The Units Table) and the built-in rewrite rules (see section Rearranging Formulas using Selections) which Calc would otherwise need to rebuild every time those features were used.

The `make compile' shell command is simply a convenient way to start an Emacs and give it a

`calc-compile` command.

Auto-loading

To teach Emacs how to load in Calc when you type M-# for the first time, add these lines to your `.emacs' file (if you are installing Calc just for your own use), or the system's `lisp/default' file (if you are installing Calc publicly). The `make private' and `make public' commands, respectively, take care of this. (Note that `make' runs `make private', and `make install' runs `make public'.)

```
(autoload 'calc-dispatch          "calc" "Calculator Options" t)
(autoload 'full-calc              "calc" "Full-screen Calculator" t)
(autoload 'full-calc-keypad       "calc" "Full-screen X Calculator" t)
(autoload 'calc-eval              "calc" "Use Calculator from Lisp")
(autoload 'defmath                "calc" nil t t)
(autoload 'calc                   "calc" "Calculator Mode" t)
(autoload 'quick-calc             "calc" "Quick Calculator" t)
(autoload 'calc-keypad            "calc" "X windows Calculator" t)
(autoload 'calc-embedded          "calc" "Use Calc from any buffer" t)
(autoload 'calc-embedded-activate "calc" "Activate =>'s in buffer" t)
(autoload 'calc-grab-region       "calc" "Grab region of Calc data" t)
(autoload 'calc-grab-rectangle    "calc" "Grab rectangle of data" t)
```

Unless you have installed the Calc files in Emacs' main `lisp/' directory, you will also have to add a command that looks like the following to tell Emacs where to find them. In this example, we have put the files in directory `/usr/gnu/src/calc-2.00'.

```
(setq load-path (append load-path (list "/usr/gnu/src/calc-2.00")))
```

The `make public' and `make private' commands also do this (they use the then-current directory as the name to add to the path). If you move Calc to a new location, just repeat the `make public' or `make private' command to have this new location added to the `load-path'.

The `autoload' command for `calc-dispatch' is what loads `calc.elc' when you type M-#. It is the only `autoload' that is absolutely necessary for Calc to work. The others are for commands and features that you may wish to use before typing M-# for the first time. In particular, `full-calc' and `full-calc-keypad' are autoloaded to support "standalone" operation (see section Standalone Operation), `calc-eval' and `defmath' are autoloaded to allow other Emacs Lisp programs to use Calc facilities (see section Calling Calc from Your Lisp Programs), and `calc-embedded-activate' is autoloaded because some Embedded Mode files may call it as soon as they are read into Emacs (see section Assignments in Embedded Mode).

Finding Component Files

There is no need to write `autoload` commands that point to all the various Calc component files like `` `calc-misc.elc' `` and `` `calc-alg.elc' ``. The main file, `` `calc.elc' ``, contains all the necessary `autoload` commands for these files.

(Actually, to conserve space `` `calc.elc' `` only autoloads a few of the component files, plus `` `calc-ext.elc' ``, which in turn autoloads the rest of the components. This allows Calc to load a little faster in the beginning, but the net effect is the same.)

This autoloading mechanism assumes that all the component files can be found on the `load-path`. The `make public' and `make private' commands take care of this, but Calc has a few other strategies in case you have installed it in an unusual way.

If, when Calc is loaded, it is unable to find its components on the `load-path` it is given, it checks the file name in the original `autoload` command for `calc-dispatch`. If that name included directory information, Calc adds that directory to the `load-path`:

```
(autoload 'calc-dispatch "calc-2.00/calc" "Calculator" t)
```

Suppose the directory `` `/usr/gnu/src/emacs/lisp' `` is on the path, and the above `autoload` allows Emacs to find Calc under the name `` `/usr/gnu/src/emacs/lisp/calc-2.00/calc.elc' ``. Then when Calc starts up it will add `` `/usr/gnu/src/emacs/lisp/calc-2.00' `` to the path so that it will later be able to find its component files.

If the above strategy does not locate the component files, Calc examines the variable `calc-autoload-directory`. This is initially `nil`, but you can store the name of Calc's home directory in it as a sure-fire way of getting Calc to find its components.

Merging Source Files

If the `autoload` mechanism is not managing to load each part of Calc when it is needed, you can concatenate all the `` `.el' `` files into one big file. The order should be `` `calc.el' ``, then `` `calc-ext.el' ``, then all the other files in any order. Byte-compile the resulting big file. This merged Calculator ought to work just like Calc normally does, though it will be *substantially* slower to load.

Key Bindings

Calc is normally bound to the M-# key. To set up this key binding, include the following command in your `` `.emacs' `` or `` `lisp/default' `` file. (This is done automatically by `make private' or `make public', respectively.)

```
(global-set-key "\e#" 'calc-dispatch)
```

Note that `calc-dispatch` actually works as a prefix for various two-key sequences. If you have a convenient unused function key on your keyboard, you may wish to bind `calc-dispatch` to that as well. You may even wish to bind other specific Calc functions like `calc` or `quick-calc` to other handy function keys.

Even if you bind `calc-dispatch` to other keys, it is best to bind it to M-# as well if you possibly can: There are references to M-# all throughout the Calc manual which would confuse novice users if they didn't work as advertised.

Another key binding issue is the DEL key. Some installations use a different key (such as backspace) for this purpose. Calc normally scans the entire keymap and maps all keys defined like DEL to the `calc-pop` command. However, this may be slow. You can set the variable `calc-scan-for-dels` to `nil` to cause only the actual DEL key to be mapped to `calc-pop`; this will speed loading of Calc.

The `macedit' Package

The file `macedit.el' contains another useful Emacs extension called `edit-kbd-macro`. It allows you to edit a keyboard macro in human-readable form. The Z E command in Calc knows how to use it to edit user commands that have been defined by keyboard macros. To autoload it, you will want to include the commands,

```
(autoload 'edit-kbd-macro      "macedit" "Edit Keyboard Macro" t)
(autoload 'edit-last-kbd-macro "macedit" "Edit Keyboard Macro" t)
(autoload 'read-kbd-macro      "macedit" "Read Keyboard Macro" t)
```

The `make public' and `make private' commands do this.

The GNUPLOT Program

Calc's graphing commands use the GNUPLOT program. If you have GNUPLOT but you must type some command other than `gnuplot' to get it, you should add a command to set the Lisp variable `calc-gnuplot-name` to the appropriate file name. You may also need to change the variables `calc-gnuplot-plot-command` and `calc-gnuplot-print-command` in order to get correct displays and hardcopies, respectively, of your plots.

On-Line Documentation

The documentation for Calc (this manual) comes in a file called `calc.texinfo'. To format this for use as an on-line manual, type `make info' (to use the `makeinfo` program), or `make texinfo' (to use the `texinfmt.el` program which runs inside of Emacs). The former command is recommended if it works on your system; it is faster and produces nicer-looking output.

The `makeinfo` program will report inconsistencies involving the nodes "Copying" and "Interactive Tutorial"; these messages should be ignored.

The result will be a collection of files whose names begin with `calc.info'. You may wish to ad reference to the first of these, `calc.info' itself, to your Info system's `dir' file. (This is optio since the M-# i command can access `calc.info' whether or not it appears in the `dir' file.)

There is a Lisp variable called calc-info-filename which holds the name of the Info file containing Calc's on-line documentation. Its default value is "calc.info", which will work correctly if the Info files are stored in Emacs' main `info/' directory, or if they are in any of the directories listed in the load-path. If you keep them elsewhere, you will want to put a command of the form,

```
(setq calc-info-filename ".../calc.info")
```

in your `.emacs' or `lisp/default' file, where `...' represents the directory containing the Info files. This will not be necessary if you follow the normal installation procedures.

The `make info' and `make texinfo' commands compare the dates on the files `calc.texinfo' and `calc.info', and run the appropriate program only if the latter file is older or does not exist.

Printed Documentation

Because the Calc manual is so large, you should only make a printed copy if you really need it. To print the manual, you will need the TeX typesetting program (this is a free program by Donald Knuth at Stanford University) as well as the `texindex' program and `texinfo.tex' file, both of which can be obtained from the FSF as part of the texinfo2 package.

To print the Calc manual in one huge 550 page tome, type `make tex'. This will take care of running the manual through TeX twice so that references to later parts of the manual will have correct page numbers. (Don't worry if you get some "overfull box" warnings.)

The result will be a device-independent output file called `calc.dvi', which you must print in whatever way is right for your system. On many systems, the command is

```
lpr -d calc.dvi
```

Marginal notes for each function and key sequence normally alternate between the left and right sides of the page, which is correct if the manual is going to be bound as double-sided pages. Near the top of the file `calc.texinfo' you will find alternate definitions of the \bumpoddpages macro that put the marginal notes always on the same side, best if you plan to be binding single-sided pages.

Some people find the Calc manual to be too large to handle easily. In fact, some versions of TeX have too little memory to print it. So Calc includes a calc-split-manual command that splits `calc.texinfo' into two volumes, the Calc Tutorial and the Calc Reference. The easiest way to use it is to type `make tex2' instead of `make tex'. The result will be two smaller files, `calctut.dvi' and `calcref.dvi'. The former contains the tutorial part of the manual; the latter contains the reference part. Both volumes include copies of the "Getting Started" chapter and licensing information.

To save disk space, you may wish to delete `calctut.*' and `calcref.*' after you're done. Don't delete `calc.texinfo', because you will need it to install future patches to Calc. The `make

tex2' command takes care of all of this for you.

The `make textut' command formats only the Calc Tutorial volume, producing `calctut.dvi' but not `calcref.dvi'. Likewise, `make texref' formats only the Calc Reference volume.

Finally, there is a `calc-split-summary` command that splits off just the Calc Summary appendix suitable for printing by itself. Type `make summary' instead of `make tex'. The resulting `calcsum.dvi' file will print in less than 20 pages. If the Key Index file `calc.ky' is present, left over from a previous `make tex' command, then `make summary' will insert a column of page numbers into the summary using that information.

The `make isummary' command is like `make summary', but it prints a summary that is designed to be substituted into the regular manual. (The two summaries will be identical except for the additional column of page numbers.) To make a complete manual, run `make tex' and `make isummary', print the two resulting `.dvi' files, then discard the Summary pages that came from `calc.dvi' and insert the ones from `calcsum.dvi' in their place. Also, remember that the table of contents prints at the end of the manual but should generally be moved to the front (after the title and copyright pages).

If you don't have TeX, you can print the summary as a plain text file by going to the "Summary" node in Calc's Info file, then typing M-x print-buffer (see section Calc Summary).

Settings File

Another variable you might want to set is `calc-settings-file`, which holds the file name in which commands like m m and Z P store "permanent" definitions. The default value for this variable is `"~/.emacs"`. If `calc-settings-file` does not contain `".emacs"` as a substring, and if the variable `calc-loaded-settings-file` is `nil`, then Calc will automatically load your settings file (if it exists) the first time Calc is invoked.

Testing the Installation

To test your installation of Calc, start a new Emacs and type M-# c to make sure the autoloads and key bindings work. Type M-# i to make sure Calc can find its Info documentation. Press q to exit the Info system and M-# c to re-enter the Calculator. Type 20 S to compute the sine of 20 degrees; this will test the autoloading of the extensions modules. The result should be 0.342020143326. Finally, press M-# c again to make sure the Calculator can exit.

You may also wish to test the GNUPLOT interface; to plot a sine wave, type ' [0 .. 360], sin(x) RET g f. Type g q when you are done viewing the plot.

Calc is now ready to use. If you wish to go through the Calc Tutorial, press M-# t to begin.

(The above text is included in both the Calc documentation and the file INSTALL in the Calc distribution directory.)

Go to the previous, next section.

TOC

Go to the previous, next section.

Reporting Bugs

If you find a bug in Calc, send e-mail to Dave Gillespie,

```
daveg@synaptics.com           or
daveg@csvax.cs.caltech.edu
```

While I cannot guarantee that I will have time to work on your bug, I do try to fix bugs quickly whenever I can.

You can obtain a current version of Calc from anonymous FTP on `csvax.cs.caltech.edu' in `pub/calc-version.tar.Z'. Calc is also available on the GNU machine, `prep.ai.mit.edu'. Patches are posted to the `comp.sources.misc' Usenet group, and are also available on `csvax`.

There is an automatic M-x report-calc-bug command which helps you to report bugs. This command prompts you for a brief subject line, then leaves you in a mail editing buffer. Type C-c C-c to send your mail. Make sure your subject line indicates that you are reporting a Calc bug; this command sends mail to my regular mailbox.

If you have suggestions for additional features for Calc, I would love to hear them. Some have dared to suggest that Calc is already top-heavy with features; I really don't see what they're talking about, so, if you have ideas, send them right in. (I may even have time to implement them!)

At the front of the source file, `calc.el', is a list of ideas for future work which I have not had time to do. If any enthusiastic souls wish to take it upon themselves to work on these, I would be delighted. Please let me know if you plan to contribute to Calc so I can coordinate your efforts with mine and those of others. I will do my best to help you in whatever way I can.

Go to the previous, next section.

Go to the previous, next section.

Calc Summary

This section includes a complete list of Calc 2.02 keystroke commands. Each line lists the stack entries used by the command (top-of-stack last), the keystrokes themselves, the prompts asked by the command, and the result of the command (also with top-of-stack last). The result is expressed using the equivalent algebraic function. Commands which put no results on the stack show the full M-x command name in that position. Numbers preceding the result or command name refer to notes at the end.

Algebraic functions and M-x commands that don't have corresponding keystrokes are not listed in this summary. See section Index of Calculator Commands. See section Index of Algebraic Functions.

@begingroup @let=@sumsep @let@r=@sumrow @catcode`@(=@active @let(=@sumlpar @catcode`@)=@active @let)=@sumrpar @catcode`@,=@active @let,=@sumcomma @catcode`!=@active @let!=@sumexcl

```
@advance@baselineskip-2.5pt
@let@tt@ninett
@let@c@sumbreak
             M-# a                      33  calc-embedded-activate
             M-# b                          calc-big-or-small
             M-# c                          calc
             M-# d                          calc-embedded-duplicate
             M-# e                      34  calc-embedded
             M-# f  formula                 calc-embedded-new-formula
             M-# g                      35  calc-grab-region
             M-# i                          calc-info
             M-# j                          calc-embedded-select
             M-# k                          calc-keypad
             M-# l                          calc-load-everything
             M-# m                          read-kbd-macro
             M-# n                       4  calc-embedded-next
             M-# o                          calc-other-window
             M-# p                       4  calc-embedded-previous
             M-# q  formula                 quick-calc
             M-# r                      36  calc-grab-rectangle
             M-# s                          calc-info-summary
             M-# t                          calc-tutorial
             M-# u                          calc-embedded-update
             M-# w                          calc-embedded-word
             M-# x                          calc-quit
             M-# y                  1,28,49 calc-copy-to-buffer
             M-# z                          calc-user-invocation
```


```
            M-# :                        36  calc-grab-sum-down
            M-# _                        36  calc-grab-sum-across
            M-# `  editing               30  calc-embedded-edit
            M-# 0  (zero)                    calc-reset

             0-9   number                    number
             .     number                    0.number
             _     number                    -number
             e     number                    1e number
             #     number                    current-radix#number
             P     (in number)               +/-
             M     (in number)               mod
             @ ' "   (in number)             HMS form
             h m s   (in number)             HMS form

             '     formula             37,46  formula
             $     formula             37,46  $formula
             "     string              37,46  string

   a b       +                            2  add(a,b)  a+b
   a b       -                            2  sub(a,b)  a-b
   a b       *                            2  mul(a,b)  a b, a*b
   a b       /                            2  div(a,b)  a/b
   a b       ^                            2  pow(a,b)  a^b
   a b     I ^                            2  nroot(a,b)  a^(1/b)
   a b       %                            2  mod(a,b)  a%b
   a b       \                            2  idiv(a,b)  a\b
   a b       :                            2  fdiv(a,b)
   a b       |                            2  vconcat(a,b)  a|b
   a b     I |                               vconcat(b,a)  b|a
   a b     H |                            2  append(a,b)
   a b   I H |                               append(b,a)
     a       &                            1  inv(a)  1/a
     a       !                            1  fact(a)  a!
     a       =                            1  evalv(a)
     a       M-%                             percent(a)  a%

  ... a      RET                          1  ... a a
  ... a      SPC                          1  ... a a
... a b      TAB                          3  ... b a
. a b c      M-TAB                        3  ... b c a
... a b      LFD                          1  ... a b a
  ... a      DEL                          1  ...
... a b      M-DEL                        1  ... b
             M-RET                        4  calc-last-args
       a     `     editing             1,30  calc-edit
```

...	a		C-d	1	...
			C-k	27	calc-kill
			C-w	27	calc-kill-region
			C-y		calc-yank
			C-_	4	calc-undo
			M-k	27	calc-copy-as-kill
			M-w	27	calc-copy-region-as-kill
			[		[...
[..	a	b	]		[a,b]
			(		(...
(..	a	b	)		(a,b)
			,		vector or rect complex
			;		matrix or polar complex
			..		interval
			~		calc-num-prefix
			<	4	calc-scroll-left
			>	4	calc-scroll-right
			{	4	calc-scroll-down
			}	4	calc-scroll-up
			?		calc-help
	a		n	1	neg(a) -a
			o	4	calc-realign
			p precision	31	calc-precision
			q		calc-quit
			w		calc-why
			x command		M-x calc-command
	a		y	1,28,49	calc-copy-to-buffer
	a		A	1	abs(a)
	a	b	B	2	log(a,b)
	a	b I	B	2	alog(a,b) b^a
	a		C	1	cos(a)
	a	I	C	1	arccos(a)
	a	H	C	1	cosh(a)
	a	I H	C	1	arccosh(a)
			D	4	calc-redo
	a		E	1	exp(a)
	a	H	E	1	exp10(a) 10.^a
	a		F	1,11	floor(a,d)
	a	I	F	1,11	ceil(a,d)
	a	H	F	1,11	ffloor(a,d)
	a	I H	F	1,11	fceil(a,d)


```
     a       G                          1  arg(a)
             H     command             32  Hyperbolic
             I     command             32  Inverse
     a       J                          1  conj(a)
             K     command             32  Keep-args
     a       L                          1  ln(a)
     a     H L                          1  log10(a)
             M                             calc-more-recursion-depth
           I M                             calc-less-recursion-depth
     a       N                          5  evalvn(a)
             P                             pi
           I P                             gamma
           H P                             e
        I  H P                             phi
     a       Q                          1  sqrt(a)
     a     I Q                          1  sqr(a)  a^2
     a       R                        1,11  round(a,d)
     a     I R                        1,11  trunc(a,d)
     a     H R                        1,11  fround(a,d)
     a  I  H R                        1,11  ftrunc(a,d)
     a       S                          1  sin(a)
     a     I S                          1  arcsin(a)
     a     H S                          1  sinh(a)
     a  I  H S                          1  arcsinh(a)
     a       T                          1  tan(a)
     a     I T                          1  arctan(a)
     a     H T                          1  tanh(a)
     a  I  H T                          1  arctanh(a)
             U                          4  calc-undo
             X                          4  calc-call-last-kbd-macro

   a b       a =                        2  eq(a,b)  a=b
   a b       a #                        2  neq(a,b)  a!=b
   a b       a <                        2  lt(a,b)  a<b
   a b       a >                        2  gt(a,b)  a>b
   a b       a [                        2  leq(a,b)  a<=b
   a b       a ]                        2  geq(a,b)  a>=b
   a b       a {                        2  in(a,b)
   a b       a &                      2,45  land(a,b)  a&&b
   a b       a |                      2,45  lor(a,b)  a||b
     a       a !                      1,45  lnot(a)  !a
 a b c       a :                        45  if(a,b,c)  a?b:c
     a       a .                         1  rmeq(a)
     a       a "                       7,8  calc-expand-formula

     a       a +   i, l, h            6,38  sum(a,i,l,h)
```

	a		a -	i, l, h	6,38	asum(a,i,l,h)
	a		a *	i, l, h	6,38	prod(a,i,l,h)
a b			a _		2	subscr(a,b) a_b
a b			a \		2	pdiv(a,b)
a b			a %		2	prem(a,b)
a b			a /		2	pdivrem(a,b) [q,r]
a b		H	a /		2	pdivide(a,b) q+r/b
	a		a a		1	apart(a)
	a		a b	old, new	38	subst(a,old,new)
	a		a c	v	38	collect(a,v)
	a		a d	v	4,38	deriv(a,v)
	a	H	a d	v	4,38	tderiv(a,v)
	a		a e			esimplify(a)
	a		a f		1	factor(a)
	a	H	a f		1	factors(a)
a b			a g		2	pgcd(a,b)
	a		a i	v	38	integ(a,v)
	a		a m	pats	38	match(a,pats)
	a	I	a m	pats	38	matchnot(a,pats)
data x			a p		28	polint(data,x)
data x		H	a p		28	ratint(data,x)
	a		a n		1	nrat(a)
	a		a r	rules	4,8,38	rewrite(a,rules,n)
	a		a s			simplify(a)
	a		a t	v, n	31,39	taylor(a,v,n)
	a		a v		7,8	calc-alg-evaluate
	a		a x		4,8	expand(a)
data			a F	model, vars	48	fit(m,iv,pv,data)
data		I	a F	model, vars	48	xfit(m,iv,pv,data)
data		H	a F	model, vars	48	efit(m,iv,pv,data)
	a		a I	v, l, h	38	ninteg(a,v,l,h)
a b			a M	op	22	mapeq(op,a,b)
a b		I	a M	op	22	mapeqr(op,a,b)
a b		H	a M	op	22	mapeqp(op,a,b)
a g			a N	v	38	minimize(a,v,g)
a g		H	a N	v	38	wminimize(a,v,g)
	a		a P	v	38	roots(a,v)
a g			a R	v	38	root(a,v,g)
a g		H	a R	v	38	wroot(a,v,g)
	a		a S	v	38	solve(a,v)
	a	I	a S	v	38	finv(a,v)
	a	H	a S	v	38	fsolve(a,v)
	a	I H	a S	v	38	ffinv(a,v)


```
      a       a T    i, l, h          6,38  table(a,i,l,h)
    a g       a X    v                  38  maximize(a,v,g)
    a g     H a X    v                  38  wmaximize(a,v,g)

    a b       b a                        9  and(a,b,w)
      a       b c                        9  clip(a,w)
    a b       b d                        9  diff(a,b,w)
      a       b l                       10  lsh(a,n,w)
    a n     H b l                        9  lsh(a,n,w)
      a       b n                        9  not(a,w)
    a b       b o                        9  or(a,b,w)
      v       b p                        1  vpack(v)
      a       b r                       10  rsh(a,n,w)
    a n     H b r                        9  rsh(a,n,w)
      a       b t                       10  rot(a,n,w)
    a n     H b t                        9  rot(a,n,w)
      a       b u                        1  vunpack(a)
              b w    w                9,50  calc-word-size
    a b       b x                        9  xor(a,b,w)

c s l p       b D                           ddb(c,s,l,p)
  r n p       b F                           fv(r,n,p)
  r n p     I b F                           fvb(r,n,p)
  r n p     H b F                           fvl(r,n,p)
      v       b I                       19  irr(v)
      v     I b I                       19  irrb(v)
      a       b L                       10  ash(a,n,w)
    a n     H b L                        9  ash(a,n,w)
  r n a       b M                           pmt(r,n,a)
  r n a     I b M                           pmtb(r,n,a)
  r n a     H b M                           pmtl(r,n,a)
    r v       b N                       19  npv(r,v)
    r v     I b N                       19  npvb(r,v)
  r n p       b P                           pv(r,n,p)
  r n p     I b P                           pvb(r,n,p)
  r n p     H b P                           pvl(r,n,p)
      a       b R                       10  rash(a,n,w)
    a n     H b R                        9  rash(a,n,w)
  c s l       b S                           sln(c,s,l)
  n p a       b T                           rate(n,p,a)
  n p a     I b T                           rateb(n,p,a)
  n p a     H b T                           ratel(n,p,a)
c s l p       b Y                           syd(c,s,l,p)

  r p a       b #                           nper(r,p,a)
  r p a     I b #                           nperb(r,p,a)
```


```
 r p a    H b #                          nperl(r,p,a)
   a b      b %                          relch(a,b)

     a      c c                      5   pclean(a,p)
     a      c 0-9                        pclean(a,p)
     a    H c c                      5   clean(a,p)
     a    H c 0-9                      clean(a,p)
     a      c d                      1   deg(a)
     a      c f                      1   pfloat(a)
     a    H c f                      1   float(a)
     a      c h                      1   hms(a)
     a      c p                          polar(a)
     a    I c p                          rect(a)
     a      c r                      1   rad(a)

     a      c F                      5   pfrac(a,p)
     a    H c F                      5   frac(a,p)

     a      c %                          percent(a*100)

            d .   char                50  calc-point-char
            d ,   char                50  calc-group-char
            d <                    13,50  calc-left-justify
            d =                    13,50  calc-center-justify
            d >                    13,50  calc-right-justify
            d {   label               50  calc-left-label
            d }   label               50  calc-right-label
            d [                        4  calc-truncate-up
            d ]                        4  calc-truncate-down
            d "                    12,50  calc-display-strings
            d SPC                         calc-refresh
            d RET                      1  calc-refresh-top

            d 0                       50  calc-decimal-radix
            d 2                       50  calc-binary-radix
            d 6                       50  calc-hex-radix
            d 8                       50  calc-octal-radix

            d b                 12,13,50  calc-line-breaking
            d c                       50  calc-complex-notation
            d d   format              50  calc-date-notation
            d e                    5,50  calc-eng-notation
            d f   num              31,50  calc-fix-notation
            d g                 12,13,50  calc-group-digits
            d h   format              50  calc-hms-notation
            d i                       50  calc-i-notation
```

			d j			50	calc-j-notation
			d l			12,50	calc-line-numbering
			d n			5,50	calc-normal-notation
			d o		format	50	calc-over-notation
			d p			12,50	calc-show-plain
			d r		radix	31,50	calc-radix
			d s			5,50	calc-sci-notation
			d t			27	calc-truncate-stack
			d w			12,13	calc-auto-why
			d z			12,50	calc-leading-zeros
			d B			50	calc-big-language
			d C			50	calc-c-language
			d E			50	calc-eqn-language
			d F			50	calc-fortran-language
			d M			50	calc-mathematica-language
			d N			50	calc-normal-language
			d O			50	calc-flat-language
			d P			50	calc-pascal-language
			d T			50	calc-tex-language
			d U			50	calc-unformatted-language
			d W			50	calc-maple-language
		a	f [			4	decr(a,n)
		a	f]			4	incr(a,n)
	a	b	f b			2	beta(a,b)
		a	f e			1	erf(a)
		a	f e	I		1	erfc(a)
		a	f g			1	gamma(a)
	a	b	f h			2	hypot(a,b)
		a	f i			1	im(a)
	n	a	f j			2	besJ(n,a)
	a	b	f n			2	min(a,b)
		a	f r			1	re(a)
		a	f s			1	sign(a)
	a	b	f x			2	max(a,b)
	n	a	f y			2	besY(n,a)
		a	f A			1	abssqr(a)
x	a	b	f B				betaI(x,a,b)
x	a	b	f B	H			betaB(x,a,b)
		a	f E			1	expm1(a)
	a	x	f G			2	gammaP(a,x)
	a	x	f G	I		2	gammaQ(a,x)
	a	x	f G	H		2	gammag(a,x)

a	x	I	H	f	G	2	gammaG(a,x)
a	b			f	I	2	ilog(a,b)
a	b		I	f	I	2	alog(a,b) b^a
	a			f	L	1	lnp1(a)
	a			f	M	1	mant(a)
	a			f	Q	1	isqrt(a)
	a		I	f	Q	1	sqr(a) a^2
a	n			f	S	2	scf(a,n)
y	x			f	T		arctan2(y,x)
	a			f	X	1	xpon(a)
x	y			g	a	28,40	calc-graph-add
				g	b	12	calc-graph-border
				g	c		calc-graph-clear
				g	d	41	calc-graph-delete
x	y			g	f	28,40	calc-graph-fast
				g	g	12	calc-graph-grid
				g	h	title	calc-graph-header
				g	j	4	calc-graph-juggle
				g	k	12	calc-graph-key
				g	l	12	calc-graph-log-x
				g	n	name	calc-graph-name
				g	p	42	calc-graph-plot
				g	q		calc-graph-quit
				g	r	range	calc-graph-range-x
				g	s	12,13	calc-graph-line-style
				g	t	title	calc-graph-title-x
				g	v		calc-graph-view-commands
				g	x	display	calc-graph-display
				g	z	12	calc-graph-zero-x
x	y	z		g	A	28,40	calc-graph-add-3d
				g	C	command	calc-graph-command
				g	D	device	43,44 calc-graph-device
x	y	z		g	F	28,40	calc-graph-fast-3d
				g	H	12	calc-graph-hide
				g	K		calc-graph-kill
				g	L	12	calc-graph-log-y
				g	N	number	43,51 calc-graph-num-points
				g	O	filename	43,44 calc-graph-output
				g	P	42	calc-graph-print
				g	R	range	calc-graph-range-y
				g	S	12,13	calc-graph-point-style
				g	T	title	calc-graph-title-y
				g	V		calc-graph-view-trail
				g	X	format	calc-graph-geometry


```
g Z                      12  calc-graph-zero-y

g C-l                    12  calc-graph-log-z
g C-r range                  calc-graph-range-z
g C-t title                  calc-graph-title-z

h b                          calc-describe-bindings
h c   key                    calc-describe-key-briefly
h f   function               calc-describe-function
h h                          calc-full-help
h i                          calc-info
h k   key                    calc-describe-key
h n                          calc-view-news
h s                          calc-info-summary
h t                          calc-tutorial
h v   var                    calc-describe-variable

j 1-9                        calc-select-part
j RET                    27  calc-copy-selection
j DEL                    27  calc-del-selection
j '   formula            27  calc-enter-selection
j `   editing         27,30  calc-edit-selection
j "                    7,27  calc-sel-expand-formula

j +   formula            27  calc-sel-add-both-sides
j -   formula            27  calc-sel-sub-both-sides
j *   formula            27  calc-sel-mul-both-sides
j /   formula            27  calc-sel-div-both-sides
j &                      27  calc-sel-invert

j a                      27  calc-select-additional
j b                      12  calc-break-selections
j c                          calc-clear-selections
j d                   12,50  calc-show-selections
j e                      12  calc-enable-selections
j l                    4,27  calc-select-less
j m                    4,27  calc-select-more
j n                       4  calc-select-next
j o                    4,27  calc-select-once
j p                       4  calc-select-previous
j r   rules         4,8,27  calc-rewrite-selection
j s                    4,27  calc-select-here
j u                      27  calc-unselect
j v                    7,27  calc-sel-evaluate

j C                      27  calc-sel-commute
```

		j D	4,27	calc-sel-distribute
		j E	27	calc-sel-jump-equals
		j I	27	calc-sel-isolate
	H	j I	27	calc-sel-isolate (full)
		j L	4,27	calc-commute-left
		j M	27	calc-sel-merge
		j N	27	calc-sel-negate
		j O	4,27	calc-select-once-maybe
		j R	4,27	calc-commute-right
		j S	4,27	calc-select-here-maybe
		j U	27	calc-sel-unpack
		k a		calc-random-again
n		k b	1	bern(n)
n	x	H k b	2	bern(n,x)
n	m	k c	2	choose(n,m)
n	m	H k c	2	perm(n,m)
n		k d	1	dfact(n) n!!
n		k e	1	euler(n)
n	x	H k e	2	euler(n,x)
n		k f	4	prfac(n)
n	m	k g	2	gcd(n,m)
m	n	k h	14	shuffle(n,m)
n	m	k l	2	lcm(n,m)
n		k m	1	moebius(n)
n		k n	4	nextprime(n)
n		I k n	4	prevprime(n)
n		k p	4,28	calc-prime-test
m		k r	14	random(m)
n	m	k s	2	stir1(n,m)
n	m	H k s	2	stir2(n,m)
n		k t	1	totient(n)
n	p x	k B		utpb(x,n,p)
n	p x	I k B		ltpb(x,n,p)
v	x	k C		utpc(x,v)
v	x	I k C		ltpc(x,v)
n	m	k E		egcd(n,m)
v1	v2 x	k F		utpf(x,v1,v2)
v1	v2 x	I k F		ltpf(x,v1,v2)
m	s x	k N		utpn(x,m,s)
m	s x	I k N		ltpn(x,m,s)
m	x	k P		utpp(x,m)
m	x	I k P		ltpp(x,m)
v	x	k T		utpt(x,v)
v	x	I k T		ltpt(x,v)

	m a		12,13	calc-algebraic-mode
	m d			calc-degrees-mode
	m f		12	calc-frac-mode
	m g		52	calc-get-modes
	m h			calc-hms-mode
	m i		12,13	calc-infinite-mode
	m m			calc-save-modes
	m p		12	calc-polar-mode
	m r			calc-radians-mode
	m s		12	calc-symbolic-mode
	m t		12	calc-total-algebraic-mode
	m v		12,13	calc-matrix-mode
	m w		13	calc-working
	m x			calc-always-load-extensions
	m A		12	calc-alg-simplify-mode
	m B		12	calc-bin-simplify-mode
	m C		12	calc-auto-recompute
	m D			calc-default-simplify-mode
	m E		12	calc-ext-simplify-mode
	m F	filename	13	calc-settings-file-name
	m N		12	calc-num-simplify-mode
	m O		12	calc-no-simplify-mode
	m R		12,13	calc-mode-record-mode
	m S		12	calc-shift-prefix
	m U		12	calc-units-simplify-mode
	s c	var1, var2	29	calc-copy-variable
	s d	var, decl		calc-declare-variable
	s e	var, editing	29,30	calc-edit-variable
	s i	buffer		calc-insert-variables
a b	s l	var	29	a (letting var=b)
a ...	s m	op, var	22,29	calc-store-map
	s n	var	29,47	calc-store-neg (v/-1)
	s p	var	29	calc-permanent-variable
	s r	var	29	v (recalled value)
	r 0-9			calc-recall-quick
a	s s	var	28,29	calc-store
a	s 0-9			calc-store-quick
a	s t	var	29	calc-store-into
a	t 0-9			calc-store-into-quick
	s u	var	29	calc-unstore
a	s x	var	29	calc-store-exchange
	s A	editing	30	calc-edit-AlgSimpRules


```
            s D   editing          30  calc-edit-Decls
            s E   editing          30  calc-edit-EvalRules
            s F   editing          30  calc-edit-FitRules
            s G   editing          30  calc-edit-GenCount
            s H   editing          30  calc-edit-Holidays
            s I   editing          30  calc-edit-IntegLimit
            s L   editing          30  calc-edit-LineStyles
            s P   editing          30  calc-edit-PointStyles
            s R   editing          30  calc-edit-PlotRejects
            s T   editing          30  calc-edit-TimeZone
            s U   editing          30  calc-edit-Units
            s X   editing          30  calc-edit-ExtSimpRules

    a       s +   var           29,47  calc-store-plus  (v+a)
    a       s -   var           29,47  calc-store-minus  (v-a)
    a       s *   var           29,47  calc-store-times  (v*a)
    a       s /   var           29,47  calc-store-div  (v/a)
    a       s ^   var           29,47  calc-store-power  (v^a)
    a       s |   var           29,47  calc-store-concat  (v|a)
            s &   var           29,47  calc-store-inv  (v^-1)
            s [   var           29,47  calc-store-decr  (v-1)
            s ]   var           29,47  calc-store-incr  (v-(-1))
  a b       s :                     2  assign(a,b)  a := b
    a       s =                     1  evalto(a,b)  a =>

            t [                     4  calc-trail-first
            t ]                     4  calc-trail-last
            t <                     4  calc-trail-scroll-left
            t >                     4  calc-trail-scroll-right
            t .                    12  calc-full-trail-vectors

            t b                     4  calc-trail-backward
            t d                 12,50  calc-trail-display
            t f                     4  calc-trail-forward
            t h                        calc-trail-here
            t i                        calc-trail-in
            t k                     4  calc-trail-kill
            t m   string               calc-trail-marker
            t n                     4  calc-trail-next
            t o                        calc-trail-out
            t p                     4  calc-trail-previous
            t r   string               calc-trail-isearch-backward
            t s   string               calc-trail-isearch-forward
            t y                     4  calc-trail-yank

    d       t C   oz, nz               tzconv(d,oz,nz)
```

d	oz	nz		t C	$	tzconv(d,oz,nz)
		d		t D		15 date(d)
		d		t I		4 incmonth(d,n)
		d		t J		16 julian(d,z)
		d		t M		17 newmonth(d,n)
				t N		16 now(z)
		d		t P	1	31 year(d)
		d		t P	2	31 month(d)
		d		t P	3	31 day(d)
		d		t P	4	31 hour(d)
		d		t P	5	31 minute(d)
		d		t P	6	31 second(d)
		d		t P	7	31 weekday(d)
		d		t P	8	31 yearday(d)
		d		t P	9	31 time(d)
		d		t U		16 unixtime(d,z)
		d		t W		17 newweek(d,w)
		d		t Y		17 newyear(d,n)
	a	b		t +		2 badd(a,b)
	a	b		t -		2 bsub(a,b)
				u a		12 calc-autorange-units
		a		u b		calc-base-units
		a		u c	units	18 calc-convert-units
	defn			u d	unit, descr	calc-define-unit
				u e		calc-explain-units
				u g	unit	calc-get-unit-definition
				u p		calc-permanent-units
		a		u r		calc-remove-units
		a		u s		usimplify(a)
		a		u t	units	18 calc-convert-temperature
				u u	unit	calc-undefine-unit
				u v		calc-enter-units-table
		a		u x		calc-extract-units
		a		u 0-9		calc-quick-units
	v1	v2		u C		20 vcov(v1,v2)
	v1	v2	I	u C		20 vpcov(v1,v2)
	v1	v2	H	u C		20 vcorr(v1,v2)
		v		u G		19 vgmean(v)
	a	b	H	u G		2 agmean(a,b)
		v		u M		19 vmean(v)
		v	I	u M		19 vmeane(v)
		v	H	u M		19 vmedian(v)
		v	I H	u M		19 vhmean(v)


```
    v       u N                      19  vmin(v)
    v       u S                      19  vsdev(v)
    v     I u S                      19  vpsdev(v)
    v     H u S                      19  vvar(v)
    v   I H u S                      19  vpvar(v)
            u V                          calc-view-units-table
    v       u X                      19  vmax(v)

    v       u +                      19  vsum(v)
    v       u *                      19  vprod(v)
    v       u #                      19  vcount(v)

            V (                      50  calc-vector-parens
            V {                      50  calc-vector-braces
            V [                      50  calc-vector-brackets
            V ]    ROCP              50  calc-matrix-brackets
            V ,                      50  calc-vector-commas
            V <                      50  calc-matrix-left-justify
            V =                      50  calc-matrix-center-justify
            V >                      50  calc-matrix-right-justify
            V /                   12,50  calc-break-vectors
            V .                   12,50  calc-full-vectors

  s t       V ^                       2  vint(s,t)
  s t       V -                       2  vdiff(s,t)
    s       V ~                       1  vcompl(s)
    s       V #                       1  vcard(s)
    s       V :                       1  vspan(s)
    s       V +                       1  rdup(s)

    m       V &                       1  inv(m)  1/m

    v       v a    n                      arrange(v,n)
    a       v b    n                      cvec(a,n)
    v       v c    n >0            21,31  mcol(v,n)
    v       v c    n <0               31  mrcol(v,-n)
    m       v c    0                  31  getdiag(m)
    v       v d                       25  diag(v,n)
  v m       v e                        2  vexp(v,m)
v m f     H v e                        2  vexp(v,m,f)
  v a       v f                       26  find(v,a,n)
    v       v h                        1  head(v)
    v     I v h                        1  tail(v)
    v     H v h                        1  rhead(v)
    v   I H v h                        1  rtail(v)
            v i    n                  31  idn(1,n)
```

		v	i	0	31	idn(1)
h	t		v	k	2	cons(h,t)
h	t	H	v	k	2	rcons(h,t)
v			v	l	1	vlen(v)
v		H	v	l	1	mdims(v)
v	m		v	m	2	vmask(v,m)
v			v	n	1	rnorm(v)
a	b	c	v	p	24	calc-pack
v			v	r	n >0	21,31	mrow(v,n)
v			v	r	n <0	31	mrrow(v,-n)
m			v	r	0	31	getdiag(m)
v	i	j	v	s		subvec(v,i,j)
v	i	j	I	v	s		rsubvec(v,i,j)
m			v	t	1	trn(m)
v			v	u	24	calc-unpack
v			v	v	1	rev(v)
			v	x	n	31	index(n)
n	s	i	C-u	v	x		index(n,s,i)
v			V	A	op	22	apply(op,v)
v1	v2		V	C	2	cross(v1,v2)
m			V	D	1	det(m)
s			V	E	1	venum(s)
s			V	F	1	vfloor(s)
v			V	G		grade(v)
v		I	V	G		rgrade(v)
v			V	H	n	31	histogram(v,n)
v	w	H	V	H	n	31	histogram(v,w,n)
v1	v2		V	I	mop aop	22	inner(mop,aop,v1,v2)
m			V	J	1	ctrn(m)
m			V	L	1	lud(m)
v			V	M	op	22,23	map(op,v)
v			V	N	1	cnorm(v)
v1	v2		V	O	op	22	outer(op,v1,v2)
v			V	R	op	22,23	reduce(op,v)
v		I	V	R	op	22,23	rreduce(op,v)
a	n	H	V	R	op	22	nest(op,a,n)
a		I	H	V	R	op	22	fixp(op,a)
v			V	S		sort(v)
v		I	V	S		rsort(v)
m			V	T	1	tr(m)
v			V	U	op	22	accum(op,v)
v		I	V	U	op	22	raccum(op,v)
a	n	H	V	U	op	22	anest(op,a,n)
a		I	H	V	U	op	22	afixp(op,a)
s	t		V	V	2	vunion(s,t)

s t	V X		2	vxor(s,t)
	Y			user commands
	z			user commands
c	Z [		45	calc-kbd-if
c	Z |		45	calc-kbd-else-if
	Z :			calc-kbd-else
	Z]			calc-kbd-end-if
	Z {		4	calc-kbd-loop
c	Z /		45	calc-kbd-break
	Z }			calc-kbd-end-loop
n	Z <			calc-kbd-repeat
	Z >			calc-kbd-end-repeat
n m	Z (			calc-kbd-for
s	Z)			calc-kbd-end-for
	Z C-g			cancel if/loop command
	Z `			calc-kbd-push
	Z '			calc-kbd-pop
a	Z =	message	28	calc-kbd-report
	Z #	prompt		calc-kbd-query
comp	Z C	func, args	50	calc-user-define-composition
	Z D	key, command		calc-user-define
	Z E	key, editing	30	calc-user-define-edit
defn	Z F	k, c, f, a, n	28	calc-user-define-formula
	Z G	key		calc-get-user-defn
	Z I			calc-user-define-invocation
	Z K	key, command		calc-user-define-kbd-macro
	Z P	key		calc-user-define-permanent
	Z S		30	calc-edit-user-syntax
	Z T		12	calc-timing
	Z U	key		calc-user-undefine

NOTES

1. Positive prefix arguments apply to n stack entries. Negative prefix arguments apply to the -nth stack entry. A prefix of zero applies to the entire stack. (For LFD and M-DEL, the meaning of the sign is reversed.)
2. Positive prefix arguments apply to n stack entries. Negative prefix arguments apply to the top stack entry and the next -n stack entries.

3. Positive prefix arguments rotate top n stack entries by one. Negative prefix arguments rotate th entire stack by -n. A prefix of zero reverses the entire stack.
4. Prefix argument specifies a repeat count or distance.
5. Positive prefix arguments specify a precision p. Negative prefix arguments reduce the current precision by -p.
6. A prefix argument is interpreted as an additional step-size parameter. A plain C-u prefix means to prompt for the step size.
7. A prefix argument specifies simplification level and depth. 1=Default, 2=like a s, 3=like a e.
8. A negative prefix operates only on the top level of the input formula.
9. Positive prefix arguments specify a word size of w bits, unsigned. Negative prefix arguments specify a word size of w bits, signed.
10. Prefix arguments specify the shift amount n. The w argument cannot be specified in the keyboard version of this command.
11. From the keyboard, d is omitted and defaults to zero.
12. Mode is toggled; a positive prefix always sets the mode, and a negative prefix always clears the mode.
13. Some prefix argument values provide special variations of the mode.
14. A prefix argument, if any, is used for m instead of taking m from the stack. M may take any of these values: {@advance@tableindent10pt

    Integer

        Random integer in the interval [0 .. m).

    Float

        Random floating-point number in the interval [0 .. m).

    0.0

        Gaussian with mean 1 and standard deviation 0.

    Error form

        Gaussian with specified mean and standard deviation.

    Interval

        Random integer or floating-point number in that interval.

    Vector

        Random element from the vector.

    }
15. A prefix argument from 1 to 6 specifies number of date components to remove from the stack. See section Date Conversions.
16. A prefix argument specifies a time zone; C-u says to take the time zone number or name from the top of the stack. See section Time Zones.
17. A prefix argument specifies a day number (0-6, 0-31, or 0-366).
18. If the input has no units, you will be prompted for both the old and the new units.

19. With a prefix argument, collect that many stack entries to form the input data set. Each entry may be a single value or a vector of values.
20. With a prefix argument of 1, take a single @c{$N\times2$} Nx2 matrix from the stack instead of two separate data vectors.
21. The row or column number n may be given as a numeric prefix argument instead. A plain C-u prefix says to take n from the top of the stack. If n is a vector or interval, a subvector/submatrix of the input is created.
22. The op prompt can be answered with the key sequence for the desired function, or with x or z followed by a function name, or with $ to take a formula from the top of the stack, or with ' and a typed formula. In the last two cases, the formula may be a nameless function like `<#1+#2>' or `<x, y : x+y>', or it may include $, $$, etc. (where $ will correspond to the last argument of the created function), or otherwise you will be prompted for an argument list. The number of vectors popped from the stack by V M depends on the number of arguments of the function.
23. One of the mapping direction keys _ (horizontal, i.e., map by rows or reduce across), : (vertical, i.e., map by columns or reduce down), or = (map or reduce by rows) may be used before entering op; these modify the function name by adding the letter `r` for "rows," `c` for "columns," `a` for "across," or `d` for "down."
24. The prefix argument specifies a packing mode. A nonnegative mode is the number of items (for v p) or the number of levels (for v u). A negative mode is as described below. With no prefix argument, the mode is taken from the top of the stack and may be an integer or a vector of integers. {@advance@tableindent-20pt

    -1
        (2) Rectangular complex number.

    -2
        (2) Polar complex number.

    -3
        (3) HMS form.

    -4
        (2) Error form.

    -5
        (2) Modulo form.

    -6
        (2) Closed interval.

    -7
        (2) Closed .. open interval.

    -8
        (2) Open .. closed interval.

    -9
        (2) Open interval.

-10

(2) Fraction.

-11

(2) Float with integer mantissa.

-12

(2) Float with mantissa in [1 .. 10).

-13

(1) Date form (using date numbers).

-14

(3) Date form (using year, month, day).

-15

(6) Date form (using year, month, day, hour, minute, second).

}

25. A prefix argument specifies the size n of the matrix. With no prefix argument, n is omitted and the size is inferred from the input vector.
26. The prefix argument specifies the starting position n (default 1).
27. Cursor position within stack buffer affects this command.
28. Arguments are not actually removed from the stack by this command.
29. Variable name may be a single digit or a full name.
30. Editing occurs in a separate buffer. Press M-# M-# (or C-c C-c, LFD, or in some cases RET) to finish the edit, or press M-# x to cancel the edit. The LFD key prevents evaluation of the result of the edit.
31. The number prompted for can also be provided as a prefix argument.
32. Press this key a second time to cancel the prefix.
33. With a negative prefix, deactivate all formulas. With a positive prefix, deactivate and then reactivate from scratch.
34. Default is to scan for nearest formula delimiter symbols. With a prefix of zero, formula is delimited by mark and point. With a non-zero prefix, formula is delimited by scanning forward or backward by that many lines.
35. Parse the region between point and mark as a vector. A nonzero prefix parses n lines before or after point as a vector. A zero prefix parses the current line as a vector. A C-u prefix parses the region between point and mark as a single formula.
36. Parse the rectangle defined by point and mark as a matrix. A positive prefix n divides the rectangle into columns of width n. A zero or C-u prefix parses each line as one formula. A negative prefix suppresses special treatment of bracketed portions of a line.
37. A numeric prefix causes the current language mode to be ignored.
38. Responding to a prompt with a blank line answers that and all later prompts by popping additional stack entries.
39. Answer for v may also be of the form v = v_0 or v - v_0.

40. With a positive prefix argument, stack contains many y's and one common x. With a zero prefix, stack contains a vector of ys and a common x. With a negative prefix, stack contains many [x,y] vectors. (For 3D plots, substitute z for y and x,y for x.)
41. With any prefix argument, all curves in the graph are deleted.
42. With a positive prefix, refines an existing plot with more data points. With a negative prefix, forces recomputation of the plot data.
43. With any prefix argument, set the default value instead of the value for this graph.
44. With a negative prefix argument, set the value for the printer.
45. Condition is considered "true" if it is a nonzero real or complex number, or a formula whose value is known to be nonzero; it is "false" otherwise.
46. Several formulas separated by commas are pushed as multiple stack entries. Trailing),], }, >, and " delimiters may be omitted. The notation $$$ refers to the value in stack level three, and causes the formula to replace the top three stack levels. The notation $3 refers to stack level three without causing that value to be removed from the stack. Use LFD in place of RET to prevent evaluation; use M-= in place of RET to evaluate variables.
47. The variable is replaced by the formula shown on the right. The Inverse flag reverses the order of the operands, e.g., I s - x assigns @c{$x \coloneq a-x$} x := a-x.
48. Press ? repeatedly to see how to choose a model. Answer the variables prompt with iv or iv;pv to specify independent and parameter variables. A positive prefix argument takes *N+1* vectors from the stack; a zero prefix takes a matrix and a vector from the stack.
49. With a plain C-u prefix, replace the current region of the destination buffer with the yanked text instead of inserting.
50. All stack entries are reformatted; the H prefix inhibits this. The I prefix sets the mode temporarily, redraws the top stack entry, then restores the original setting of the mode.
51. A negative prefix sets the default 3D resolution instead of the default 2D resolution.
52. This grabs a vector of the form [prec, wsize, ssize, radix, flfmt, ang, frac, symb, polar, matrix, simp, inf]. A prefix argument from 1 to 12 grabs the nth mode value only.

(Space is provided below for you to keep your own written notes.) @endgroup

Go to the previous, next section.

Go to the previous, next section.

Index of Key Sequences

(

- (

)

-)

*

- *

+

- +

,

- ,

-

- -

.

- .
- ..

/

- /

0

- 0-9


TOC

]

-]

^

- ^

_

- _

`

- `

a

- A
- a !
- a "
- a #
- a %
- a &
- A (vectors)
- a *
- a +
- a -
- a .
- a /
- a :
- a <
- a =
- a >
- a [

b

c

d

e

f

g

h

i

j

k

l

- L
- LFD

m

- M
- m (HMS forms)
- M (modulo forms)
- m a
- m A
- m B
- m C
- m d
- m D
- m E
- m F
- m f
- m g
- m h
- m i
- m m
- m N
- m O
- m p
- m R
- m r
- m S
- m s
- m t
- m U
- m v
- m w

n

o

p

q

r

S

t

u

V

w

x

y

z

Go to the previous, next section.

Go to the previous, next section.

Index of Calculator Commands

Since all Calculator commands begin with the prefix `calc-', the x key has been provided as a variant of M-x which automatically types `calc-' for you. Thus, x last-args is short for M-x calc-last-args.

a

c

d

f

q

r

Go to the previous, next section.

Go to the previous, next section.

Index of Algebraic Functions

This is a list of built-in functions and operators usable in algebraic expressions. Their full Lisp names are derived by adding the prefix `calcFunc-', as in `calcFunc-sqrt'. All functions except those noted with "*" have corresponding Calc keystrokes and can also be found in the Calc Summary.

!

- !
- !!
- !!!
- !=

%

- %

&

- &&
- &&&

*

- *

+

- +
- +/-

-

- -


TOC

/

:

<

=

>

?

\

^

- ^

_

- _

a

- abs
- abs (vectors)
- abssqr
- accum
- acute
- add
- afixp
- agmean
- alog
- an
- and
- anest
- apart
- append
- apply
- apply (rewrites)
- arccos
- arccosh
- arcsin
- arcsincos
- arcsinh
- arctan
- arctan2
- arctanh

b

c

d

e

f

g

h

i

j

- julian

l

- lambda
- land
- lcm
- leq
- let
- lin
- linnt
- ln
- lnot
- lnp1
- log
- log10
- lor
- lsh
- lt
- ltpb
- ltpc
- ltpf
- ltpn
- ltpp
- ltpt
- lud

m

- makemod
- mant
- map

n

o

p

q

r

S

t

u

V

w

x

y

|

TOC

- |||

Go to the previous, next section.

Go to the previous, next section.

Concept Index

"

.

=

a

b

c

d

e

f

TOC

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

V

W

Go to the previous, next section.

Go to the previous, next section.

Index of Variables

The variables in this list that do not contain dashes are accessible as Calc variables. Add a `var-' prefix to get the name of the corresponding Lisp variable.

The remaining variables are Lisp variables suitable for `setq`ing in your `.emacs' file.

a

c

d

e

f

g

h

i

j

l

m

n

p

q

r

t

u

Go to the previous, next section.

Go to the previous section.

Index of Lisp Math Functions

The following functions are meant to be used with `defmath`, not `defun` definitions. For names that do not start with `calc-', the corresponding full Lisp name is derived by adding a prefix of `math-'.

a

b

c

d

e

f

h

i

l

m

n

o

p

q

r

s

t

u

v

w

z

Go to the previous section.

Common Lisp Extensions

Common Lisp Extensions

Overview

Common Lisp is a huge language, and Common Lisp systems tend to be massive and extremely complex. Emacs Lisp, by contrast, is rather minimalist in the choice of Lisp features it offers the programmer. As Emacs Lisp programmers have grown in number, and the applications they write have grown more ambitious, it has become clear that Emacs Lisp could benefit from many of the conveniences of Common Lisp.

The CL package adds a number of Common Lisp functions and control structures to Emacs Lisp. While not a 100% complete implementation of Common Lisp, CL adds enough functionality to make Emacs Lisp programming significantly more convenient.

Some Common Lisp features have been omitted from this package for various reasons:

- Some features are too complex or bulky relative to their benefit to Emacs Lisp programmers. CLOS and Common Lisp streams are fine examples of this group.
- Other features cannot be implemented without modification to the Emacs Lisp interpreter itself, such as multiple return values, lexical scoping, case-insensitive symbols, and complex numbers. The CL package generally makes no attempt to emulate these features.
- Some features conflict with existing things in Emacs Lisp. For example, Emacs' `assoc` function is incompatible with the Common Lisp `assoc`. In such cases, this package usually adds the suffix `*' to the function name of the Common Lisp version of the function (e.g., `assoc*`).

The package described here was written by Dave Gillespie, `daveg@synaptics.com'. It is a total rewrite of the original 1986 `cl.el' package by Cesar Quiroz. Most features of the the Quiroz package have been retained; any incompatibilities are noted in the descriptions below. Care has been taken in this version to ensure that each function is defined efficiently, concisely, and with minimal impact on the rest of the Emacs environment.

Usage

Lisp code that uses features from the CL package should include at the beginning:

```
(require 'cl)
```

If you want to ensure that the new (Gillespie) version of CL is the one that is present, add an additional `(require 'cl-19)` call:

```
(require 'cl)
(require 'cl-19)
```

The second call will fail (with "`cl-19.el' not found") if the old `cl.el' package was in use.

It is safe to arrange to load CL at all times, e.g., in your `.emacs' file. But it's a good idea, for portability, to `(require 'cl)` in your code even if you do this.

Organization

The Common Lisp package is organized into four files:

`cl.el'
: This is the "main" file, which contains basic functions and information about the package. This file is relatively compact--about 700 lines.

`cl-extra.el'
: This file contains the larger, more complex or unusual functions. It is kept separate so that packages which only want to use Common Lisp fundamentals like the `cadr` function won't need to pay the overhead of loading the more advanced functions.

`cl-seq.el'
: This file contains most of the advanced functions for operating on sequences or lists, such as `delete-if` and `assoc*`.

`cl-macs.el'
: This file contains the features of the packages which are macros instead of functions. Macros expand when the caller is compiled, not when it is run, so the macros generally only need to be present when the byte-compiler is running (or when the macros are used in uncompiled code such as a `.emacs' file). Most of the macros of this package are isolated in `cl-macs.el' so that they won't take up memory unless you are compiling.

The file `cl.el' includes all necessary `autoload` commands for the functions and macros in the other three files. All you have to do is `(require 'cl)`, and `cl.el' will take care of pulling in the other files when they are needed.

There is another file, `cl-compat.el', which defines some routines from the older `cl.el' package that are no longer present in the new package. This includes internal routines like `setelt` and `zip-lists`, deprecated features like `defkeyword`, and an emulation of the old-style multiple-values feature. See section Old CL Compatibility.

Installation

Installation of the CL package is simple: Just put the byte-compiled files `cl.elc', `cl-extra.elc', `cl-seq.elc', `cl-macs.elc', and `cl-compat.elc' into a directory on your `load-path`.

There are no special requirements to compile this package: The files do not have to be loaded before they are compiled, nor do they need to be compiled in any particular order.

You may choose to put the files into your main `lisp/' directory, replacing the original `cl.el' file there. Or, you could put them into a directory that comes before `lisp/' on your `load-path` so that the old `cl.el' is effectively hidden.

Also, format the `cl.texinfo' file and put the resulting Info files in the `info/' directory or another suitable place.

You may instead wish to leave this package's components all in their own directory, and then add this

directory to your `load-path` and (Emacs 19 only) `Info-directory-list`. Add the directory to the front of the list so the old CL package and its documentation are hidden.

Naming Conventions

Except where noted, all functions defined by this package have the same names and calling conventions as their Common Lisp counterparts.

Following is a complete list of functions whose names were changed from Common Lisp, usually to avoid conflicts with Emacs. In each case, a `*' has been appended to the Common Lisp name to obtain the Emacs name:

```
defun*          defsubst*       defmacro*       function*
member*         assoc*          rassoc*         get*
remove*         delete*         mapcar*         sort*
floor*          ceiling*        truncate*       round*
mod*            rem*            random*
```

Internal function and variable names in the package are prefixed by `cl-`. Here is a complete list of functions *not* prefixed by `cl-` which were not taken from Common Lisp:

```
member          delete          remove          remq
rassoc          floatp-safe     lexical-let     lexical-let*
callf           callf2          letf            letf*
defsubst*       defalias        add-hook        eval-when-compile
```

(Most of these are Emacs 19 features provided to Emacs 18 users, or introduced, like `remq`, for reasons of symmetry with similar features.)

The following simple functions and macros are defined in `cl.el'; they do not cause other components like `cl-extra' to be loaded.

```
eql             floatp-safe     abs             endp
evenp           oddp            plusp           minusp
last            butlast         nbutlast        caar .. cddddr
list*           ldiff           rest            first .. tenth
member [1]      copy-list       subst           mapcar* [2]
adjoin [3]      acons           pairlis         when
unless          pop [4]         push [4]        pushnew [3,4]
incf [4]        decf [4]        proclaim        declaim
add-hook
```

[1] This is the Emacs 19-compatible function, not `member*`.

[2] Only for one sequence argument or two list arguments.

[3] Only if `:test` is `eq`, `equal`, or unspecified, and `:key` is not used.

[4] Only when place is a plain variable name.

@chapno=4

Program Structure

This section describes features of the CL package which have to do with programs as a whole: advanced argument lists for functions, and the `eval-when` construct.

@secno=1

Argument Lists

Emacs Lisp's notation for argument lists of functions is a subset of the Common Lisp notation. As well as the familiar `&optional` and `&rest` markers, Common Lisp allows you to specify default values for optional arguments, and it provides the additional markers `&key` and `&aux`.

Since argument parsing is built-in to Emacs, there is no way for this package to implement Common Lisp argument lists seamlessly. Instead, this package defines alternates for several Lisp forms which you must use if you need Common Lisp argument lists.

Special Form: **defun*** *name arglist body...*

This form is identical to the regular `defun` form, except that arglist is allowed to be a full Common Lisp argument list. Also, the function body is enclosed in an implicit block called name; see section Blocks and Exits.

Special Form: **defsubst*** *name arglist body...*

This is just like `defun*`, except that the function that is defined is automatically proclaimed `inline`, i.e., calls to it may be expanded into in-line code by the byte compiler. This is analogous to the `defsubst` form in Emacs 19; `defsubst*` uses a different method (compiler macros) which works in all version of Emacs, and also generates somewhat more efficient inline expansions. In particular, `defsubst*` arranges for the processing of keyword arguments, default values, etc., to be done at compile-time whenever possible.

Special Form: **defmacro*** *name arglist body...*

This is identical to the regular `defmacro` form, except that arglist is allowed to be a full Common Lisp argument list. The `&environment` keyword is supported as described in Steele. The `&whole` keyword is supported only within destructured lists (see below); top-level `&whole` cannot be implemented with the current Emacs Lisp interpreter. The macro expander body is enclosed in an implicit block called name.

Special Form: **function*** *symbol-or-lambda*

This is identical to the regular `function` form, except that if the argument is a `lambda` form then that form may use a full Common Lisp argument list.

Also, all forms (such as `defsetf` and `flet`) defined in this package that include arglists in their syntax allow full Common Lisp argument lists.

Note that it is *not* necessary to use `defun*` in order to have access to most CL features in your function.

These features are always present; `defun*`'s only difference from `defun` is its more flexible argument and its implicit block.

The full form of a Common Lisp argument list is

```
(var...
 &optional (var initform svar)...
 &rest var
 &key ((keyword var) initform svar)...
 &aux (var initform)...)
```

Each of the five argument list sections is optional. The svar, initform, and keyword parts are optional; if they are omitted, then `(var)' may be written simply `var'.

The first section consists of zero or more required arguments. These arguments must always be specified in a call to the function; there is no difference between Emacs Lisp and Common Lisp as far as required arguments are concerned.

The second section consists of optional arguments. These arguments may be specified in the function call; if they are not, initform specifies the default value used for the argument. (No initform means to use `nil` as the default.) The initform is evaluated with the bindings for the preceding arguments already established; `(a &optional (b (1+ a)))` matches one or two arguments, with the second argument defaulting to one plus the first argument. If the svar is specified, it is an auxiliary variable which is bound to `t` if the optional argument was specified, or to `nil` if the argument was omitted. If you don't use an svar, then there will be no way for your function to tell whether it was called with no argument, or with the default value passed explicitly as an argument.

The third section consists of a single rest argument. If more arguments were passed to the function than are accounted for by the required and optional arguments, those extra arguments are collected into a list and bound to the "rest" argument variable. Common Lisp's `&rest` is equivalent to that of Emacs Lisp. Common Lisp accepts `&body` as a synonym for `&rest` in macro contexts; this package accepts it all the time.

The fourth section consists of keyword arguments. These are optional arguments which are specified by name rather than positionally in the argument list. For example,

```
(defun* foo (a &optional b &key c d (e 17)))
```

defines a function which may be called with one, two, or more arguments. The first two arguments are bound to `a` and `b` in the usual way. The remaining arguments must be pairs of the form `:c`, `:d`, or `:e` followed by the value to be bound to the corresponding argument variable. (Symbols whose names begin with a colon are called keywords, and they are self-quoting in the same way as `nil` and `t`.)

For example, the call `(foo 1 2 :d 3 :c 4)` sets the five arguments to 1, 2, 4, 3, and 17, respectively. If the same keyword appears more than once in the function call, the first occurrence takes precedence over the later ones. Note that it is not possible to specify keyword arguments without specifying the optional argument `b` as well, since `(foo 1 :c 2)` would bind `b` to the keyword `:c`, then signal an error because `2` is not a valid keyword.

If a keyword symbol is explicitly specified in the argument list as shown in the above diagram, then that keyword will be used instead of just the variable name prefixed with a colon. You can specify a keyword

symbol which does not begin with a colon at all, but such symbols will not be self-quoting; you will have to quote them explicitly with an apostrophe in the function call.

Ordinarily it is an error to pass an unrecognized keyword to a function, e.g., `(foo 1 2 :c 3 :goober 4)`. You can ask Lisp to ignore unrecognized keywords, either by adding the marker `&allow-other-keys` after the keyword section of the argument list, or by specifying an `:allow-other-keys` argument in the call whose value is non-`nil`. If the function uses both `&rest` and `&key` at the same time, the "rest" argument is bound to the keyword list as it appears in the call. For example:

```
(defun* find-thing (thing &rest rest &key need &allow-other-keys)
  (or (apply 'member* thing thing-list :allow-other-keys t rest)
      (if need (error "Thing not found"))))
```

This function takes a `:need` keyword argument, but also accepts other keyword arguments which are passed on to the `member*` function. `allow-other-keys` is used to keep both `find-thing` and `member*` from complaining about each others' keywords in the arguments.

In Common Lisp, keywords are recognized by the Lisp parser itself and treated as special entities. In Emacs, keywords are just symbols whose names begin with colons, which `defun*` has arranged to set equal to themselves so that they will essentially be self-quoting.

As a (significant) performance optimization, this package implements the scan for keyword arguments by calling `memq` to search for keywords in a "rest" argument. Technically speaking, this is incorrect, since `memq` looks at the odd-numbered values as well as the even-numbered keywords. The net effect is that if you happen to pass a keyword symbol as the *value* of another keyword argument, where that keyword symbol happens to equal the name of a valid keyword argument of the same function, then the keyword parser will become confused. This minor bug can only affect you if you use keyword symbols as general-purpose data in your program; this practice is strongly discouraged in Emacs Lisp.

The fifth section of the argument list consists of auxiliary variables. These are not really arguments at all, but simply variables which are bound to `nil` or to the specified initforms during execution of the function. There is no difference between the following two functions, except for a matter of stylistic taste:

```
(defun* foo (a b &aux (c (+ a b)) d)
  body)

(defun* foo (a b)
  (let ((c (+ a b)) d)
    body))
```

Argument lists support destructuring. In Common Lisp, destructuring is only allowed with `defmacro`; this package allows it with `defun*` and other argument lists as well. In destructuring, any argument variable (var in the above diagram) can be replaced by a list of variables, or more generally, a recursive argument list. The corresponding argument value must be a list whose elements match this recursive argument list. For example:

```
(defmacro* dolist ((var listform &optional resultform)
                   &rest body)
  ...)
```

This says that the first argument of `dolist` must be a list of two or three items; if there are other arguments as well as this list, they are stored in `body`. All features allowed in regular argument lists are allowed in these recursive argument lists. In addition, the clause `&whole var' is allowed at the front of a recursive argument list. It binds var to the whole list being matched; thus `(&whole all a b)` matches a list of two things, with `a` bound to the first thing, `b` bound to the second thing, and `all` bound to the list itself. (Common Lisp allows `&whole` in top-level `defmacro` argument lists as well, but Emacs Lisp does not support this usage.)

One last feature of destructuring is that the argument list may be dotted, so that the argument list `(a b . c)` is functionally equivalent to `(a b &rest c)`.

If the optimization quality `safety` is set to 0 (see section Declarations), error checking for wrong number of arguments and invalid keyword arguments is disabled. By default, argument lists are rigorously checked.

Time of Evaluation

Normally, the byte-compiler does not actually execute the forms in a file it compiles. For example, if a file contains `(setq foo t)`, the act of compiling it will not actually set `foo` to `t`. This is true even if the `setq` was a top-level form (i.e., not enclosed in a `defun` or other form). Sometimes, though, you would like to have certain top-level forms evaluated at compile-time. For example, the compiler effectively evaluates `defmacro` forms at compile-time so that later parts of the file can refer to the macros that are defined.

Special Form: **eval-when** *(situations...) forms...*

This form controls when the body forms are evaluated. The situations list may contain any set of the symbols `compile`, `load`, and `eval` (or their long-winded ANSI equivalents, `:compile-toplevel`, `:load-toplevel`, and `:execute`).

The `eval-when` form is handled differently depending on whether or not it is being compiled as a top-level form. Specifically, it gets special treatment if it is being compiled by a command such as `byte-compile-file` which compiles files or buffers of code, and it appears either literally at the top level of the file or inside a top-level `progn`.

For compiled top-level `eval-when`s, the body forms are executed at compile-time if `compile` is in the situations list, and the forms are written out to the file (to be executed at load-time) if `load` is in the situations list.

For non-compiled-top-level forms, only the `eval` situation is relevant. (This includes forms executed by the interpreter, forms compiled with `byte-compile` rather than `byte-compile-file`, and non-top-level forms.) The `eval-when` acts like a `progn` if `eval` is specified, and like `nil` (ignoring the body forms) if not.

The rules become more subtle when `eval-when`s are nested; consult Steele (second edition) for the gruesome details (and some gruesome examples).

Some simple examples:

```
;; Top-level forms in foo.el:
(eval-when (compile)           (setq foo1 'bar))
(eval-when (load)              (setq foo2 'bar))
```


```
(eval-when (compile load)      (setq foo3 'bar))
(eval-when (eval)              (setq foo4 'bar))
(eval-when (eval compile)      (setq foo5 'bar))
(eval-when (eval load)         (setq foo6 'bar))
(eval-when (eval compile load) (setq foo7 'bar))
```

When `foo.el' is compiled, these variables will be set during the compilation itself:

```
foo1  foo3  foo5  foo7      ; `compile'
```

When `foo.elc' is loaded, these variables will be set:

```
foo2  foo3  foo6  foo7      ; `load'
```

And if `foo.el' is loaded uncompiled, these variables will be set:

```
foo4  foo5  foo6  foo7      ; `eval'
```

If these seven `eval-when`s had been, say, inside a `defun`, then the first three would have been equivalent to `nil` and the last four would have been equivalent to the corresponding `setq`s.

Note that `(eval-when (load eval) ...)` is equivalent to `(progn ...)` in all contexts. The compiler treats certain top-level forms, like `defmacro` (sort-of) and `require`, as if they were wrapped in `(eval-when (compile load eval) ...)`.

Emacs 19 includes two special forms related to `eval-when`. One of these, `eval-when-compile`, is not quite equivalent to any `eval-when` construct and is described below. This package defines a version of `eval-when-compile` for the benefit of Emacs 18 users.

The other form, `(eval-and-compile ...)`, is exactly equivalent to `(eval-when (compile load eval) ...)' and so is not itself defined by this package.

Special Form: **eval-when-compile** *forms...*

The forms are evaluated at compile-time; at execution time, this form acts like a quoted constant of the resulting value. Used at top-level, `eval-when-compile` is just like `eval-when (compile eval)'. In other contexts, `eval-when-compile` allows code to be evaluated once at compile-time for efficiency or other reasons.

This form is similar to the `#.' syntax of true Common Lisp.

Special Form: **load-time-value** *form*

The form is evaluated at load-time; at execution time, this form acts like a quoted constant of the resulting value.

Early Common Lisp had a `#,' syntax that was similar to this, but ANSI Common Lisp replaced it with `load-time-value` and gave it more well-defined semantics.

In a compiled file, `load-time-value` arranges for form to be evaluated when the `.elc' file is loaded and then used as if it were a quoted constant. In code compiled by `byte-compile` rather than `byte-compile-file`, the effect is identical to `eval-when-compile`. In uncompiled code, both

`eval-when-compile` and `load-time-value` act exactly like `progn`.

```
(defun report ()
  (insert "This function was executed on: "
          (current-time-string)
          ", compiled on: "
          (eval-when-compile (current-time-string))
          ;; or '#.(current-time-string) in real Common Lisp
          ", and loaded on: "
          (load-time-value (current-time-string))))
```

Byte-compiled, the above defun will result in the following code (or its compiled equivalent, of course) in the `.elc' file:

```
(setq --temp-- (current-time-string))
(defun report ()
  (insert "This function was executed on: "
          (current-time-string)
          ", compiled on: "
          '"Wed Jun 23 18:33:43 1993"
          ", and loaded on: "
          --temp--))
```

Function Aliases

This section describes a feature from GNU Emacs 19 which this package makes available in other versions of Emacs.

Function: **defalias** *symbol function*

This function sets symbol's function cell to function. It is equivalent to `fset`, except that in GNU Emacs 19 it also records the setting in `load-history` so that it can be undone by a later `unload-feature`.

In other versions of Emacs, `defalias` is a synonym for `fset`.

Predicates

This section describes functions for testing whether various facts are true or false.

Type Predicates

The CL package defines a version of the Common Lisp `typep` predicate.

Function: **typep** *object type*

Check if object is of type type, where type is a (quoted) type name of the sort used by Common Lisp. For

example, `(typep foo 'integer)` is equivalent to `(integerp foo)`.

The type argument to the above function is either a symbol or a list beginning with a symbol.

- If the type name is a symbol, Emacs appends `-p' to the symbol name to form the name of a predicate function for testing the type. (Built-in predicates whose names end in `p' rather than `-p' are used when appropriate.)
- The type symbol `t` stands for the union of all types. `(typep object t)` is always true. Likewise, the type symbol `nil` stands for nothing at all, and `(typep object nil)` is always false.
- The type symbol `null` represents the symbol `nil`. Thus `(typep object 'null)` is equivalent to `(null object)`.
- The type symbol `real` is a synonym for `number`, and `fixnum` is a synonym for `integer`.
- The type symbols `character` and `string-char` match integers in the range from 0 to 255.
- The type symbol `float` uses the `floatp-safe` predicate defined by this package rather than `floatp`, so it will work correctly even in Emacs versions without floating-point support.
- The type list `(integer low high)` represents all integers between low and high, inclusive. Either bound may be a list of a single integer to specify an exclusive limit, or a `*` to specify no limit. The type `(integer * *)` is thus equivalent to `integer`.
- Likewise, lists beginning with `float`, `real`, or `number` represent numbers of that type falling in a particular range.
- Lists beginning with `and`, `or`, and `not` form combinations of types. For example, `(or integer (float 0 *))` represents all objects that are integers or non-negative floats.
- Lists beginning with `member` or `member*` represent objects `eql` to any of the following values. For example, `(member 1 2 3 4)` is equivalent to `(integer 1 4)`, and `(member nil)` is equivalent to `null`.
- Lists of the form `(satisfies predicate)` represent all objects for which predicate returns true when called with that object as an argument.

The following function and macro (not technically predicates) are related to `typep`.

Function: **coerce** *object type*

This function attempts to convert object to the specified type. If object is already of that type as determined by `typep`, it is simply returned. Otherwise, certain types of conversions will be made: If type is any sequence type (`string`, `list`, etc.) then object will be converted to that type if possible. If type is `character`, then strings of length one and symbols with one-character names can be coerced. If type is `float`, then integers can be coerced in versions of Emacs that support floats. In all other circumstances, `coerce` signals an error.

Special Form: **deftype** *name arglist forms...*

This macro defines a new type called name. It is similar to `defmacro` in many ways; when name is encountered as a type name, the body forms are evaluated and should return a type specifier that is equivalent to the type. The arglist is a Common Lisp argument list of the sort accepted by `defmacro*`. The type specifier `(name args...)' is expanded by calling the expander with those arguments; the type symbol `name' is expanded by calling the expander with no arguments. The arglist is processed the same as for `defmacro*` except that optional arguments without explicit defaults use `*` instead of `nil` as the "default" default. Some examples:


```
(deftype null () '(satisfies null))    ; predefined
(deftype list () '(or null cons))      ; predefined
(deftype unsigned-byte (&optional bits)
  (list 'integer 0 (if (eq bits '*) bits (1- (lsh 1 bits)))))
(unsigned-byte 8)  ==  (integer 0 255)
(unsigned-byte)  ==  (integer 0 *)
unsigned-byte  ==  (integer 0 *)
```

The last example shows how the Common Lisp `unsigned-byte` type specifier could be implemented if desired; this package does not implement `unsigned-byte` by default.

The `typecase` and `check-type` macros also use type names. See section Conditionals. See section Assertions and Errors. The `map`, `concatenate`, and `merge` functions take type-name arguments to specify the type of sequence to return. See section Sequences.

Equality Predicates

This package defines two Common Lisp predicates, `eql` and `equalp`.

Function: **eql** *a b*

This function is almost the same as `eq`, except that if a and b are numbers of the same type, it compares them for numeric equality (as if by `equal` instead of `eq`). This makes a difference only for versions of Emacs that are compiled with floating-point support, such as Emacs 19. Emacs floats are allocated objects just like cons cells, which means that `(eq 3.0 3.0)` will not necessarily be true--if the two `3.0`s were allocated separately, the pointers will be different even though the numbers are the same. But `(eql 3.0 3.0)` will always be true.

The types of the arguments must match, so `(eql 3 3.0)` is still false.

Note that Emacs integers are "direct" rather than allocated, which basically means `(eq 3 3)` will always be true. Thus `eq` and `eql` behave differently only if floating-point numbers are involved, and are indistinguishable on Emacs versions that don't support floats.

There is a slight inconsistency with Common Lisp in the treatment of positive and negative zeros. Some machines, notably those with IEEE standard arithmetic, represent `+0` and `-0` as distinct values. Normally this doesn't matter because the standard specifies that `(= 0.0 -0.0)` should always be true, and this is indeed what Emacs Lisp and Common Lisp do. But the Common Lisp standard states that `(eql 0.0 -0.0)` and `(equal 0.0 -0.0)` should be false on IEEE-like machines; Emacs Lisp does not do this, and in fact the only known way to distinguish between the two zeros in Emacs Lisp is to `format` them and check for a minus sign.

Function: **equalp** *a b*

This function is a more flexible version of `equal`. In particular, it compares strings case-insensitively, and it compares numbers without regard to type (so that `(equalp 3 3.0)` is true). Vectors and conses are compared recursively. All other objects are compared as if by `equal`.

This function differs from Common Lisp `equalp` in several respects. First, Common Lisp's `equalp` also

compares *characters* case-insensitively, which would be impractical in this package since Emacs does not distinguish between integers and characters. In keeping with the idea that strings are less vector-like in Emacs Lisp, this package's `equalp` also will not compare strings against vectors of integers. Finally, Common Lisp's `equalp` compares hash tables without regard to ordering, whereas this package simply compares hash tables in terms of their underlying structure (which means vectors for Lucid Emacs 19 hash tables, or lists for other hash tables).

Also note that the Common Lisp functions `member` and `assoc` use `eql` to compare elements, whereas Emacs Lisp follows the MacLisp tradition and uses `equal` for these two functions. In Emacs, use `member*` and `assoc*` to get functions which use `eql` for comparisons.

Control Structure

The features described in the following sections implement various advanced control structures, including the powerful `setf` facility and a number of looping and conditional constructs.

Assignment

The `psetq` form is just like `setq`, except that multiple assignments are done in parallel rather than sequentially.

Special Form: **psetq** *[symbol form]...*

This special form (actually a macro) is used to assign to several variables simultaneously. Given only one symbol and form, it has the same effect as `setq`. Given several symbol and form pairs, it evaluates all the forms in advance and then stores the corresponding variables afterwards.

```
(setq x 2 y 3)
(setq x (+ x y)  y (* x y))
x
     => 5
y                       ; y was computed after x was set.
     => 15
(setq x 2 y 3)
(psetq x (+ x y)  y (* x y))
x
     => 5
y                       ; y was computed before x was set.
     => 6
```

The simplest use of `psetq` is `(psetq x y y x)`, which exchanges the values of two variables. (The `rotatef` form provides an even more convenient way to swap two variables; see section Modify Macros.)

`psetq` always returns `nil`.

Generalized Variables

A "generalized variable" or "place form" is one of the many places in Lisp memory where values can be stored. The simplest place form is a regular Lisp variable. But the cars and cdrs of lists, elements of arrays, properties of symbols, and many other locations are also places where Lisp values are stored.

The `setf` form is like `setq`, except that it accepts arbitrary place forms on the left side rather than just symbols. For example, `(setf (car a) b)` sets the car of `a` to `b`, doing the same operation as `(setcar a b)` but without having to remember two separate functions for setting and accessing every type of place.

Generalized variables are analogous to "lvalues" in the C language, where `x = a[i]' gets an element from an array and `a[i] = x' stores an element using the same notation. Just as certain forms like `a[i]` can be lvalues in C, there is a set of forms that can be generalized variables in Lisp.

Basic Setf

The `setf` macro is the most basic way to operate on generalized variables.

Special Form: **setf** *[place form]...*

This macro evaluates form and stores it in place, which must be a valid generalized variable form. If there are several place and form pairs, the assignments are done sequentially just as with `setq`. `setf` returns the value of the last form.

The following Lisp forms will work as generalized variables, and so may legally appear in the place argument of `setf`:

- A symbol naming a variable. In other words, `(setf x y)` is exactly equivalent to `(setq x y)`, and `setq` itself is strictly speaking redundant now that `setf` exists. Many programmers continue to prefer `setq` for setting simple variables, though, purely for stylistic or historical reasons. The macro `(setf x y)` actually expands to `(setq x y)`, so there is no performance penalty for using it in compiled code.
- A call to any of the following Lisp functions:

```
car                 cdr                 caar .. cddddr
nth                 rest                first .. tenth
aref                elt                 nthcdr
symbol-function     symbol-value        symbol-plist
get                 get*                getf
gethash             subseq
```

  Note that for `nthcdr` and `getf`, the list argument of the function must itself be a valid place form. For example, `(setf (nthcdr 0 foo) 7)` will set `foo` itself to 7. Note that `push` and `pop` on an `nthcdr` place can be used to insert or delete at any position in a list. The use of `nthcdr` as a place form is an extension to standard Common Lisp.
- The following Emacs-specific functions are also `setf`-able. (Some of these are defined only in Emacs 19 or only in Lucid Emacs.)


```
buffer-file-name                marker-position
buffer-modified-p               match-data
buffer-name                     mouse-position
buffer-string                   overlay-end
buffer-substring                overlay-get
current-buffer                  overlay-start
current-case-table              point
current-column                  point-marker
current-global-map              point-max
current-input-mode              point-min
current-local-map               process-buffer
current-window-configuration    process-filter
default-file-modes              process-sentinel
default-value                   read-mouse-position
documentation-property          screen-height
extent-data                     screen-menubar
extent-end-position             screen-width
extent-start-position           selected-window
face-background                 selected-screen
face-background-pixmap          selected-frame
face-font                       standard-case-table
face-foreground                 syntax-table
face-underline-p                window-buffer
file-modes                      window-dedicated-p
frame-height                    window-display-table
frame-parameters                window-height
frame-visible-p                 window-hscroll
frame-width                     window-point
get-register                    window-start
getenv                          window-width
global-key-binding              x-get-cut-buffer
keymap-parent                   x-get-cutbuffer
local-key-binding               x-get-secondary-selection
mark                            x-get-selection
mark-marker
```

Most of these have directly corresponding "set" functions, like `use-local-map` for `current-local-map`, or `goto-char` for `point`. A few, like `point-min`, expand to longer sequences of code when they are `setf`'d (`(narrow-to-region x (point-max))` in this case).

- A call of the form `(substring subplace n [m])`, where subplace is itself a legal generalized variable whose current value is a string, and where the value stored is also a string. The new string is spliced into the specified part of the destination string. For example:

```
(setq a (list "hello" "world"))
     => ("hello" "world")
(cadr a)
```


```
     => "world"
(substring (cadr a) 2 4)
     => "rl"
(setf (substring (cadr a) 2 4) "o")
     => "o"
(cadr a)
     => "wood"
a
     => ("hello" "wood")
```

The generalized variable `buffer-substring`, listed above, also works in this way by replacing a portion of the current buffer.

- A call of the form `(apply 'func ...)` or `(apply (function func) ...)`, where func is a `setf`-able function whose store function is "suitable" in the sense described in Steele's book; since none of the standard Emacs place functions are suitable in this sense, this feature is only interesting when used with places you define yourself with `define-setf-method` or the long form of `defsetf`.
- A macro call, in which case the macro is expanded and `setf` is applied to the resulting form.
- Any form for which a `defsetf` or `define-setf-method` has been made.

Using any forms other than these in the place argument to `setf` will signal an error.

The `setf` macro takes care to evaluate all subforms in the proper left-to-right order; for example,

```
(setf (aref vec (incf i)) i)
```

looks like it will evaluate `(incf i)` exactly once, before the following access to `i`; the `setf` expander will insert temporary variables as necessary to ensure that it does in fact work this way no matter what setf-method is defined for `aref`. (In this case, `aset` would be used and no such steps would be necessary since `aset` takes its arguments in a convenient order.)

However, if the place form is a macro which explicitly evaluates its arguments in an unusual order, this unusual order will be preserved. Adapting an example from Steele, given

```
(defmacro wrong-order (x y) (list 'aref y x))
```

the form `(setf (wrong-order a b) 17)` will evaluate b first, then a, just as in an actual call to `wrong-order`.

Modify Macros

This package defines a number of other macros besides `setf` that operate on generalized variables. Many are interesting and useful even when the place is just a variable name.

Special Form: **psetf** *[place form]...*

This macro is to `setf` what `psetq` is to `setq`: When several places and forms are involved, the assignments take place in parallel rather than sequentially. Specifically, all subforms are evaluated from left to right, then all the assignments are done (in an undefined order).

Special Form: **incf** *place &optional x*

This macro increments the number stored in place by one, or by x if specified. The incremented value is returned. For example, `(incf i)` is equivalent to `(setq i (1+ i))`, and `(incf (car x) 2)` is equivalent to `(setcar x (+ (car x) 2))`.

Once again, care is taken to preserve the "apparent" order of evaluation. For example,

```
(incf (aref vec (incf i)))
```

appears to increment `i` once, then increment the element of `vec` addressed by `i`; this is indeed exactly what it does, which means the above form is *not* equivalent to the "obvious" expansion,

```
(setf (aref vec (incf i)) (1+ (aref vec (incf i))))   ; Wrong!
```

but rather to something more like

```
(let ((temp (incf i)))
  (setf (aref vec temp) (1+ (aref vec temp))))
```

Again, all of this is taken care of automatically by `incf` and the other generalized-variable macros.

As a more Emacs-specific example of `incf`, the expression `(incf (point) n)` is essentially equivalent to `(forward-char n)`.

Special Form: **decf** *place &optional x*

This macro decrements the number stored in place by one, or by x if specified.

Special Form: **pop** *place*

This macro removes and returns the first element of the list stored in place. It is analogous to `(prog1 (car place) (setf place (cdr place)))`, except that it takes care to evaluate all subforms only once.

Special Form: **push** *x place*

This macro inserts x at the front of the list stored in place. It is analogous to `(setf place (cons x place))`, except for evaluation of the subforms.

Special Form: **pushnew** *x place* `&key :test :test-not :key`

This macro inserts x at the front of the list stored in place, but only if x was not `eql` to any existing element of the list. The optional keyword arguments are interpreted in the same way as for `adjoin`. See section Lists as Sets.

Special Form: **shiftf** *place... newvalue*

This macro shifts the places left by one, shifting in the value of newvalue (which may be any Lisp expression, not just a generalized variable), and returning the value shifted out of the first place. Thus, `(shiftf a b c d)` is equivalent to

```
(prog1
```


```
   a
 (psetf a b
        b c
        c d))
```

except that the subforms of a, b, and c are actually evaluated only once each and in the apparent order.

Special Form: **rotatef** *place...*

This macro rotates the places left by one in circular fashion. Thus, `(rotatef a b c d)` is equivalent to

```
(psetf a b
       b c
       c d
       d a)
```

except for the evaluation of subforms. `rotatef` always returns `nil`. Note that `(rotatef a b)` conveniently exchanges a and b.

The following macros were invented for this package; they have no analogues in Common Lisp.

Special Form: **letf** *(bindings...) forms...*

This macro is analogous to `let`, but for generalized variables rather than just symbols. Each binding should be of the form `(place value)`; the original contents of the places are saved, the values are stored in them, and then the body forms are executed. Afterwards, the places are set back to their original saved contents. This cleanup happens even if the forms exit irregularly due to a `throw` or an error.

For example,

```
(letf (((point) (point-min))
       (a 17))
  ...)
```

moves "point" in the current buffer to the beginning of the buffer, and also binds `a` to 17 (as if by a normal `let`, since `a` is just a regular variable). After the body exits, `a` is set back to its original value and point is moved back to its original position.

Note that `letf` on `(point)` is not quite like a `save-excursion`, as the latter effectively saves a marker which tracks insertions and deletions in the buffer. Actually, a `letf` of `(point-marker)` is much closer to this behavior. (`point` and `point-marker` are equivalent as `setf` places; each will accept either an integer or a marker as the stored value.)

Since generalized variables look like lists, `let`'s shorthand of using `foo' for `(foo nil)' as a binding would be ambiguous in `letf` and is not allowed.

However, a binding specifier may be a one-element list `(place)', which is similar to `(place place)'. In other words, the place is not disturbed on entry to the body, and the only effect of the `letf` is to restore the original value of place afterwards. (The redundant access-and-store suggested by the `(place place)` example does not actually occur.)

In most cases, the place must have a well-defined value on entry to the `letf` form. The only exceptions are

plain variables and calls to `symbol-value` and `symbol-function`. If the symbol is not bound on entry it is simply made unbound by `makunbound` or `fmakunbound` on exit.

Special Form: **letf*** *(bindings...) forms...*

This macro is to `letf` what `let*` is to `let`: It does the bindings in sequential rather than parallel order.

Special Form: **callf** *function place args...*

This is the "generic" modify macro. It calls function, which should be an unquoted function name, macro name, or lambda. It passes place and args as arguments, and assigns the result back to place. For example, `(incf place n)` is the same as `(callf + place n)`. Some more examples:

```
(callf abs my-number)
(callf concat (buffer-name) "<" (int-to-string n) ">")
(callf union happy-people (list joe bob) :test 'same-person)
```

See section Customizing Setf, for `define-modify-macro`, a way to create even more concise notations for modify macros. Note again that `callf` is an extension to standard Common Lisp.

Special Form: **callf2** *function arg1 place args...*

This macro is like `callf`, except that place is the *second* argument of function rather than the first. For example, `(push x place)` is equivalent to `(callf2 cons x place)`.

The `callf` and `callf2` macros serve as building blocks for other macros like `incf`, `pushnew`, and `define-modify-macro`. The `letf` and `letf*` macros are used in the processing of symbol macros; see section Macro Bindings.

Customizing Setf

Common Lisp defines three macros, `define-modify-macro`, `defsetf`, and `define-setf-method`, that allow the user to extend generalized variables in various ways.

Special Form: **define-modify-macro** *name arglist function [doc-string]*

This macro defines a "read-modify-write" macro similar to `incf` and `decf`. The macro name is defined to take a place argument followed by additional arguments described by arglist. The call

```
(name place args...)
```

will be expanded to

```
(callf func place args...)
```

which in turn is roughly equivalent to

```
(setf place (func place args...))
```

For example:


```
(define-modify-macro incf (&optional (n 1)) +)
(define-modify-macro concatf (&rest args) concat)
```

Note that `&key` is not allowed in arglist, but `&rest` is sufficient to pass keywords on to the function.

Most of the modify macros defined by Common Lisp do not exactly follow the pattern of `define-modify-macro`. For example, `push` takes its arguments in the wrong order, and `pop` is completely irregular. You can define these macros "by hand" using `get-setf-method`, or consult the source file `` `cl-macs.el' `` to see how to use the internal `setf` building blocks.

Special Form: **defsetf** *access-fn update-fn*

This is the simpler of two `defsetf` forms. Where access-fn is the name of a function which accesses a place, this declares update-fn to be the corresponding store function. From now on,

```
(setf (access-fn arg1 arg2 arg3) value)
```

will be expanded to

```
(update-fn arg1 arg2 arg3 value)
```

The update-fn is required to be either a true function, or a macro which evaluates its arguments in a function-like way. Also, the update-fn is expected to return value as its result. Otherwise, the above expansion would not obey the rules for the way `setf` is supposed to behave.

As a special (non-Common-Lisp) extension, a third argument of `t` to `defsetf` says that the `update-fn`'s return value is not suitable, so that the above `setf` should be expanded to something more like

```
(let ((temp value))
  (update-fn arg1 arg2 arg3 temp)
  temp)
```

Some examples of the use of `defsetf`, drawn from the standard suite of setf methods, are:

```
(defsetf car setcar)
(defsetf symbol-value set)
(defsetf buffer-name rename-buffer t)
```

Special Form: **defsetf** *access-fn arglist (store-var) forms...*

This is the second, more complex, form of `defsetf`. It is rather like `defmacro` except for the additional store-var argument. The forms should return a Lisp form which stores the value of store-var into the generalized variable formed by a call to access-fn with arguments described by arglist. The forms may begin with a string which documents the `setf` method (analogous to the doc string that appears at the front of a function).

For example, the simple form of `defsetf` is shorthand for

```
(defsetf access-fn (&rest args) (store)
  (append '(update-fn) args (list store)))
```

The Lisp form that is returned can access the arguments from arglist and store-var in an unrestricted fash macros like `setf` and `incf` which invoke this setf-method will insert temporary variables as needed to make sure the apparent order of evaluation is preserved.

Another example drawn from the standard package:

```
(defsetf nth (n x) (store)
  (list 'setcar (list 'nthcdr n x) store))
```

Special Form: **define-setf-method** *access-fn arglist forms...*

This is the most general way to create new place forms. When a `setf` to access-fn with arguments described by arglist is expanded, the forms are evaluated and must return a list of five items:

1. A list of temporary variables.
2. A list of value forms corresponding to the temporary variables above. The temporary variables will be bound to these value forms as the first step of any operation on the generalized variable.
3. A list of exactly one store variable (generally obtained from a call to `gensym`).
4. A Lisp form which stores the contents of the store variable into the generalized variable, assuming the temporaries have been bound as described above.
5. A Lisp form which accesses the contents of the generalized variable, assuming the temporaries have been bound.

This is exactly like the Common Lisp macro of the same name, except that the method returns a list of five values rather than the five values themselves, since Emacs Lisp does not support Common Lisp's notion of multiple return values.

Once again, the forms may begin with a documentation string.

A setf-method should be maximally conservative with regard to temporary variables. In the setf-methods generated by `defsetf`, the second return value is simply the list of arguments in the place form, and the first return value is a list of a corresponding number of temporary variables generated by `gensym`. Macros like `setf` and `incf` which use this setf-method will optimize away most temporaries that turn out to be unnecessary, so there is little reason for the setf-method itself to optimize.

Function: **get-setf-method** *place &optional env*

This function returns the setf-method for place, by invoking the definition previously recorded by `defsetf` or `define-setf-method`. The result is a list of five values as described above. You can use this function to build your own `incf`-like modify macros. (Actually, it is better to use the internal functions `cl-setf-do-modify` and `cl-setf-do-store`, which are a bit easier to use and which also do a number of optimizations; consult the source code for the `incf` function for a simple example.)

The argument env specifies the "environment" to be passed on to `macroexpand` if `get-setf-method` should need to expand a macro in place. It should come from an `&environment` argument to the macro or setf-method that called `get-setf-method`.

See also the source code for the setf-methods for `apply` and `substring`, each of which works by calling `get-setf-method` on a simpler case, then massaging the result in various ways.

Modern Common Lisp defines a second, independent way to specify the `setf` behavior of a function,

namely "`setf` functions" whose names are lists `(setf name)` rather than symbols. For example, `(defsetf foo ...)` `(setf foo) ...)` defines the function that is used when `setf` is applied to `foo`. This package does currently support `setf` functions. In particular, it is a compile-time error to use `setf` on a form which has not already been `defsetf`'d or otherwise declared; in newer Common Lisps, this would not be an error since the function `(setf func)` might be defined later.

@secno=4

Variable Bindings

These Lisp forms make bindings to variables and function names, analogous to Lisp's built-in `let` form.

See section Modify Macros, for the `letf` and `letf*` forms which are also related to variable bindings.

Dynamic Bindings

The standard `let` form binds variables whose names are known at compile-time. The `progv` form provides an easy way to bind variables whose names are computed at run-time.

Special Form: **progv** *symbols values forms...*

This form establishes `let`-style variable bindings on a set of variables computed at run-time. The expressions symbols and values are evaluated, and must return lists of symbols and values, respectively. The symbols are bound to the corresponding values for the duration of the body forms. If values is shorter than symbols, the last few symbols are made unbound (as if by `makunbound`) inside the body. If symbols is shorter than values, the excess values are ignored.

Lexical Bindings

The CL package defines the following macro which more closely follows the Common Lisp `let` form:

Special Form: **lexical-let** *(bindings...) forms...*

This form is exactly like `let` except that the bindings it establishes are purely lexical. Lexical bindings are similar to local variables in a language like C: Only the code physically within the body of the `lexical-let` (after macro expansion) may refer to the bound variables.

```
(setq a 5)
(defun foo (b) (+ a b))
(let ((a 2)) (foo a))
     => 4
(lexical-let ((a 2)) (foo a))
     => 7
```

In this example, a regular `let` binding of `a` actually makes a temporary change to the global variable `a`, so `foo` is able to see the binding of `a` to 2. But `lexical-let` actually creates a distinct local variable `a` for use within its body, without any effect on the global variable of the same name.

The most important use of lexical bindings is to create closures. A closure is a function object that refers to

an outside lexical variable. For example:

```
(defun make-adder (n)
  (lexical-let ((n n))
    (function (lambda (m) (+ n m)))))
(setq add17 (make-adder 17))
(funcall add17 4)
     => 21
```

The call `(make-adder 17)` returns a function object which adds 17 to its argument. If `let` had been used instead of `lexical-let`, the function object would have referred to the global `n`, which would have been bound to 17 only during the call to `make-adder` itself.

```
(defun make-counter ()
  (lexical-let ((n 0))
    (function* (lambda (&optional (m 1)) (incf n m)))))
(setq count-1 (make-counter))
(funcall count-1 3)
     => 3
(funcall count-1 14)
     => 17
(setq count-2 (make-counter))
(funcall count-2 5)
     => 5
(funcall count-1 2)
     => 19
(funcall count-2)
     => 6
```

Here we see that each call to `make-counter` creates a distinct local variable `n`, which serves as a private counter for the function object that is returned.

Closed-over lexical variables persist until the last reference to them goes away, just like all other Lisp objects. For example, `count-2` refers to a function object which refers to an instance of the variable `n`; this is the only reference to that variable, so after `(setq count-2 nil)` the garbage collector would be able to delete this instance of `n`. Of course, if a `lexical-let` does not actually create any closures, then the lexical variables are free as soon as the `lexical-let` returns.

Many closures are used only during the extent of the bindings they refer to; these are known as "downward funargs" in Lisp parlance. When a closure is used in this way, regular Emacs Lisp dynamic bindings suffice and will be more efficient than `lexical-let` closures:

```
(defun add-to-list (x list)
  (mapcar (function (lambda (y) (+ x y))) list))
(add-to-list 7 '(1 2 5))
     => (8 9 12)
```

Since this lambda is only used while `x` is still bound, it is not necessary to make a true closure out of it.

You can use `defun` or `flet` inside a `lexical-let` to create a named closure. If several closures are created in the body of a single `lexical-let`, they all close over the same instance of the lexical variable.

The `lexical-let` form is an extension to Common Lisp. In true Common Lisp, all bindings are lexical unless declared otherwise.

Special Form: **lexical-let*** *(bindings...) forms...*

This form is just like `lexical-let`, except that the bindings are made sequentially in the manner of `let*`.

Function Bindings

These forms make `let`-like bindings to functions instead of variables.

Special Form: **flet** *(bindings...) forms...*

This form establishes `let`-style bindings on the function cells of symbols rather than on the value cells. Each binding must be a list of the form `(name arglist forms...)', which defines a function exactly as if it were a `defun*` form. The function name is defined accordingly for the duration of the body of the `flet`; then the old function definition, or lack thereof, is restored.

While `flet` in Common Lisp establishes a lexical binding of name, Emacs Lisp `flet` makes a dynamic binding. The result is that `flet` affects indirect calls to a function as well as calls directly inside the `flet` form itself.

You can use `flet` to disable or modify the behavior of a function in a temporary fashion. This will even work on Emacs primitives, although note that some calls to primitive functions internal to Emacs are made without going through the symbol's function cell, and so will not be affected by `flet`. For example,

```
(flet ((message (&rest args) (push args saved-msgs)))
  (do-something))
```

This code attempts to replace the built-in function `message` with a function that simply saves the messages in a list rather than displaying them. The original definition of `message` will be restored after `do-something` exits. This code will work fine on messages generated by other Lisp code, but messages generated directly inside Emacs will not be caught since they make direct C-language calls to the message routines rather than going through the Lisp `message` function.

Functions defined by `flet` may use the full Common Lisp argument notation supported by `defun*`; also, the function body is enclosed in an implicit block as if by `defun*`. See section Program Structure.

Special Form: **labels** *(bindings...) forms...*

The `labels` form is a synonym for `flet`. (In Common Lisp, `labels` and `flet` differ in ways that depend on their lexical scoping; these distinctions vanish in dynamically scoped Emacs Lisp.)

Macro Bindings

These forms create local macros and "symbol macros."

Special Form: **macrolet** *(bindings...) forms...*

This form is analogous to `flet`, but for macros instead of functions. Each binding is a list of the same form as the arguments to `defmacro*` (i.e., a macro name, argument list, and macro-expander forms). The macro is defined accordingly for use within the body of the `macrolet`.

Because of the nature of macros, `macrolet` is lexically scoped even in Emacs Lisp: The `macrolet` binding will affect only calls that appear physically within the body forms, possibly after expansion of other macros in the body.

Special Form: **symbol-macrolet** *(bindings...) forms...*

This form creates symbol macros, which are macros that look like variable references rather than function calls. Each binding is a list `(var expansion)'; any reference to var within the body forms is replaced by expansion.

```
(setq bar '(5 . 9))
(symbol-macrolet ((foo (car bar)))
  (incf foo))
bar
     => (6 . 9)
```

A `setq` of a symbol macro is treated the same as a `setf`. I.e., `(setq foo 4)` in the above would be equivalent to `(setf foo 4)`, which in turn expands to `(setf (car bar) 4)`.

Likewise, a `let` or `let*` binding a symbol macro is treated like a `letf` or `letf*`. This differs from true Common Lisp, where the rules of lexical scoping cause a `let` binding to shadow a `symbol-macrolet` binding. In this package, only `lexical-let` and `lexical-let*` will shadow a symbol macro.

There is no analogue of `defmacro` for symbol macros; all symbol macros are local. A typical use of `symbol-macrolet` is in the expansion of another macro:

```
(defmacro* my-dolist ((x list) &rest body)
  (let ((var (gensym)))
    (list 'loop 'for var 'on list 'do
          (list* 'symbol-macrolet (list (list x (list 'car var)))
                 body))))

(setq mylist '(1 2 3 4))
(my-dolist (x mylist) (incf x))
mylist
     => (2 3 4 5)
```

In this example, the `my-dolist` macro is similar to `dolist` (see section Iteration) except that the variable `x` becomes a true reference onto the elements of the list. The `my-dolist` call shown here expands to

```
(loop for G1234 on mylist do
      (symbol-macrolet ((x (car G1234)))
        (incf x)))
```

which in turn expands to


```
(loop for G1234 on mylist do (incf (car G1234)))
```

See section Loop Facility, for a description of the `loop` macro. This package defines a nonstandard `in-ref` loop clause that works much like `my-dolist`.

Conditionals

These conditional forms augment Emacs Lisp's simple `if`, `and`, `or`, and `cond` forms.

Special Form: **when** *test forms...*

This is a variant of `if` where there are no "else" forms, and possibly several "then" forms. In particular,

```
(when test a b c)
```

is entirely equivalent to

```
(if test (progn a b c) nil)
```

Special Form: **unless** *test forms...*

This is a variant of `if` where there are no "then" forms, and possibly several "else" forms:

```
(unless test a b c)
```

is entirely equivalent to

```
(when (not test) a b c)
```

Special Form: **case** *keyform clause...*

This macro evaluates keyform, then compares it with the key values listed in the various clauses. Whichever clause matches the key is executed; comparison is done by `eql`. If no clause matches, the `case` form returns `nil`. The clauses are of the form

```
(keylist body-forms...)
```

where keylist is a list of key values. If there is exactly one value, and it is not a cons cell or the symbol `nil` or `t`, then it can be used by itself as a keylist without being enclosed in a list. All key values in the `case` form must be distinct. The final clauses may use `t` in place of a keylist to indicate a default clause that should be taken if none of the other clauses match. (The symbol `otherwise` is also recognized in place of `t`. To make a clause that matches the actual symbol `t`, `nil`, or `otherwise`, enclose the symbol in a list.)

For example, this expression reads a keystroke, then does one of four things depending on whether it is an `a', a `b', a RET or LFD, or anything else.

```
(case (read-char)
  (?a (do-a-thing))
```


```
(?b (do-b-thing))
((?\r ?\n) (do-ret-thing))
(t (do-other-thing)))
```

Special Form: **ecase** *keyform clause...*

This macro is just like `case`, except that if the key does not match any of the clauses, an error is signalled rather than simply returning `nil`.

Special Form: **typecase** *keyform clause...*

This macro is a version of `case` that checks for types rather than values. Each clause is of the form `(type body...)'. See section Type Predicates, for a description of type specifiers. For example,

```
(typecase x
  (integer (munch-integer x))
  (float (munch-float x))
  (string (munch-integer (string-to-int x)))
  (t (munch-anything x)))
```

The type specifier `t` matches any type of object; the word `otherwise` is also allowed. To make one clause match any of several types, use an `(or ...)` type specifier.

Special Form: **etypecase** *keyform clause...*

This macro is just like `typecase`, except that if the key does not match any of the clauses, an error is signalled rather than simply returning `nil`.

Blocks and Exits

Common Lisp blocks provide a non-local exit mechanism very similar to `catch` and `throw`, but lexically rather than dynamically scoped. This package actually implements `block` in terms of `catch`; however, the lexical scoping allows the optimizing byte-compiler to omit the costly `catch` step if the body of the block does not actually `return-from` the block.

Special Form: **block** *name forms...*

The forms are evaluated as if by a `progn`. However, if any of the forms execute `(return-from name)`, they will jump out and return directly from the `block` form. The `block` returns the result of the last form unless a `return-from` occurs.

The `block`/`return-from` mechanism is quite similar to the `catch`/`throw` mechanism. The main differences are that block names are unevaluated symbols, rather than forms (such as quoted symbols) which evaluate to a tag at run-time; and also that blocks are lexically scoped whereas `catch`/`throw` are dynamically scoped. This means that functions called from the body of a `catch` can also `throw` to the `catch`, but the `return-from` referring to a block name must appear physically within the forms that make up the body of the block. They may not appear within other called functions, although they may appear within macro expansions or `lambda`s in the body. Block names and `catch` names form independent name-spaces.

In true Common Lisp, `defun` and `defmacro` surround the function or expander bodies with implicit bl with the same name as the function or macro. This does not occur in Emacs Lisp, but this package provi `defun*` and `defmacro*` forms which do create the implicit block.

The Common Lisp looping constructs defined by this package, such as `loop` and `dolist`, also create implicit blocks just as in Common Lisp.

Because they are implemented in terms of Emacs Lisp `catch` and `throw`, blocks have the same overhead as actual `catch` constructs (roughly two function calls). However, Zawinski and Furuseth's optimizing byte compiler (standard in Emacs 19) will optimize away the `catch` if the block does not in fact contain any `return` or `return-from` calls that jump to it. This means that `do` loops and `defun*` functions which don't use `return` don't pay the overhead to support it.

Special Form: **return-from** *name [result]*

This macro returns from the block named name, which must be an (unevaluated) symbol. If a result form is specified, it is evaluated to produce the result returned from the `block`. Otherwise, `nil` is returned.

Special Form: **return** *[result]*

This macro is exactly like `(return-from nil result)`. Common Lisp loops like `do` and `dolist` implicitly enclose themselves in `nil` blocks.

Iteration

The macros described here provide more sophisticated, high-level looping constructs to complement Emacs Lisp's basic `while` loop.

Special Form: **loop** *forms...*

The CL package supports both the simple, old-style meaning of `loop` and the extremely powerful and flexible feature known as the Loop Facility or Loop Macro. This more advanced facility is discussed in the following section; see section Loop Facility. The simple form of `loop` is described here.

If `loop` is followed by zero or more Lisp expressions, then `(loop exprs...)` simply creates an infinite loop executing the expressions over and over. The loop is enclosed in an implicit `nil` block. Thus,

```
(loop (foo)  (if (no-more) (return 72))  (bar))
```

is exactly equivalent to

```
(block nil (while t (foo)  (if (no-more) (return 72))  (bar)))
```

If any of the expressions are plain symbols, the loop is instead interpreted as a Loop Macro specification as described later. (This is not a restriction in practice, since a plain symbol in the above notation would simply access and throw away the value of a variable.)

Special Form: **do** *(spec...) (end-test [result...]) forms...*

This macro creates a general iterative loop. Each spec is of the form


```
(var [init [step]])
```

The loop works as follows: First, each var is bound to the associated init value as if by a `let` form. Then, in each iteration of the loop, the end-test is evaluated; if true, the loop is finished. Otherwise, the body forms are evaluated, then each var is set to the associated step expression (as if by a `psetq` form) and the next iteration begins. Once the end-test becomes true, the result forms are evaluated (with the vars still bound to their values) to produce the result returned by `do`.

The entire `do` loop is enclosed in an implicit `nil` block, so that you can use `(return)` to break out of the loop at any time.

If there are no result forms, the loop returns `nil`. If a given var has no step form, it is bound to its init value but not otherwise modified during the `do` loop (unless the code explicitly modifies it); this case is just a shorthand for putting a `(let ((var init)) ...)` around the loop. If init is also omitted it defaults to `nil`, and in this case a plain `var' can be used in place of `(var)', again following the analogy with `let`.

This example (from Steele) illustrates a loop which applies the function `f` to successive pairs of values from the lists `foo` and `bar`; it is equivalent to the call `(mapcar* 'f foo bar)`. Note that this loop has no body forms at all, performing all its work as side effects of the rest of the loop.

```
(do ((x foo (cdr x))
     (y bar (cdr y))
     (z nil (cons (f (car x) (car y)) z)))
  ((or (null x) (null y))
   (nreverse z)))
```

Special Form: **do*** *(spec...) (end-test [result...]) forms...*

This is to `do` what `let*` is to `let`. In particular, the initial values are bound as if by `let*` rather than `let`, and the steps are assigned as if by `setq` rather than `psetq`.

Here is another way to write the above loop:

```
(do* ((xp foo (cdr xp))
      (yp bar (cdr yp))
      (x (car xp) (car xp))
      (y (car yp) (car yp))
      z)
  ((or (null xp) (null yp))
   (nreverse z))
  (push (f x y) z))
```

Special Form: **dolist** *(var list [result]) forms...*

This is a more specialized loop which iterates across the elements of a list. list should evaluate to a list; the body forms are executed with var bound to each element of the list in turn. Finally, the result form (or `nil`) is evaluated with var bound to `nil` to produce the result returned by the loop. The loop is surrounded by an implicit `nil` block.

Special Form: **dotimes** *(var count [result]) forms...*

This is a more specialized loop which iterates a specified number of times. The body is executed with va bound to the integers from zero (inclusive) to count (exclusive), in turn. Then the `result` form is evalu with var bound to the total number of iterations that were done (i.e., `(max 0 count)`) to get the return value for the loop form. The loop is surrounded by an implicit `nil` block.

Special Form: **do-symbols** *(var [obarray [result]]) forms...*

This loop iterates over all interned symbols. If obarray is specified and is not `nil`, it loops over all symbols in that obarray. For each symbol, the body forms are evaluated with var bound to that symbol. The symbols are visited in an unspecified order. Afterward the result form, if any, is evaluated (with var bound to `nil`) to get the return value. The loop is surrounded by an implicit `nil` block.

Special Form: **do-all-symbols** *(var [result]) forms...*

This is identical to `do-symbols` except that the obarray argument is omitted; it always iterates over the default obarray.

See section Mapping over Sequences, for some more functions for iterating over vectors or lists.

Loop Facility

A common complaint with Lisp's traditional looping constructs is that they are either too simple and limited, such as Common Lisp's `dotimes` or Emacs Lisp's `while`, or too unreadable and obscure, like Common Lisp's `do` loop.

To remedy this, recent versions of Common Lisp have added a new construct called the "Loop Facility" or "`loop` macro," with an easy-to-use but very powerful and expressive syntax.

Loop Basics

The `loop` macro essentially creates a mini-language within Lisp that is specially tailored for describing loops. While this language is a little strange-looking by the standards of regular Lisp, it turns out to be very easy to learn and well-suited to its purpose.

Since `loop` is a macro, all parsing of the loop language takes place at byte-compile time; compiled `loop`s are just as efficient as the equivalent `while` loops written longhand.

Special Form: **loop** *clauses...*

A loop construct consists of a series of clauses, each introduced by a symbol like `for` or `do`. Clauses are simply strung together in the argument list of `loop`, with minimal extra parentheses. The various types of clauses specify initializations, such as the binding of temporary variables, actions to be taken in the loop, stepping actions, and final cleanup.

Common Lisp specifies a certain general order of clauses in a loop:

```
(loop name-clause
      var-clauses...
      action-clauses...)
```

The name-clause optionally gives a name to the implicit block that surrounds the loop. By default, the implicit block is named `nil`. The var-clauses specify what variables should be bound during the loop, and how they should be modified or iterated throughout the course of the loop. The action-clauses are things to be done during the loop, such as computing, collecting, and returning values.

The Emacs version of the `loop` macro is less restrictive about the order of clauses, but things will behave most predictably if you put the variable-binding clauses `with`, `for`, and `repeat` before the action clauses. As in Common Lisp, `initially` and `finally` clauses can go anywhere.

Loops generally return `nil` by default, but you can cause them to return a value by using an accumulation clause like `collect`, an end-test clause like `always`, or an explicit `return` clause to jump out of the implicit block. (Because the loop body is enclosed in an implicit block, you can also use regular Lisp `return` or `return-from` to break out of the loop.)

The following sections give some examples of the Loop Macro in action, and describe the particular loop clauses in great detail. Consult the second edition of Steele's Common Lisp, the Language, for additional discussion and examples of the `loop` macro.

Loop Examples

Before listing the full set of clauses that are allowed, let's look at a few example loops just to get a feel for the `loop` language.

```
(loop for buf in (buffer-list)
      collect (buffer-file-name buf))
```

This loop iterates over all Emacs buffers, using the list returned by `buffer-list`. For each buffer `buf`, it calls `buffer-file-name` and collects the results into a list, which is then returned from the `loop` construct. The result is a list of the file names of all the buffers in Emacs' memory. The words `for`, `in`, and `collect` are reserved words in the `loop` language.

```
(loop repeat 20 do (insert "Yowsa\n"))
```

This loop inserts the phrase "Yowsa" twenty times in the current buffer.

```
(loop until (eobp) do (munch-line) (forward-line 1))
```

This loop calls `munch-line` on every line until the end of the buffer. If point is already at the end of the buffer, the loop exits immediately.

```
(loop do (munch-line) until (eobp) do (forward-line 1))
```

This loop is similar to the above one, except that `munch-line` is always called at least once.

```
(loop for x from 1 to 100
      for y = (* x x)
      until (>= y 729)
      finally return (list x (= y 729)))
```

This more complicated loop searches for a number `x` whose square is 729. For safety's sake it only exam `x` values up to 100; dropping the phrase `to 100' would cause the loop to count upwards with no limit. Th second `for` clause defines `y` to be the square of `x` within the loop; the expression after the `=` sign is reevaluated each time through the loop. The `until` clause gives a condition for terminating the loop, and the `finally` clause says what to do when the loop finishes. (This particular example was written less concisely than it could have been, just for the sake of illustration.)

Note that even though this loop contains three clauses (two `for`s and an `until`) that would have been enough to define loops all by themselves, it still creates a single loop rather than some sort of triple-nested loop. You must explicitly nest your `loop` constructs if you want nested loops.

For Clauses

Most loops are governed by one or more `for` clauses. A `for` clause simultaneously describes variables to be bound, how those variables are to be stepped during the loop, and usually an end condition based on those variables.

The word `as` is a synonym for the word `for`. This word is followed by a variable name, then a word like `from` or `across` that describes the kind of iteration desired. In Common Lisp, the phrase `being the` sometimes precedes the type of iteration; in this package both `being` and `the` are optional. The word `each` is a synonym for `the`, and the word that follows it may be singular or plural: `for x being the elements of y' or `for x being each element of y'. Which form you use is purely a matter of style.

The variable is bound around the loop as if by `let`:

```
(setq i 'happy)
(loop for i from 1 to 10 do (do-something-with i))
i
     => happy
```

`for var from expr1 to expr2 by expr3`

This type of `for` clause creates a counting loop. Each of the three sub-terms is optional, though there must be at least one term so that the clause is marked as a counting clause.

The three expressions are the starting value, the ending value, and the step value, respectively, of the variable. The loop counts upwards by default (expr3 must be positive), from expr1 to expr2 inclusively. If you omit the `from` term, the loop counts from zero; if you omit the `to` term, the loop counts forever without stopping (unless stopped by some other loop clause, of course); if you omit the `by` term, the loop counts in steps of one.

You can replace the word `from` with `upfrom` or `downfrom` to indicate the direction of the loop. Likewise, you can replace `to` with `upto` or `downto`. For example, `for x from 5 downto 1' executes five times with `x` taking on the integers from 5 down to 1 in turn. Also, you can replace `to` with `below` or `above`, which are like `upto` and `downto` respectively except that they are exclusive rather than inclusive limits:

```
(loop for x to 10 collect x)
     => (0 1 2 3 4 5 6 7 8 9 10)
(loop for x below 10 collect x)
```


```
     => (0 1 2 3 4 5 6 7 8 9)
```

The `by` value is always positive, even for downward-counting loops. Some sort of `from` value is required for downward loops; `for x downto 5' is not a legal loop clause all by itself.

`for var in list by function`

This clause iterates var over all the elements of list, in turn. If you specify the `by` term, then function is used to traverse the list instead of `cdr`; it must be a function taking one argument. For example:

```
(loop for x in '(1 2 3 4 5 6) collect (* x x))
     => (1 4 9 16 25 36)
(loop for x in '(1 2 3 4 5 6) by 'cddr collect (* x x))
     => (1 9 25)
```

`for var on list by function`

This clause iterates var over all the cons cells of list.

```
(loop for x on '(1 2 3 4) collect x)
     => ((1 2 3 4) (2 3 4) (3 4) (4))
```

With `by`, there is no real reason that the `on` expression must be a list. For example:

```
(loop for x on first-animal by 'next-animal collect x)
```

where `(next-animal x)` takes an "animal" x and returns the next in the (assumed) sequence of animals, or `nil` if x was the last animal in the sequence.

`for var in-ref list by function`

This is like a regular `in` clause, but var becomes a `setf`-able "reference" onto the elements of the list rather than just a temporary variable. For example,

```
(loop for x in-ref my-list do (incf x))
```

increments every element of `my-list` in place. This clause is an extension to standard Common Lisp.

`for var across array`

This clause iterates var over all the elements of array, which may be a vector or a string.

```
(loop for x across "aeiou"
      do (use-vowel (char-to-string x)))
```

`for var across-ref array`

This clause iterates over an array, with var a `setf`-able reference onto the elements; see `in-ref` above.

`for var being the elements of sequence`

This clause iterates over the elements of sequence, which may be a list, vector, or string. Since the type must be determined at run-time, this is somewhat less efficient than `in` or `across`. The clause may be followed by the additional term `using (index var2)' to cause var2 to be bound to the successive indices (starting at 0) of the elements.

This clause type is taken from older versions of the `loop` macro, and is not present in modern Common Lisp. The `using (sequence ...)' term of the older macros is not supported.

`for var being the elements of-ref sequence`

This clause iterates over a sequence, with var a `setf`-able reference onto the elements; see `in-ref` above.

`for var being the symbols [of obarray]`

This clause iterates over symbols, either over all interned symbols or over all symbols in obarray. The loop is executed with var bound to each symbol in turn. The symbols are visited in an unspecified order.

As an example,

```
(loop for sym being the symbols
      when (fboundp sym)
      when (string-match "^map" (symbol-name sym))
      collect sym)
```

returns a list of all the functions whose names begin with `map'.

The Common Lisp words `external-symbols` and `present-symbols` are also recognized but are equivalent to `symbols` in Emacs Lisp.

Due to a minor implementation restriction, it will not work to have more than one `for` clause iterating over symbols, hash tables, keymaps, overlays, or intervals in a given `loop`. Fortunately, it would rarely if ever be useful to do so. It *is* legal to mix one of these types of clauses with other clauses like `for ... to` or `while`.

`for var being the hash-keys of hash-table`

This clause iterates over the entries in hash-table. For each hash table entry, var is bound to the entry's key. If you write `the hash-values' instead, var is bound to the values of the entries. The clause may be followed by the additional term `using (hash-values var2)' (where `hash-values` is the opposite word of the word following `the`) to cause var and var2 to be bound to the two parts of each hash table entry.

`for var being the key-codes of keymap`

This clause iterates over the entries in keymap. In GNU Emacs 18 and 19, keymaps are either alists or vectors, and key-codes are integers or symbols. In Lucid Emacs 19, keymaps are a special new data type, and key-codes are symbols or lists of symbols. The iteration does not enter nested keymaps or inherited (parent) keymaps. You can use `the key-bindings' to access the commands bound to the keys rather than the key codes, and you can add a `using` clause to access both the codes and the bindings together.

`for var being the key-seqs of keymap`

This clause iterates over all key sequences defined by keymap and its nested keymaps, where var takes on values which are strings in Emacs 18 or vectors in Emacs 19. The strings or vectors are reused for each iteration, so you must copy them if you wish to keep them permanently. You can add a `using (key-bindings ...)' clause to get the command bindings as well.

`for var being the overlays [of buffer] ...`

This clause iterates over the Emacs 19 "overlays" or Lucid Emacs "extents" of a buffer (the clause

`extents` is synonymous with `overlays`). Under Emacs 18, this clause iterates zero times. If th term is omitted, the current buffer is used. This clause also accepts optional `from pos' and `to pos' terms, limiting the clause to overlays which overlap the specified region.

`for var being the intervals [of buffer] ...`

This clause iterates over all intervals of a buffer with constant text properties. The variable var will be bound to conses of start and end positions, where one start position is always equal to the previous end position. The clause allows `of`, `from`, `to`, and `property` terms, where the latter term restricts the search to just the specified property. The `of` term may specify either a buffer or a string. This clause is useful only in GNU Emacs 19; in other versions, all buffers and strings consist of a single interval.

`for var being the frames`

This clause iterates over all frames, i.e., X window system windows open on Emacs files. This clause works only under Emacs 19. The clause `screens` is a synonym for `frames`. The frames are visited in `next-frame` order starting from `selected-frame`.

`for var being the windows [of frame]`

This clause iterates over the windows (in the Emacs sense) of the current frame, or of the specified frame. (In Emacs 18 there is only ever one frame, and the `of` term is not allowed there.)

`for var being the buffers`

This clause iterates over all buffers in Emacs. It is equivalent to `for var in (buffer-list)'.

`for var = expr1 then expr2`

This clause does a general iteration. The first time through the loop, var will be bound to expr1. On the second and successive iterations it will be set by evaluating expr2 (which may refer to the old value of var). For example, these two loops are effectively the same:

```
(loop for x on my-list by 'cddr do ...)
(loop for x = my-list then (cddr x) while x do ...)
```

Note that this type of `for` clause does not imply any sort of terminating condition; the above example combines it with a `while` clause to tell when to end the loop.

If you omit the `then` term, expr1 is used both for the initial setting and for successive settings:

```
(loop for x = (random) when (> x 0) return x)
```

This loop keeps taking random numbers from the `(random)` function until it gets a positive one, which it then returns.

If you include several `for` clauses in a row, they are treated sequentially (as if by `let*` and `setq`). You can instead use the word `and` to link the clauses, in which case they are processed in parallel (as if by `let` and `psetq`).

```
(loop for x below 5 for y = nil then x collect (list x y))
     => ((0 nil) (1 1) (2 2) (3 3) (4 4))
(loop for x below 5 and y = nil then x collect (list x y))
     => ((0 nil) (1 0) (2 1) (3 2) (4 3))
```

In the first loop, `y` is set based on the value of `x` that was just set by the previous clause; in the second loop, `x`

and `y` are set simultaneously so `y` is set based on the value of `x` left over from the previous time through the loop.

Another feature of the `loop` macro is destructuring, similar in concept to the destructuring provided by `defmacro`. The var part of any `for` clause can be given as a list of variables instead of a single variable. The values produced during loop execution must be lists; the values in the lists are stored in the corresponding variables.

```
(loop for (x y) in '((2 3) (4 5) (6 7)) collect (+ x y))
     => (5 9 13)
```

In loop destructuring, if there are more values than variables the trailing values are ignored, and if there are more variables than values the trailing variables get the value `nil`. If `nil` is used as a variable name, the corresponding values are ignored. Destructuring may be nested, and dotted lists of variables like `(x . y)` are allowed.

Iteration Clauses

Aside from `for` clauses, there are several other loop clauses that control the way the loop operates. They might be used by themselves, or in conjunction with one or more `for` clauses.

`repeat integer`

This clause simply counts up to the specified number using an internal temporary variable. The loops

```
(loop repeat n do ...)
(loop for temp to n do ...)
```

are identical except that the second one forces you to choose a name for a variable you aren't actually going to use.

`while condition`

This clause stops the loop when the specified condition (any Lisp expression) becomes `nil`. For example, the following two loops are equivalent, except for the implicit `nil` block that surrounds the second one:

```
(while cond forms...)
(loop while cond do forms...)
```

`until condition`

This clause stops the loop when the specified condition is true, i.e., non-`nil`.

`always condition`

This clause stops the loop when the specified condition is `nil`. Unlike `while`, it stops the loop using `return nil` so that the `finally` clauses are not executed. If all the conditions were non-`nil`, the loop returns `t`:

```
(if (loop for size in size-list always (> size 10))
    (some-big-sizes)
  (no-big-sizes))
```

`never condition`

This clause is like `always`, except that the loop returns `t` if any conditions were false, or `nil` otherwise.

`thereis condition`

This clause stops the loop when the specified form is non-`nil`; in this case, it returns that non-`nil` value. If all the values were `nil`, the loop returns `nil`.

Accumulation Clauses

These clauses cause the loop to accumulate information about the specified Lisp form. The accumulated result is returned from the loop unless overridden, say, by a `return` clause.

`collect form`

This clause collects the values of form into a list. Several examples of `collect` appear elsewhere in this manual.

The word `collecting` is a synonym for `collect`, and likewise for the other accumulation clauses.

`append form`

This clause collects lists of values into a result list using `append`.

`nconc form`

This clause collects lists of values into a result list by destructively modifying the lists rather than copying them.

`concat form`

This clause concatenates the values of the specified form into a string. (It and the following clause are extensions to standard Common Lisp.)

`vconcat form`

This clause concatenates the values of the specified form into a vector.

`count form`

This clause counts the number of times the specified form evaluates to a non-`nil` value.

`sum form`

This clause accumulates the sum of the values of the specified form, which must evaluate to a number.

`maximize form`

This clause accumulates the maximum value of the specified form, which must evaluate to a number. The return value is undefined if `maximize` is executed zero times.

`minimize form`

This clause accumulates the minimum value of the specified form.

Accumulation clauses can be followed by `into var' to cause the data to be collected into variable var (which is automatically `let`-bound during the loop) rather than an unnamed temporary variable. Also, `into` accumulations do not automatically imply a return value. The loop must use some explicit mechanism, such as `finally return`, to return the accumulated result.

It is legal for several accumulation clauses of the same type to accumulate into the same place. From Steele:


```
(loop for name in '(fred sue alice joe june)
      for kids in '((bob ken) () () (kris sunshine) ())
      collect name
      append kids)
     => (fred bob ken sue alice joe kris sunshine june)
```

Other Clauses

This section describes the remaining loop clauses.

`with var = value`

This clause binds a variable to a value around the loop, but otherwise leaves the variable alone during the loop. The following loops are basically equivalent:

```
(loop with x = 17 do ...)
(let ((x 17)) (loop do ...))
(loop for x = 17 then x do ...)
```

Naturally, the variable var might be used for some purpose in the rest of the loop. For example:

```
(loop for x in my-list  with res = nil  do (push x res)
      finally return res)
```

This loop inserts the elements of `my-list` at the front of a new list being accumulated in `res`, then returns the list `res` at the end of the loop. The effect is similar to that of a `collect` clause, but the list gets reversed by virtue of the fact that elements are being pushed onto the front of `res` rather than the end.

If you omit the `=` term, the variable is initialized to `nil`. (Thus the `= nil' in the above example is unnecessary.)

Bindings made by `with` are sequential by default, as if by `let*`. Just like `for` clauses, `with` clauses can be linked with `and` to cause the bindings to be made by `let` instead.

`if condition clause`

This clause executes the following loop clause only if the specified condition is true. The following clause should be an accumulation, `do`, `return`, `if`, or `unless` clause. Several clauses may be linked by separating them with `and`. These clauses may be followed by `else` and a clause or clauses to execute if the condition was false. The whole construct may optionally be followed by the word `end` (which may be used to disambiguate an `else` or `and` in a nested `if`).

The actual non-`nil` value of the condition form is available by the name `it` in the "then" part. For example:

```
(setq funny-numbers '(6 13 -1))
     => (6 13 -1)
(loop for x below 10
      if (oddp x)
        collect x into odds
```


```
          and if (memq x funny-numbers) return (cdr it) end
        else
          collect x into evens
        finally return (vector odds evens))
     => [(1 3 5 7 9) (0 2 4 6 8)]
(setq funny-numbers '(6 7 13 -1))
     => (6 7 13 -1)
(loop <same thing again>)
     => (13 -1)
```

Note the use of `and` to put two clauses into the "then" part, one of which is itself an `if` clause. Note also that `end`, while normally optional, was necessary here to make it clear that the `else` refers to the outermost `if` clause. In the first case, the loop returns a vector of lists of the odd and even values of x. In the second case, the odd number 7 is one of the `funny-numbers` so the loop returns early; the actual returned value is based on the result of the `memq` call.

`when condition clause`

This clause is just a synonym for `if`.

`unless condition clause`

The `unless` clause is just like `if` except that the sense of the condition is reversed.

`named name`

This clause gives a name other than `nil` to the implicit block surrounding the loop. The name is the symbol to be used as the block name.

`initially [do] forms...`

This keyword introduces one or more Lisp forms which will be executed before the loop itself begins (but after any variables requested by `for` or `with` have been bound to their initial values). `initially` clauses can appear anywhere; if there are several, they are executed in the order they appear in the loop. The keyword `do` is optional.

`finally [do] forms...`

This introduces Lisp forms which will be executed after the loop finishes (say, on request of a `for` or `while`). `initially` and `finally` clauses may appear anywhere in the loop construct, but they are executed (in the specified order) at the beginning or end, respectively, of the loop.

`finally return form`

This says that form should be executed after the loop is done to obtain a return value. (Without this, or some other clause like `collect` or `return`, the loop will simply return `nil`.) Variables bound by `for`, `with`, or `into` will still contain their final values when form is executed.

`do forms...`

The word `do` may be followed by any number of Lisp expressions which are executed as an implicit `progn` in the body of the loop. Many of the examples in this section illustrate the use of `do`.

`return form`

This clause causes the loop to return immediately. The following Lisp form is evaluated to give the return value of the `loop` form. The `finally` clauses, if any, are not executed. Of course, `return` is generally used inside an `if` or `unless`, as its use in a top-level loop clause would mean the loop would never get to "loop" more than once.

The clause `return form' is equivalent to `do (return form)' (or `return-from` if the loop was nam
The `return` clause is implemented a bit more efficiently, though.

While there is no high-level way to add user extensions to `loop` (comparable to `defsetf` for `setf`, say), this package does offer two properties called `cl-loop-handler` and `cl-loop-for-handler` which are functions to be called when a given symbol is encountered as a top-level loop clause or `for` clause, respectively. Consult the source code in file `` `cl-macs.el' `` for details.

This package's `loop` macro is compatible with that of Common Lisp, except that a few features are not implemented: `loop-finish` and data-type specifiers. Naturally, the `for` clauses which iterate over keymaps, overlays, intervals, frames, windows, and buffers are Emacs-specific extensions.

Multiple Values

Common Lisp functions can return zero or more results. Emacs Lisp functions, by contrast, always return exactly one result. This package makes no attempt to emulate Common Lisp multiple return values; Emacs versions of Common Lisp functions that return more than one value either return just the first value (as in `compiler-macroexpand`) or return a list of values (as in `get-setf-method`). This package *does* define placeholders for the Common Lisp functions that work with multiple values, but in Emacs Lisp these functions simply operate on lists instead. The `values` form, for example, is a synonym for `list` in Emacs.

Special Form: **multiple-value-bind** *(var...) values-form forms...*

This form evaluates values-form, which must return a list of values. It then binds the vars to these respective values, as if by `let`, and then executes the body forms. If there are more vars than values, the extra vars are bound to `nil`. If there are fewer vars than values, the excess values are ignored.

Special Form: **multiple-value-setq** *(var...) form*

This form evaluates form, which must return a list of values. It then sets the vars to these respective values, as if by `setq`. Extra vars or values are treated the same as in `multiple-value-bind`.

The older Quiroz package attempted a more faithful (but still imperfect) emulation of Common Lisp multiple values. The old method "usually" simulated true multiple values quite well, but under certain circumstances would leave spurious return values in memory where a later, unrelated `multiple-value-bind` form would see them.

Since a perfect emulation is not feasible in Emacs Lisp, this package opts to keep it as simple and predictable as possible.

Macros

This package implements the various Common Lisp features of `defmacro`, such as destructuring, `&environment`, and `&body`. Top-level `&whole` is not implemented for `defmacro` due to technical difficulties. See section Argument Lists.

Destructuring is made available to the user by way of the following macro:

Special Form: **destructuring-bind** *arglist expr forms...*

This macro expands to code which executes forms, with the variables in arglist bound to the list of values returned by expr. The arglist can include all the features allowed for `defmacro` argument lists, including destructuring. (The `&environment` keyword is not allowed.) The macro expansion will signal an error if expr returns a list of the wrong number of arguments or with incorrect keyword arguments.

This package also includes the Common Lisp `define-compiler-macro` facility, which allows you to define compile-time expansions and optimizations for your functions.

Special Form: **define-compiler-macro** *name arglist forms...*

This form is similar to `defmacro`, except that it only expands calls to name at compile-time; calls processed by the Lisp interpreter are not expanded, nor are they expanded by the `macroexpand` function.

The argument list may begin with a `&whole` keyword and a variable. This variable is bound to the macro-call form itself, i.e., to a list of the form `(name args...)'. If the macro expander returns this form unchanged, then the compiler treats it as a normal function call. This allows compiler macros to work as optimizers for special cases of a function, leaving complicated cases alone.

For example, here is a simplified version of a definition that appears as a standard part of this package:

```
(define-compiler-macro member* (&whole form a list &rest keys)
  (if (and (null keys)
           (eq (car-safe a) 'quote)
           (not (floatp-safe (cadr a))))
      (list 'memq a list)
    form))
```

This definition causes `(member* a list)` to change to a call to the faster `memq` in the common case where a is a non-floating-point constant; if a is anything else, or if there are any keyword arguments in the call, then the original `member*` call is left intact. (The actual compiler macro for `member*` optimizes a number of other cases, including common `:test` predicates.)

Function: **compiler-macroexpand** *form*

This function is analogous to `macroexpand`, except that it expands compiler macros rather than regular macros. It returns form unchanged if it is not a call to a function for which a compiler macro has been defined, or if that compiler macro decided to punt by returning its `&whole` argument. Like `macroexpand`, it expands repeatedly until it reaches a form for which no further expansion is possible.

See section Macro Bindings, for descriptions of the `macrolet` and `symbol-macrolet` forms for making "local" macro definitions.

Declarations

Common Lisp includes a complex and powerful "declaration" mechanism that allows you to give the compiler special hints about the types of data that will be stored in particular variables, and about the ways those variables and functions will be used. This package defines versions of all the Common Lisp declaration forms: `declare`, `locally`, `proclaim`, `declaim`, and `the`.

Most of the Common Lisp declarations are not currently useful in Emacs Lisp, as the byte-code system provides little opportunity to benefit from type information, and `special` declarations are redundant in a fully dynamically-scoped Lisp. A few declarations are meaningful when the optimizing Emacs 19 byte compiler is being used, however. Under the earlier non-optimizing compiler, these declarations will effectively be ignored.

Function: **proclaim** *decl-spec*

This function records a "global" declaration specified by decl-spec. Since `proclaim` is a function, decl-spec is evaluated and thus should normally be quoted.

Special Form: **declaim** *decl-specs...*

This macro is like `proclaim`, except that it takes any number of decl-spec arguments, and the arguments are unevaluated and unquoted. The `declaim` macro also puts an `(eval-when (compile load eval) ...)` around the declarations so that they will be registered at compile-time as well as at run-time. (This is vital, since normally the declarations are meant to influence the way the compiler treats the rest of the file that contains the `declaim` form.)

Special Form: **declare** *decl-specs...*

This macro is used to make declarations within functions and other code. Common Lisp allows declarations in various locations, generally at the beginning of any of the many "implicit `progn`s" throughout Lisp syntax, such as function bodies, `let` bodies, etc. Currently the only declaration understood by `declare` is `special`.

Special Form: **locally** *declarations... forms...*

In this package, `locally` is no different from `progn`.

Special Form: **the** *type form*

Type information provided by `the` is ignored in this package; in other words, `(the type form)` is equivalent to form. Future versions of the optimizing byte-compiler may make use of this information.

For example, `mapcar` can map over both lists and arrays. It is hard for the compiler to expand `mapcar` into an in-line loop unless it knows whether the sequence will be a list or an array ahead of time. With `(mapcar 'car (the vector foo))`, a future compiler would have enough information to expand the loop in-line. For now, Emacs Lisp will treat the above code as exactly equivalent to `(mapcar 'car foo)`.

Each decl-spec in a `proclaim`, `declaim`, or `declare` should be a list beginning with a symbol that says what kind of declaration it is. This package currently understands `special`, `inline`, `notinline`, `optimize`, and `warn` declarations. (The `warn` declaration is an extension of standard Common Lisp.) Other Common Lisp declarations, such as `type` and `ftype`, are silently ignored.

`special`

Since all variables in Emacs Lisp are "special" (in the Common Lisp sense), `special` declarations are only advisory. They simply tell the optimizing byte compiler that the specified variables are intentionally being referred to without being bound in the body of the function. The compiler normally emits warnings for such references, since they could be typographical errors for references to local variables.

The declaration `(declare (special var1 var2))` is equivalent to `(defvar var1)` `(defvar var2)` in the optimizing compiler, or to nothing at all in older compilers (which do not warn for non-local references).

In top-level contexts, it is generally better to write `(defvar var)` than `(declaim (special var))`, since `defvar` makes your intentions clearer. But the older byte compilers can not handle `defvar`s appearing inside of functions, while `(declare (special var))` takes care to work correctly with all compilers.

`inline`

The `inline` decl-spec lists one or more functions whose bodies should be expanded "in-line" into calling functions whenever the compiler is able to arrange for it. For example, the Common Lisp function `cadr` is declared `inline` by this package so that the form `(cadr x)` will expand directly into `(car (cdr x))` when it is called in user functions, for a savings of one (relatively expensive) function call.

The following declarations are all equivalent. Note that the `defsubst` form is a convenient way to define a function and declare it inline all at once, but it is available only in Emacs 19.

```
(declaim (inline foo bar))
(eval-when (compile load eval) (proclaim '(inline foo bar)))
(proclaim-inline foo bar)      ; Lucid Emacs only
(defsubst foo (...) ...)       ; instead of defun; Emacs 19 only
```

Note: This declaration remains in effect after the containing source file is done. It is correct to use it to request that a function you have defined should be inlined, but it is impolite to use it to request inlining of an external function.

In Common Lisp, it is possible to use `(declare (inline ...))` before a particular call to a function to cause just that call to be inlined; the current byte compilers provide no way to implement this, so `(declare (inline ...))` is currently ignored by this package.

`notinline`

The `notinline` declaration lists functions which should not be inlined after all; it cancels a previous `inline` declaration.

`optimize`

This declaration controls how much optimization is performed by the compiler. Naturally, it is ignored by the earlier non-optimizing compilers.

The word `optimize` is followed by any number of lists like `(speed 3)` or `(safety 2)`. Common Lisp defines several optimization "qualities"; this package ignores all but `speed` and `safety`. The value of a quality should be an integer from 0 to 3, with 0 meaning "unimportant" and 3

meaning "very important." The default level for both qualities is 1.

In this package, with the Emacs 19 optimizing compiler, the `speed` quality is tied to the `byte-compile-optimize` flag, which is set to `nil` for `(speed 0)` and to `t` for higher settings; and the `safety` quality is tied to the `byte-compile-delete-errors` flag, which is set to `t` for `(safety 3)` and to `nil` for all lower settings. (The latter flag controls whether the compiler is allowed to optimize out code whose only side-effect could be to signal an error, e.g., rewriting `(progn foo bar)` to `bar` when it is not known whether `foo` will be bound at run-time.)

Note that even compiling with `(safety 0)`, the Emacs byte-code system provides sufficient checking to prevent real harm from being done. For example, barring serious bugs in Emacs itself, Emacs will not crash with a segmentation fault just because of an error in a fully-optimized Lisp program.

The `optimize` declaration is normally used in a top-level `proclaim` or `declaim` in a file; Common Lisp allows it to be used with `declare` to set the level of optimization locally for a given form, but this will not work correctly with the current version of the optimizing compiler. (The `declare` will set the new optimization level, but that level will not automatically be unset after the enclosing form is done.)

`warn`

This declaration controls what sorts of warnings are generated by the byte compiler. Again, only the optimizing compiler generates warnings. The word `warn` is followed by any number of "warning qualities," similar in form to optimization qualities. The currently supported warning types are `redefine`, `callargs`, `unresolved`, and `free-vars`; in the current system, a value of 0 will disable these warnings and any higher value will enable them. See the documentation for the optimizing byte compiler for details.

Symbols

This package defines several symbol-related features that were missing from Emacs Lisp.

Property Lists

These functions augment the standard Emacs Lisp functions `get` and `put` for operating on properties attached to symbols. There are also functions for working with property lists as first-class data structures not attached to particular symbols.

Function: **get*** *symbol property &optional default*

This function is like `get`, except that if the property is not found, the default argument provides the return value. (The Emacs Lisp `get` function always uses `nil` as the default; this package's `get*` is equivalent to Common Lisp's `get`.)

The `get*` function is `setf`-able; when used in this fashion, the default argument is allowed but ignored.

Function: **remprop** *symbol property*

This function removes the entry for property from the property list of symbol. It returns a true value if the

property was indeed found and removed, or `nil` if there was no such property. (This function was proba omitted from Emacs originally because, since `get` did not allow a default, it was very difficult to disting between a missing property and a property whose value was `nil`; thus, setting a property to `nil` was close enough to `remprop` for most purposes.)

Function: **getf** *place property &optional default*

This function scans the list place as if it were a property list, i.e., a list of alternating property names and values. If an even-numbered element of place is found which is `eq` to property, the following odd-numbered element is returned. Otherwise, default is returned (or `nil` if no default is given).

In particular,

```
(get sym prop)  ==  (getf (symbol-plist sym) prop)
```

It is legal to use `getf` as a `setf` place, in which case its place argument must itself be a legal `setf` place. The default argument, if any, is ignored in this context. The effect is to change (via `setcar`) the value cell in the list that corresponds to property, or to cons a new property-value pair onto the list if the property is not yet present.

```
(put sym prop val)  ==  (setf (getf (symbol-plist sym) prop) val)
```

The `get` and `get*` functions are also `setf`-able. The fact that `default` is ignored can sometimes be useful:

```
(incf (get* 'foo 'usage-count 0))
```

Here, symbol `foo`'s `usage-count` property is incremented if it exists, or set to 1 (an incremented 0) otherwise.

When not used as a `setf` form, `getf` is just a regular function and its place argument can actually be any Lisp expression.

Special Form: **remf** *place property*

This macro removes the property-value pair for property from the property list stored at place, which is any `setf`-able place expression. It returns true if the property was found. Note that if property happens to be first on the list, this will effectively do a `(setf place (cddr place))`, whereas if it occurs later, this simply uses `setcdr` to splice out the property and value cells.

@secno=2

Creating Symbols

These functions create unique symbols, typically for use as temporary variables.

Function: **gensym** *&optional x*

This function creates a new, uninterned symbol (using `make-symbol`) with a unique name. (The name of an uninterned symbol is relevant only if the symbol is printed.) By default, the name is generated from an

increasing sequence of numbers, `G1000', `G1001', `G1002', etc. If the optional argument x is a string, th string is used as a prefix instead of `G'. Uninterned symbols are used in macro expansions for temporary variables, to ensure that their names will not conflict with "real" variables in the user's code.

Variable: ***gensym-counter***

This variable holds the counter used to generate `gensym` names. It is incremented after each use by `gensym`. In Common Lisp this is initialized with 0, but this package initializes it with a random (time-dependent) value to avoid trouble when two files that each used `gensym` in their compilation are loaded together. (Uninterned symbols become interned when the compiler writes them out to a file and the Emacs loader loads them, so their names have to be treated a bit more carefully than in Common Lisp where uninterned symbols remain uninterned after loading.)

Function: **gentemp** *&optional x*

This function is like `gensym`, except that it produces a new *interned* symbol. If the symbol that is generated already exists, the function keeps incrementing the counter and trying again until a new symbol is generated.

The Quiroz `cl.el' package also defined a `defkeyword` form for creating self-quoting keyword symbols. This package automatically creates all keywords that are called for by `&key` argument specifiers, and discourages the use of keywords as data unrelated to keyword arguments, so the `defkeyword` form has been discontinued.

@chapno=11

Numbers

This section defines a few simple Common Lisp operations on numbers which were left out of Emacs Lisp.

@secno=1

Predicates on Numbers

These functions return `t` if the specified condition is true of the numerical argument, or `nil` otherwise.

Function: **plusp** *number*

This predicate tests whether number is positive. It is an error if the argument is not a number.

Function: **minusp** *number*

This predicate tests whether number is negative. It is an error if the argument is not a number.

Function: **oddp** *integer*

This predicate tests whether integer is odd. It is an error if the argument is not an integer.

Function: **evenp** *integer*

This predicate tests whether integer is even. It is an error if the argument is not an integer.

Function: **floatp-safe** *object*

This predicate tests whether object is a floating-point number. On systems that support floating-point, this is equivalent to `floatp`. On other systems, this always returns `nil`.

@secno=3

Numerical Functions

These functions perform various arithmetic operations on numbers.

Function: **abs** *number*

This function returns the absolute value of number. (Newer versions of Emacs provide this as a built-in function; this package defines `abs` only for Emacs 18 versions which don't provide it as a primitive.)

Function: **expt** *base power*

This function returns base raised to the power of number. (Newer versions of Emacs provide this as a built-in function; this package defines `expt` only for Emacs 18 versions which don't provide it as a primitive.)

Function: **gcd** *&rest integers*

This function returns the Greatest Common Divisor of the arguments. For one argument, it returns the absolute value of that argument. For zero arguments, it returns zero.

Function: **lcm** *&rest integers*

This function returns the Least Common Multiple of the arguments. For one argument, it returns the absolute value of that argument. For zero arguments, it returns one.

Function: **isqrt** *integer*

This function computes the "integer square root" of its integer argument, i.e., the greatest integer less than or equal to the true square root of the argument.

Function: **floor*** *number &optional divisor*

This function implements the Common Lisp `floor` function. It is called `floor*` to avoid name conflicts with the simpler `floor` function built-in to Emacs 19.

With one argument, `floor*` returns a list of two numbers: The argument rounded down (toward minus infinity) to an integer, and the "remainder" which would have to be added back to the first return value to yield the argument again. If the argument is an integer x, the result is always the list `(x 0)`. If the argument is an Emacs 19 floating-point number, the first result is a Lisp integer and the second is a Lisp float between 0 (inclusive) and 1 (exclusive).

With two arguments, `floor*` divides number by divisor, and returns the floor of the quotient and the corresponding remainder as a list of two numbers. If `(floor* x y)` returns `(q r)`, then `q*y + r = x`, with r between 0 (inclusive) and r (exclusive). Also, note that `(floor* x)` is exactly equivalent to `(floor* x 1)`.

This function is entirely compatible with Common Lisp's `floor` function, except that it returns the two

results in a list since Emacs Lisp does not support multiple-valued functions.

Function: **ceiling*** *number &optional divisor*

This function implements the Common Lisp `ceiling` function, which is analogous to `floor` except that it rounds the argument or quotient of the arguments up toward plus infinity. The remainder will be between 0 and minus r.

Function: **truncate*** *number &optional divisor*

This function implements the Common Lisp `truncate` function, which is analogous to `floor` except that it rounds the argument or quotient of the arguments toward zero. Thus it is equivalent to `floor*` if the argument or quotient is positive, or to `ceiling*` otherwise. The remainder has the same sign as number.

Function: **round*** *number &optional divisor*

This function implements the Common Lisp `round` function, which is analogous to `floor` except that it rounds the argument or quotient of the arguments to the nearest integer. In the case of a tie (the argument or quotient is exactly halfway between two integers), it rounds to the even integer.

Function: **mod*** *number divisor*

This function returns the same value as the second return value of `floor`.

Function: **rem*** *number divisor*

This function returns the same value as the second return value of `truncate`.

These definitions are compatible with those in the Quiroz `` `cl.el' `` package, except that this package appends `*' to certain function names to avoid conflicts with existing Emacs 19 functions, and that the mechanism for returning multiple values is different.

@secno=8

Random Numbers

This package also provides an implementation of the Common Lisp random number generator. It uses its own additive-congruential algorithm, which is much more likely to give statistically clean random numbers than the simple generators supplied by many operating systems.

Function: **random*** *number &optional state*

This function returns a random nonnegative number less than number, and of the same type (either integer or floating-point). The state argument should be a `random-state` object which holds the state of the random number generator. The function modifies this state object as a side effect. If state is omitted, it defaults to the variable `*random-state*`, which contains a pre-initialized `random-state` object.

Variable: ***random-state***

This variable contains the system "default" `random-state` object, used for calls to `random*` that do not specify an alternative state object. Since any number of programs in the Emacs process may be accessing `*random-state*` in interleaved fashion, the sequence generated from this variable will be irreproducible for all intents and purposes.

Function: **make-random-state** *&optional state*

This function creates or copies a `random-state` object. If state is omitted or `nil`, it returns a new copy of `*random-state*`. This is a copy in the sense that future sequences of calls to `(random* n)` and `(random* n s)` (where s is the new random-state object) will return identical sequences of random numbers.

If state is a `random-state` object, this function returns a copy of that object. If state is `t`, this function returns a new `random-state` object seeded from the date and time. As an extension to Common Lisp, state may also be an integer in which case the new object is seeded from that integer; each different integer seed will result in a completely different sequence of random numbers.

It is legal to print a `random-state` object to a buffer or file and later read it back with `read`. If a program wishes to use a sequence of pseudo-random numbers which can be reproduced later for debugging, it can call `(make-random-state t)` to get a new sequence, then print this sequence to a file. When the program is later rerun, it can read the original run's random-state from the file.

Function: **random-state-p** *object*

This predicate returns `t` if object is a `random-state` object, or `nil` otherwise.

Implementation Parameters

This package defines several useful constants having to with numbers.

Variable: **most-positive-fixnum**

This constant equals the largest value a Lisp integer can hold. It is typically `2^23-1` or `2^25-1`.

Variable: **most-negative-fixnum**

This constant equals the smallest (most negative) value a Lisp integer can hold.

The following parameters have to do with floating-point numbers. This package determines their values by exercising the computer's floating-point arithmetic in various ways. Because this operation might be slow, the code for initializing them is kept in a separate function that must be called before the parameters can be used.

Function: **cl-float-limits**

This function makes sure that the Common Lisp floating-point parameters like `most-positive-float` have been initialized. Until it is called, these parameters will be `nil`. If this version of Emacs does not support floats (e.g., most versions of Emacs 18), the parameters will remain `nil`. If the parameters have already been initialized, the function returns immediately.

The algorithm makes assumptions that will be valid for most modern machines, but will fail if the machine's arithmetic is extremely unusual, e.g., decimal.

Since true Common Lisp supports up to four different floating-point precisions, it has families of constants like `most-positive-single-float`, `most-positive-double-float`, `most-positive-long-float`, and so on. Emacs has only one floating-point precision, so this package omits the precision word from the constants' names.

TOC

Variable: **most-positive-float**

This constant equals the largest value a Lisp float can hold. For those systems whose arithmetic supports infinities, this is the largest *finite* value. For IEEE machines, the value is approximately `1.79e+308`.

Variable: **most-negative-float**

This constant equals the most-negative value a Lisp float can hold. (It is assumed to be equal to `(-most-positive-float)`.)

Variable: **least-positive-float**

This constant equals the smallest Lisp float value greater than zero. For IEEE machines, it is about `4.94e-324` if denormals are supported or `2.22e-308` if not.

Variable: **least-positive-normalized-float**

This constant equals the smallest *normalized* Lisp float greater than zero, i.e., the smallest value for which IEEE denormalization will not result in a loss of precision. For IEEE machines, this value is about `2.22e-308`. For machines that do not support the concept of denormalization and gradual underflow, this constant will always equal `least-positive-float`.

Variable: **least-negative-float**

This constant is the negative counterpart of `least-positive-float`.

Variable: **least-negative-normalized-float**

This constant is the negative counterpart of `least-positive-normalized-float`.

Variable: **float-epsilon**

This constant is the smallest positive Lisp float that can be added to 1.0 to produce a distinct value. Adding a smaller number to 1.0 will yield 1.0 again due to roundoff. For IEEE machines, epsilon is about `2.22e-16`.

Variable: **float-negative-epsilon**

This is the smallest positive value that can be subtracted from 1.0 to produce a distinct value. For IEEE machines, it is about `1.11e-16`.

@chapno=13

Sequences

Common Lisp defines a number of functions that operate on sequences, which are either lists, strings, or vectors. Emacs Lisp includes a few of these, notably `elt` and `length`; this package defines most of the rest.

Sequence Basics

Many of the sequence functions take keyword arguments; see section Argument Lists. All keyword arguments are optional and, if specified, may appear in any order.

The `:key` argument should be passed either `nil`, or a function of one argument. This key function is used as a filter through which the elements of the sequence are seen; for example, `(find x y :key 'car)` is similar to `(assoc* x y)`: It searches for an element of the list whose `car` equals `x`, rather than for an element which equals `x` itself. If `:key` is omitted or `nil`, the filter is effectively the identity function.

The `:test` and `:test-not` arguments should be either `nil`, or functions of two arguments. The test function is used to compare two sequence elements, or to compare a search value with sequence elements. (The two values are passed to the test function in the same order as the original sequence function arguments from which they are derived, or, if they both come from the same sequence, in the same order as they appear in that sequence.) The `:test` argument specifies a function which must return true (non-`nil`) to indicate a match; instead, you may use `:test-not` to give a function which returns *false* to indicate a match. The default test function is `:test 'eql`.

Many functions which take item and `:test` or `:test-not` arguments also come in `-if` and `-if-not` varieties, where a predicate function is passed instead of item, and sequence elements match if the predicate returns true on them (or false in the case of `-if-not`). For example:

```
(remove* 0 seq :test '=)  ==  (remove-if 'zerop seq)
```

to remove all zeros from sequence `seq`.

Some operations can work on a subsequence of the argument sequence; these function take `:start` and `:end` arguments which default to zero and the length of the sequence, respectively. Only elements between start (inclusive) and end (exclusive) are affected by the operation. The end argument may be passed `nil` to signify the length of the sequence; otherwise, both start and end must be integers, with `0 <= start <= end <= (length seq)`. If the function takes two sequence arguments, the limits are defined by keywords `:start1` and `:end1` for the first, and `:start2` and `:end2` for the second.

A few functions accept a `:from-end` argument, which, if non-`nil`, causes the operation to go from right-to-left through the sequence instead of left-to-right, and a `:count` argument, which specifies an integer maximum number of elements to be removed or otherwise processed.

The sequence functions make no guarantees about the order in which the `:test`, `:test-not`, and `:key` functions are called on various elements. Therefore, it is a bad idea to depend on side effects of these functions. For example, `:from-end` may cause the sequence to be scanned actually in reverse, or it may be scanned forwards but computing a result "as if" it were scanned backwards. (Some functions, like `mapcar*` and `every`, *do* specify exactly the order in which the function is called so side effects are perfectly acceptable in those cases.)

Strings in GNU Emacs 19 may contain "text properties" as well as character data. Except as noted, it is undefined whether or not text properties are preserved by sequence functions. For example, `(remove* ?A str)` may or may not preserve the properties of the characters copied from str into the result.

Mapping over Sequences

These functions "map" the function you specify over the elements of lists or arrays. They are all variations on the theme of the built-in function `mapcar`.

Function: **mapcar*** *function seq &rest more-seqs*

This function calls function on successive parallel sets of elements from its argument sequences. Given a single seq argument it is equivalent to `mapcar`; given n sequences, it calls the function with the first elements of each of the sequences as the n arguments to yield the first element of the result list, then with the second elements, and so on. The mapping stops as soon as the shortest sequence runs out. The argument sequences may be any mixture of lists, strings, and vectors; the return sequence is always a list.

Common Lisp's `mapcar` accepts multiple arguments but works only on lists; Emacs Lisp's `mapcar` accepts a single sequence argument. This package's `mapcar*` works as a compatible superset of both.

Function: **map** *result-type function seq &rest more-seqs*

This function maps function over the argument sequences, just like `mapcar*`, but it returns a sequence of type result-type rather than a list. result-type must be one of the following symbols: `vector`, `string`, `list` (in which case the effect is the same as for `mapcar*`), or `nil` (in which case the results are thrown away and `map` returns `nil`).

Function: **maplist** *function list &rest more-lists*

This function calls function on each of its argument lists, then on the `cdr`s of those lists, and so on, until the shortest list runs out. The results are returned in the form of a list. Thus, `maplist` is like `mapcar*` except that it passes in the list pointers themselves rather than the `car`s of the advancing pointers.

Function: **mapc** *function seq &rest more-seqs*

This function is like `mapcar*`, except that the values returned by function are ignored and thrown away rather than being collected into a list. The return value of `mapc` is seq, the first sequence.

Function: **mapl** *function list &rest more-lists*

This function is like `maplist`, except that it throws away the values returned by function.

Function: **mapcan** *function seq &rest more-seqs*

This function is like `mapcar*`, except that it concatenates the return values (which must be lists) using `nconc`, rather than simply collecting them into a list.

Function: **mapcon** *function list &rest more-lists*

This function is like `maplist`, except that it concatenates the return values using `nconc`.

Function: **some** *predicate seq &rest more-seqs*

This function calls predicate on each element of seq in turn; if predicate returns a non-`nil` value, `some` returns that value, otherwise it returns `nil`. Given several sequence arguments, it steps through the sequences in parallel until the shortest one runs out, just as in `mapcar*`. You can rely on the left-to-right order in which the elements are visited, and on the fact that mapping stops immediately as soon as predicate

returns non-`nil`.

Function: **every** *predicate seq &rest more-seqs*

This function calls predicate on each element of the sequence(s) in turn; it returns `nil` as soon as predicate returns `nil` for any element, or `t` if the predicate was true for all elements.

Function: **notany** *predicate seq &rest more-seqs*

This function calls predicate on each element of the sequence(s) in turn; it returns `nil` as soon as predicate returns a non-`nil` value for any element, or `t` if the predicate was `nil` for all elements.

Function: **notevery** *predicate seq &rest more-seqs*

This function calls predicate on each element of the sequence(s) in turn; it returns a non-`nil` value as soon as predicate returns `nil` for any element, or `t` if the predicate was true for all elements.

Function: **reduce** *function seq* `&key :from-end :start :end :initial-value :key`

This function combines the elements of seq using an associative binary operation. Suppose function is `*` and seq is the list `(2 3 4 5)`. The first two elements of the list are combined with `(* 2 3) = 6`; this is combined with the next element, `(* 6 4) = 24`, and that is combined with the final element: `(* 24 5) = 120`. Note that the `*` function happens to be self-reducing, so that `(* 2 3 4 5)` has the same effect as an explicit call to `reduce`.

If `:from-end` is true, the reduction is right-associative instead of left-associative:

```
(reduce '- '(1 2 3 4))
     == (- (- (- 1 2) 3) 4) => -8
(reduce '- '(1 2 3 4) :from-end t)
     == (- 1 (- 2 (- 3 4))) => -2
```

If `:key` is specified, it is a function of one argument which is called on each of the sequence elements in turn.

If `:initial-value` is specified, it is effectively added to the front (or rear in the case of `:from-end`) of the sequence. The `:key` function is *not* applied to the initial value.

If the sequence, including the initial value, has exactly one element then that element is returned without ever calling function. If the sequence is empty (and there is no initial value), then function is called with no arguments to obtain the return value.

All of these mapping operations can be expressed conveniently in terms of the `loop` macro. In compiled code, `loop` will be faster since it generates the loop as in-line code with no function calls.

Sequence Functions

This section describes a number of Common Lisp functions for operating on sequences.

Function: **subseq** *sequence start &optional end*

This function returns a given subsequence of the argument sequence, which may be a list, string, or vector.

The indices start and end must be in range, and start must be no greater than end. If end is omitted, it de to the length of the sequence. The return value is always a copy; it does not share structure with sequenc

As an extension to Common Lisp, start and/or end may be negative, in which case they represent a distance back from the end of the sequence. This is for compatibility with Emacs' `substring` function. Note that `subseq` is the *only* sequence function that allows negative start and end.

You can use `setf` on a `subseq` form to replace a specified range of elements with elements from another sequence. The replacement is done as if by `replace`, described below.

Function: **concatenate** *result-type &rest seqs*

This function concatenates the argument sequences together to form a result sequence of type result-type, one of the symbols `vector`, `string`, or `list`. The arguments are always copied, even in cases such as `(concatenate 'list '(1 2 3))` where the result is identical to an argument.

Function: **fill** *seq item* `&key :start :end`

This function fills the elements of the sequence (or the specified part of the sequence) with the value item.

Function: **replace** *seq1 seq2* `&key :start1 :end1 :start2 :end2`

This function copies part of seq2 into part of seq1. The sequence seq1 is not stretched or resized; the amount of data copied is simply the shorter of the source and destination (sub)sequences. The function returns seq1.

If seq1 and seq2 are `eq`, then the replacement will work correctly even if the regions indicated by the start and end arguments overlap. However, if seq1 and seq2 are lists which share storage but are not `eq`, and the start and end arguments specify overlapping regions, the effect is undefined.

Function: **remove*** *item seq* `&key :test :test-not :key :count :start :end :from-end`

This returns a copy of seq with all elements matching item removed. The result may share storage with or be `eq` to seq in some circumstances, but the original seq will not be modified. The `:test`, `:test-not`, and `:key` arguments define the matching test that is used; by default, elements `eql` to item are removed. The `:count` argument specifies the maximum number of matching elements that can be removed (only the leftmost count matches are removed). The `:start` and `:end` arguments specify a region in seq in which elements will be removed; elements outside that region are not matched or removed. The `:from-end` argument, if true, says that elements should be deleted from the end of the sequence rather than the beginning (this matters only if count was also specified).

Function: **delete*** *item seq* `&key :test :test-not :key :count :start :end :from-end`

This deletes all elements of seq which match item. It is a destructive operation. Since Emacs Lisp does not support stretchable strings or vectors, this is the same as `remove*` for those sequence types. On lists, `remove*` will copy the list if necessary to preserve the original list, whereas `delete*` will splice out parts of the argument list. Compare `append` and `nconc`, which are analogous non-destructive and destructive list operations in Emacs Lisp.

The predicate-oriented functions `remove-if`, `remove-if-not`, `delete-if`, and `delete-if-not` are defined similarly.

Function: **delete** *item list*

This MacLisp-compatible function deletes from list all elements which are `equal` to item. The `delete` function is built-in to Emacs 19; this package defines it equivalently in Emacs 18.

Function: **remove** *item list*

This function removes from list all elements which are `equal` to item. This package defines it for symmetry with `delete`, even though `remove` is not built-in to Emacs 19.

Function: **remq** *item list*

This function removes from list all elements which are `eq` to item. This package defines it for symmetry with `delq`, even though `remq` is not built-in to Emacs 19.

Function: **remove-duplicates** *seq* `&key :test :test-not :key :start :end :from-end`

This function returns a copy of seq with duplicate elements removed. Specifically, if two elements from the sequence match according to the `:test`, `:test-not`, and `:key` arguments, only the rightmost one is retained. If `:from-end` is true, the leftmost one is retained instead. If `:start` or `:end` is specified, only elements within that subsequence are examined or removed.

Function: **delete-duplicates** *seq* `&key :test :test-not :key :start :end :from-end`

This function deletes duplicate elements from seq. It is a destructive version of `remove-duplicates`.

Function: **substitute** *new old seq* `&key :test :test-not :key :count :start :end :from-end`

This function returns a copy of seq, with all elements matching old replaced with new. The `:count`, `:start`, `:end`, and `:from-end` arguments may be used to limit the number of substitutions made.

Function: **nsubstitute** *new old seq* `&key :test :test-not :key :count :start :end :from-end`

This is a destructive version of `substitute`; it performs the substitution using `setcar` or `aset` rather than by returning a changed copy of the sequence.

The `substitute-if`, `substitute-if-not`, `nsubstitute-if`, and `nsubstitute-if-not` functions are defined similarly. For these, a predicate is given in place of the old argument.

Searching Sequences

These functions search for elements or subsequences in a sequence. (See also `member*` and `assoc*`; see section Lists.)

Function: **find** *item seq* `&key :test :test-not :key :start :end :from-end`

This function searches seq for an element matching item. If it finds a match, it returns the matching element. Otherwise, it returns `nil`. It returns the leftmost match, unless `:from-end` is true, in which case it returns the rightmost match. The `:start` and `:end` arguments may be used to limit the range of elements that are searched.

Function: **position** *item seq* `&key :test :test-not :key :start :end :from-end`

This function is like `find`, except that it returns the integer position in the sequence of the matching item rather than the item itself. The position is relative to the start of the sequence as a whole, even if `:start` is non-zero. The function returns `nil` if no matching element was found.

Function: **count** *item seq* `&key :test :test-not :key :start :end`

This function returns the number of elements of seq which match item. The result is always a nonnegative integer.

The `find-if`, `find-if-not`, `position-if`, `position-if-not`, `count-if`, and `count-if-not` functions are defined similarly.

Function: **mismatch** *seq1 seq2* `&key :test :test-not :key :start1 :end1 :start2 :end2 :from-end`

This function compares the specified parts of seq1 and seq2. If they are the same length and the corresponding elements match (according to `:test`, `:test-not`, and `:key`), the function returns `nil`. If there is a mismatch, the function returns the index (relative to seq1) of the first mismatching element. This will be the leftmost pair of elements which do not match, or the position at which the shorter of the two otherwise-matching sequences runs out.

If `:from-end` is true, then the elements are compared from right to left starting at `(1- end1)` and `(1- end2)`. If the sequences differ, then one plus the index of the rightmost difference (relative to seq1) is returned.

An interesting example is `(mismatch str1 str2 :key 'upcase)`, which compares two strings case-insensitively.

Function: **search** *seq1 seq2* `&key :test :test-not :key :from-end :start1 :end1 :start2 :end2`

This function searches seq2 for a subsequence that matches seq1 (or part of it specified by `:start1` and `:end1`.) Only matches which fall entirely within the region defined by `:start2` and `:end2` will be considered. The return value is the index of the leftmost element of the leftmost match, relative to the start of seq2, or `nil` if no matches were found. If `:from-end` is true, the function finds the *rightmost* matching subsequence.

Sorting Sequences

Function: **sort*** *seq predicate* `&key :key`

This function sorts seq into increasing order as determined by using predicate to compare pairs of elements. predicate should return true (non-`nil`) if and only if its first argument is less than (not equal to) its second argument. For example, `<` and `string-lessp` are suitable predicate functions for sorting numbers and strings, respectively; `>` would sort numbers into decreasing rather than increasing order.

This function differs from Emacs' built-in `sort` in that it can operate on any type of sequence, not just lists. Also, it accepts a `:key` argument which is used to preprocess data fed to the predicate function. For example,


```
(setq data (sort data 'string-lessp :key 'downcase))
```

sorts data, a sequence of strings, into increasing alphabetical order without regard to case. A `:key` function of `car` would be useful for sorting association lists.

The `sort*` function is destructive; it sorts lists by actually rearranging the `cdr` pointers in suitable fashion.

Function: **stable-sort** *seq predicate* `&key :key`

This function sorts seq stably, meaning two elements which are equal in terms of predicate are guaranteed not to be rearranged out of their original order by the sort.

In practice, `sort*` and `stable-sort` are equivalent in Emacs Lisp because the underlying `sort` function is stable by default. However, this package reserves the right to use non-stable methods for `sort*` in the future.

Function: **merge** *type seq1 seq2 predicate* `&key :key`

This function merges two sequences seq1 and seq2 by interleaving their elements. The result sequence, of type type (in the sense of `concatenate`), has length equal to the sum of the lengths of the two input sequences. The sequences may be modified destructively. Order of elements within seq1 and seq2 is preserved in the interleaving; elements of the two sequences are compared by predicate (in the sense of `sort`) and the lesser element goes first in the result. When elements are equal, those from seq1 precede those from seq2 in the result. Thus, if seq1 and seq2 are both sorted according to predicate, then the result will be a merged sequence which is (stably) sorted according to predicate.

Lists

The functions described here operate on lists.

List Functions

This section describes a number of simple operations on lists, i.e., chains of cons cells.

Function: **caddr** *x*

This function is equivalent to `(car (cdr (cdr x)))`. Likewise, this package defines all 28 `cxxxr` functions where xxx is up to four `a's and/or `d's. All of these functions are `setf`-able, and calls to them are expanded inline by the byte-compiler for maximum efficiency.

Function: **first** *x*

This function is a synonym for `(car x)`. Likewise, the functions `second`, `third`, ..., through `tenth` return the given element of the list x.

Function: **rest** *x*

This function is a synonym for `(cdr x)`.

Function: **endp** *x*

Common Lisp defines this function to act like `null`, but signalling an error if `x` is neither a `nil` nor a c cell. This package simply defines `endp` as a synonym for `null`.

Function: **list-length** *x*

This function returns the length of list x, exactly like `(length x)`, except that if x is a circular list (where the cdr-chain forms a loop rather than terminating with `nil`), this function returns `nil`. (The regular `length` function would get stuck if given a circular list.)

Function: **last** *x &optional n*

This function returns the last cons, or the nth-to-last cons, of the list x. If n is omitted it defaults to 1. The "last cons" means the first cons cell of the list whose `cdr` is not another cons cell. (For normal lists, the `cdr` of the last cons will be `nil`.) This function returns `nil` if x is `nil` or shorter than n. Note that the last *element* of the list is `(car (last x))`.

Function: **butlast** *x &optional n*

This function returns the list x with the last element, or the last n elements, removed. If n is greater than zero it makes a copy of the list so as not to damage the original list. In general, `(append (butlast x n) (last x n))` will return a list equal to x.

Function: **nbutlast** *x &optional n*

This is a version of `butlast` that works by destructively modifying the `cdr` of the appropriate element, rather than making a copy of the list.

Function: **list*** *arg &rest others*

This function constructs a list of its arguments. The final argument becomes the `cdr` of the last cell constructed. Thus, `(list* a b c)` is equivalent to `(cons a (cons b c))`, and `(list* a b nil)` is equivalent to `(list a b)`.

(Note that this function really is called `list*` in Common Lisp; it is not a name invented for this package like `member*` or `defun*`.)

Function: **ldiff** *list sublist*

If sublist is a sublist of list, i.e., is `eq` to one of the cons cells of list, then this function returns a copy of the part of list up to but not including sublist. For example, `(ldiff x (cddr x))` returns the first two elements of the list `x`. The result is a copy; the original list is not modified. If sublist is not a sublist of list, a copy of the entire list is returned.

Function: **copy-list** *list*

This function returns a copy of the list list. It copies dotted lists like `(1 2 . 3)` correctly.

Function: **copy-tree** *x &optional vecp*

This function returns a copy of the tree of cons cells x. Unlike `copy-sequence` (and its alias `copy-list`), which copies only along the `cdr` direction, this function copies (recursively) along both the `car` and the `cdr` directions. If x is not a cons cell, the function simply returns x unchanged. If the optional vecp argument is true, this function copies vectors (recursively) as well as cons cells.

Function: **tree-equal** *x y* `&key :test :test-not :key`

This function compares two trees of cons cells. If x and y are both cons cells, their `car`s and `cdr`s are compared recursively. If neither x nor y is a cons cell, they are compared by `eql`, or according to the specified test. The `:key` function, if specified, is applied to the elements of both trees. See section Sequences.

@secno=3

Substitution of Expressions

These functions substitute elements throughout a tree of cons cells. (See section Sequence Functions, for the `substitute` function, which works on just the top-level elements of a list.)

Function: **subst** *new old tree* `&key :test :test-not :key`

This function substitutes occurrences of old with new in tree, a tree of cons cells. It returns a substituted tree, which will be a copy except that it may share storage with the argument tree in parts where no substitutions occurred. The original tree is not modified. This function recurses on, and compares against old, both `car`s and `cdr`s of the component cons cells. If old is itself a cons cell, then matching cells in the tree are substituted as usual without recursively substituting in that cell. Comparisons with old are done according to the specified test (`eql` by default). The `:key` function is applied to the elements of the tree but not to old.

Function: **nsubst** *new old tree* `&key :test :test-not :key`

This function is like `subst`, except that it works by destructive modification (by `setcar` or `setcdr`) rather than copying.

The `subst-if`, `subst-if-not`, `nsubst-if`, and `nsubst-if-not` functions are defined similarly.

Function: **sublis** *alist tree* `&key :test :test-not :key`

This function is like `subst`, except that it takes an association list alist of old-new pairs. Each element of the tree (after applying the `:key` function, if any), is compared with the `car`s of alist; if it matches, it is replaced by the corresponding `cdr`.

Function: **nsublis** *alist tree* `&key :test :test-not :key`

This is a destructive version of `sublis`.

Lists as Sets

These functions perform operations on lists which represent sets of elements.

Function: **member** *item list*

This MacLisp-compatible function searches list for an element which is `equal` to item. The `member` function is built-in to Emacs 19; this package defines it equivalently in Emacs 18. See the following function for a Common-Lisp compatible version.

Function: **member*** *item list* `&key :test :test-not :key`

This function searches list for an element matching item. If a match is found, it returns the cons cell whose `car` was the matching element. Otherwise, it returns `nil`. Elements are compared by `eql` by default; you can use the `:test`, `:test-not`, and `:key` arguments to modify this behavior. See section Sequences.

Note that this function's name is suffixed by `*' to avoid the incompatible `member` function defined in Emacs 19. (That function uses `equal` for comparisons; it is equivalent to `(member* item list :test 'equal)`.)

The `member-if` and `member-if-not` functions analogously search for elements which satisfy a given predicate.

Function: **tailp** *sublist list*

This function returns `t` if sublist is a sublist of list, i.e., if sublist is `eql` to list or to any of its `cdr`s.

Function: **adjoin** *item list* `&key :test :test-not :key`

This function conses item onto the front of list, like `(cons item list)`, but only if item is not already present on the list (as determined by `member*`). If a `:key` argument is specified, it is applied to item as well as to the elements of list during the search, on the reasoning that item is "about" to become part of the list.

Function: **union** *list1 list2* `&key :test :test-not :key`

This function combines two lists which represent sets of items, returning a list that represents the union of those two sets. The result list will contain all items which appear in list1 or list2, and no others. If an item appears in both list1 and list2 it will be copied only once. If an item is duplicated in list1 or list2, it is undefined whether or not that duplication will survive in the result list. The order of elements in the result list is also undefined.

Function: **nunion** *list1 list2* `&key :test :test-not :key`

This is a destructive version of `union`; rather than copying, it tries to reuse the storage of the argument lists if possible.

Function: **intersection** *list1 list2* `&key :test :test-not :key`

This function computes the intersection of the sets represented by list1 and list2. It returns the list of items which appear in both list1 and list2.

Function: **nintersection** *list1 list2* `&key :test :test-not :key`

This is a destructive version of `intersection`. It tries to reuse storage of list1 rather than copying. It does *not* reuse the storage of list2.

Function: **set-difference** *list1 list2* `&key :test :test-not :key`

This function computes the "set difference" of list1 and list2, i.e., the set of elements that appear in list1 but *not* in list2.

Function: **nset-difference** *list1 list2* `&key :test :test-not :key`

This is a destructive `set-difference`, which will try to reuse list1 if possible.

Function: **set-exclusive-or** *list1 list2* `&key :test :test-not :key`

This function computes the "set exclusive or" of list1 and list2, i.e., the set of elements that appear in exactly one of list1 and list2.

Function: **nset-exclusive-or** *list1 list2* `&key :test :test-not :key`

This is a destructive `set-exclusive-or`, which will try to reuse list1 and list2 if possible.

Function: **subsetp** *list1 list2* `&key :test :test-not :key`

This function checks whether list1 represents a subset of list2, i.e., whether every element of list1 also appears in list2.

Association Lists

An association list is a list representing a mapping from one set of values to another; any list whose elements are cons cells is an association list.

Function: **assoc*** *item a-list* `&key :test :test-not :key`

This function searches the association list a-list for an element whose `car` matches (in the sense of `:test`, `:test-not`, and `:key`, or by comparison with `eql`) a given item. It returns the matching element, if any, otherwise `nil`. It ignores elements of a-list which are not cons cells. (This corresponds to the behavior of `assq` and `assoc` in Emacs Lisp; Common Lisp's `assoc` ignores `nil`s but considers any other non-cons elements of a-list to be an error.)

Function: **rassoc*** *item a-list* `&key :test :test-not :key`

This function searches for an element whose `cdr` matches item. If a-list represents a mapping, this applies the inverse of the mapping to item.

Function: **rassoc** *item a-list*

This function searches like `rassoc*` with a `:test` argument of `equal`. It is analogous to Emacs Lisp's standard `assoc` function, which derives from the MacLisp rather than the Common Lisp tradition.

The `assoc-if`, `assoc-if-not`, `rassoc-if`, and `rassoc-if-not` functions are defined similarly.

Two simple functions for constructing association lists are:

Function: **acons** *key value alist*

This is equivalent to `(cons (cons key value) alist)`.

Function: **pairlis** *keys values &optional alist*

This is equivalent to `(nconc (mapcar* 'cons keys values) alist)`.

Hash Tables

A hash table is a data structure that maps "keys" onto "values." Keys and values can be arbitrary Lisp data objects. Hash tables have the property that the time to search for a given key is roughly constant; simpler data structures like association lists take time proportional to the number of entries in the list.

Function: **make-hash-table** *&key :test :size*

This function creates and returns a hash-table object whose function for comparing elements is `:test` (`eql` by default), and which is allocated to fit about `:size` elements. The `:size` argument is purely advisory; the table will stretch automatically if you store more elements in it. If `:size` is omitted, a reasonable default is used.

Common Lisp allows only `eq`, `eql`, `equal`, and `equalp` as legal values for the `:test` argument. In this package, any reasonable predicate function will work, though if you use something else you should check the details of the hashing function described below to make sure it is suitable for your predicate.

Some versions of Emacs (like Lucid Emacs 19) include a built-in hash table type; in these versions, `make-hash-table` with a test of `eq` will use these built-in hash tables. In all other cases, it will return a hash-table object which takes the form of a list with an identifying "tag" symbol at the front. All of the hash table functions in this package can operate on both types of hash table; normally you will never know which type is being used.

This function accepts the additional Common Lisp keywords `:rehash-size` and `:rehash-threshold`, but it ignores their values.

Function: **gethash** *key table &optional default*

This function looks up key in table. If key exists in the table, in the sense that it matches any of the existing keys according to the table's test function, then the associated value is returned. Otherwise, default (or `nil`) is returned.

To store new data in the hash table, use `setf` on a call to `gethash`. If key already exists in the table, the corresponding value is changed to the stored value. If key does not already exist, a new entry is added to the table and the table is reallocated to a larger size if necessary. The default argument is allowed but ignored in this case. The situation is exactly analogous to that of `get*`; see section Property Lists.

Function: **remhash** *key table*

This function removes the entry for key from table. If an entry was removed, it returns `t`. If key does not appear in the table, it does nothing and returns `nil`.

Function: **clrhash** *table*

This function removes all the entries from table, leaving an empty hash table.

Function: **maphash** *function table*

This function calls function for each entry in table. It passes two arguments to function, the key and the value of the given entry. The return value of function is ignored; maphash itself returns `nil`. See section Loop Facility, for an alternate way of iterating over hash tables.

Function: **hash-table-count** *table*

This function returns the number of entries in table. **Warning:** The current implementation of Lucid Emacs 19 hash-tables does not decrement the stored `count` when `remhash` removes an entry. Therefore, the return value of this function is not dependable if you have used `remhash` on the table and the table's test is `eq`. A slower, but reliable, way to count the entries is `(loop for x being the hash-keys of table count t)`.

Function: **hash-table-p** *object*

This function returns `t` if object is a hash table, `nil` otherwise. It recognizes both types of hash tables (both Lucid Emacs built-in tables and tables implemented with special lists.)

Sometimes when dealing with hash tables it is useful to know the exact "hash function" that is used. This package implements hash tables using Emacs Lisp "obarrays," which are the same data structure that Emacs Lisp uses to keep track of symbols. Each hash table includes an embedded obarray. Key values given to `gethash` are converted by various means into strings, which are then looked up in the obarray using `intern` and `intern-soft`. The symbol, or "bucket," corresponding to a given key string includes as its `symbol-value` an association list of all key-value pairs which hash to that string. Depending on the test function, it is possible for many entries to hash to the same bucket. For example, if the test is `eql`, then the symbol `foo` and two separately built strings `"foo"` will create three entries in the same bucket. Search time is linear within buckets, so hash tables will be most effective if you arrange not to store too many things that hash the same.

The following algorithm is used to convert Lisp objects to hash strings:

- Strings are used directly as hash strings. (However, if the test function is `equalp`, strings are `downcase`d first.)
- Symbols are hashed according to their `symbol-name`.
- Integers are hashed into one of 16 buckets depending on their value modulo 16. Floating-point numbers are truncated to integers and hashed modulo 16.
- Cons cells are hashed according to their `car`s; nonempty vectors are hashed according to their first element.
- All other types of objects hash into a single bucket named `"*"`.

Thus, for example, searching among many buffer objects in a hash table will devolve to a (still fairly fast) linear-time search through a single bucket, whereas searching for different symbols will be very fast since each symbol will, in general, hash into its own bucket.

The size of the obarray in a hash table is automatically adjusted as the number of elements increases.

As a special case, `make-hash-table` with a `:size` argument of 0 or 1 will create a hash-table object that uses a single association list rather than an obarray of many lists. For very small tables this structure will be more efficient since lookup does not require converting the key to a string or looking it up in an obarray. However, such tables are guaranteed to take time proportional to their size to do a search.

@chapno=18

Structures

The Common Lisp structure mechanism provides a general way to define data types similar to C's `struct` types. A structure is a Lisp object containing some number of slots, each of which can hold any Lisp data object. Functions are provided for accessing and setting the slots, creating or copying structure objects, and recognizing objects of a particular structure type.

In true Common Lisp, each structure type is a new type distinct from all existing Lisp types. Since the underlying Emacs Lisp system provides no way to create new distinct types, this package implements structures as vectors (or lists upon request) with a special "tag" symbol to identify them.

Special Form: **defstruct** *name slots...*

The `defstruct` form defines a new structure type called name, with the specified slots. (The slots may begin with a string which documents the structure type.) In the simplest case, name and each of the slots are symbols. For example,

```
(defstruct person name age sex)
```

defines a struct type called `person` which contains three slots. Given a `person` object p, you can access those slots by calling `(person-name p)`, `(person-age p)`, and `(person-sex p)`. You can also change these slots by using `setf` on any of these place forms:

```
(incf (person-age birthday-boy))
```

You can create a new `person` by calling `make-person`, which takes keyword arguments `:name`, `:age`, and `:sex` to specify the initial values of these slots in the new object. (Omitting any of these arguments leaves the corresponding slot "undefined," according to the Common Lisp standard; in Emacs Lisp, such uninitialized slots are filled with `nil`.)

Given a `person`, `(copy-person p)` makes a new object of the same type whose slots are `eq` to those of p.

Given any Lisp object x, `(person-p x)` returns true if x looks like a `person`, false otherwise. (Again, in Common Lisp this predicate would be exact; in Emacs Lisp the best it can do is verify that x is a vector of the correct length which starts with the correct tag symbol.)

Accessors like `person-name` normally check their arguments (effectively using `person-p`) and signal an error if the argument is the wrong type. This check is affected by `(optimize (safety ...))` declarations. Safety level 1, the default, uses a somewhat optimized check that will detect all incorrect arguments, but may use an uninformative error message (e.g., "expected a vector" instead of "expected a `person`"). Safety level 0 omits all checks except as provided by the underlying `aref` call; safety levels 2 and 3 do rigorous checking that will always print a descriptive error message for incorrect inputs. See section Declarations.

```
(setq dave (make-person :name "Dave" :sex 'male))
     => [cl-struct-person "Dave" nil male]
(setq other (copy-person dave))
```


```
     => [cl-struct-person "Dave" nil male]
(eq dave other)
     => nil
(eq (person-name dave) (person-name other))
     => t
(person-p dave)
     => t
(person-p [1 2 3 4])
     => nil
(person-p "Bogus")
     => nil
(person-p '[cl-struct-person counterfeit person object])
     => t
```

In general, name is either a name symbol or a list of a name symbol followed by any number of struct options; each slot is either a slot symbol or a list of the form `(slot-name default-value slot-options...)'. The default-value is a Lisp form which is evaluated any time an instance of the structure type is created without specifying that slot's value.

Common Lisp defines several slot options, but the only one implemented in this package is `:read-only`. A non-`nil` value for this option means the slot should not be `setf`-able; the slot's value is determined when the object is created and does not change afterward.

```
(defstruct person
  (name nil :read-only t)
  age
  (sex 'unknown))
```

Any slot options other than `:read-only` are ignored.

For obscure historical reasons, structure options take a different form than slot options. A structure option is either a keyword symbol, or a list beginning with a keyword symbol possibly followed by arguments. (By contrast, slot options are key-value pairs not enclosed in lists.)

```
(defstruct (person (:constructor create-person)
                   (:type list)
                   :named)
  name age sex)
```

The following structure options are recognized.

@itemmax=0 in @advance@leftskip-.5@tableindent

`:conc-name`

The argument is a symbol whose print name is used as the prefix for the names of slot accessor functions. The default is the name of the struct type followed by a hyphen. The option `(:conc-name p-)` would change this prefix to `p-`. Specifying `nil` as an argument means no prefix, so that the slot names themselves are used to name the accessor functions.

`:constructor`

In the simple case, this option takes one argument which is an alternate name to use for the constructor function. The default is `make-name`, e.g., `make-person`. The above example changes this to `create-person`. Specifying `nil` as an argument means that no standard constructor should be generated at all.

In the full form of this option, the constructor name is followed by an arbitrary argument list. See section Program Structure, for a description of the format of Common Lisp argument lists. All options, such as `&rest` and `&key`, are supported. The argument names should match the slot names; each slot is initialized from the corresponding argument. Slots whose names do not appear in the argument list are initialized based on the default-value in their slot descriptor. Also, `&optional` and `&key` arguments which don't specify defaults take their defaults from the slot descriptor. It is legal to include arguments which don't correspond to slot names; these are useful if they are referred to in the defaults for optional, keyword, or `&aux` arguments which *do* correspond to slots.

You can specify any number of full-format `:constructor` options on a structure. The default constructor is still generated as well unless you disable it with a simple-format `:constructor` option.

```
(defstruct
 (person
  (:constructor nil)   ; no default constructor
  (:constructor new-person (name sex &optional (age 0)))
  (:constructor new-hound (&key (name "Rover")
                                (dog-years 0)
                          &aux (age (* 7 dog-years))
                                (sex 'canine))))
 name age sex)
```

The first constructor here takes its arguments positionally rather than by keyword. (In official Common Lisp terminology, constructors that work By Order of Arguments instead of by keyword are called "BOA constructors." No, I'm not making this up.) For example, `(new-person "Jane" 'female)` generates a person whose slots are `"Jane"`, 0, and `female`, respectively.

The second constructor takes two keyword arguments, `:name`, which initializes the `name` slot and defaults to `"Rover"`, and `:dog-years`, which does not itself correspond to a slot but which is used to initialize the `age` slot. The `sex` slot is forced to the symbol `canine` with no syntax for overriding it.

`:copier`

The argument is an alternate name for the copier function for this type. The default is `copy-name`. `nil` means not to generate a copier function. (In this implementation, all copier functions are simply synonyms for `copy-sequence`.)

`:predicate`

The argument is an alternate name for the predicate which recognizes objects of this type. The default is `name-p`. `nil` means not to generate a predicate function. (If the `:type` option is used without the `:named` option, no predicate is ever generated.)

In true Common Lisp, `typep` is always able to recognize a structure object even if `:predicate` was used. In this package, `typep` simply looks for a function called `typename-p`, so it will work for

structure types only if they used the default predicate name.

`:include`

This option implements a very limited form of C++-style inheritance. The argument is the name of another structure type previously created with `defstruct`. The effect is to cause the new structure type to inherit all of the included structure's slots (plus, of course, any new slots described by this struct's slot descriptors). The new structure is considered a "specialization" of the included one. In fact, the predicate and slot accessors for the included type will also accept objects of the new type.

If there are extra arguments to the `:include` option after the included-structure name, these options are treated as replacement slot descriptors for slots in the included structure, possibly with modified default values. Borrowing an example from Steele:

```
(defstruct person name (age 0) sex)
     => person
(defstruct (astronaut (:include person (age 45)))
  helmet-size
  (favorite-beverage 'tang))
     => astronaut

(setq joe (make-person :name "Joe"))
     => [cl-struct-person "Joe" 0 nil]
(setq buzz (make-astronaut :name "Buzz"))
     => [cl-struct-astronaut "Buzz" 45 nil nil tang]

(list (person-p joe) (person-p buzz))
     => (t t)
(list (astronaut-p joe) (astronaut-p buzz))
     => (nil t)

(person-name buzz)
     => "Buzz"
(astronaut-name joe)
     => error: "astronaut-name accessing a non-astronaut"
```

Thus, if `astronaut` is a specialization of `person`, then every `astronaut` is also a `person` (but not the other way around). Every `astronaut` includes all the slots of a `person`, plus extra slots that are specific to astronauts. Operations that work on people (like `person-name`) work on astronauts just like other people.

`:print-function`

In full Common Lisp, this option allows you to specify a function which is called to print an instance of the structure type. The Emacs Lisp system offers no hooks into the Lisp printer which would allow for such a feature, so this package simply ignores `:print-function`.

`:type`

The argument should be one of the symbols `vector` or `list`. This tells which underlying Lisp data type should be used to implement the new structure type. Vectors are used by default, but `(:type list)` will cause structure objects to be stored as lists instead.

The vector representation for structure objects has the advantage that all structure slots can be acc quickly, although creating vectors is a bit slower in Emacs Lisp. Lists are easier to create, but take relatively long time accessing the later slots.

`:named`

This option, which takes no arguments, causes a characteristic "tag" symbol to be stored at the front of the structure object. Using `:type` without also using `:named` will result in a structure type stored as plain vectors or lists with no identifying features.

The default, if you don't specify `:type` explicitly, is to use named vectors. Therefore, `:named` is only useful in conjunction with `:type`.

```
(defstruct (person1) name age sex)
(defstruct (person2 (:type list) :named) name age sex)
(defstruct (person3 (:type list)) name age sex)

(setq p1 (make-person1))
     => [cl-struct-person1 nil nil nil]
(setq p2 (make-person2))
     => (person2 nil nil nil)
(setq p3 (make-person3))
     => (nil nil nil)

(person1-p p1)
     => t
(person2-p p2)
     => t
(person3-p p3)
     => error: function person3-p undefined
```

Since unnamed structures don't have tags, `defstruct` is not able to make a useful predicate for recognizing them. Also, accessors like `person3-name` will be generated but they will not be able to do any type checking. The `person3-name` function, for example, will simply be a synonym for `car` in this case. By contrast, `person2-name` is able to verify that its argument is indeed a `person2` object before proceeding.

`:initial-offset`

The argument must be a nonnegative integer. It specifies a number of slots to be left "empty" at the front of the structure. If the structure is named, the tag appears at the specified position in the list or vector; otherwise, the first slot appears at that position. Earlier positions are filled with `nil` by the constructors and ignored otherwise. If the type `:includes` another type, then `:initial-offset` specifies a number of slots to be skipped between the last slot of the included type and the first new slot.

Except as noted, the `defstruct` facility of this package is entirely compatible with that of Common Lisp.

@chapno=23

Assertions and Errors

This section describes two macros that test assertions, i.e., conditions which must be true if the program is operating correctly. Assertions never add to the behavior of a Lisp program; they simply make "sanity checks" to make sure everything is as it should be.

If the optimization property `speed` has been set to 3, and `safety` is less than 3, then the byte-compiler will optimize away the following assertions. Because assertions might be optimized away, it is a bad idea for them to include side-effects.

Special Form: **assert** *test-form [show-args string args...]*

This form verifies that test-form is true (i.e., evaluates to a non-`nil` value). If so, it returns `nil`. If the test is not satisfied, `assert` signals an error.

A default error message will be supplied which includes test-form. You can specify a different error message by including a string argument plus optional extra arguments. Those arguments are simply passed to `error` to signal the error.

If the optional second argument show-args is `t` instead of `nil`, then the error message (with or without string) will also include all non-constant arguments of the top-level form. For example:

```
(assert (> x 10) t "x is too small: %d")
```

This usage of show-args is an extension to Common Lisp. In true Common Lisp, the second argument gives a list of places which can be `setf`'d by the user before continuing from the error. Since Emacs Lisp does not support continuable errors, it makes no sense to specify places.

Special Form: **check-type** *form type [string]*

This form verifies that form evaluates to a value of type type. If so, it returns `nil`. If not, `check-type` signals a `wrong-type-argument` error. The default error message lists the erroneous value along with type and form themselves. If string is specified, it is included in the error message in place of type. For example:

```
(check-type x (integer 1 *) "a positive integer")
```

See section Type Predicates, for a description of the type specifiers that may be used for type.

Note that in Common Lisp, the first argument to `check-type` must be a place suitable for use by `setf`, because `check-type` signals a continuable error that allows the user to modify place.

The following error-related macro is also defined:

Special Form: **ignore-errors** *forms...*

This executes forms exactly like a `progn`, except that errors are ignored during the forms. More precisely, if an error is signalled then `ignore-errors` immediately aborts execution of the forms and returns `nil`. If the forms complete successfully, `ignore-errors` returns the result of the last form.

Efficiency Concerns

Macros

Many of the advanced features of this package, such as `defun*`, `loop`, and `setf`, are implemented as Lisp macros. In byte-compiled code, these complex notations will be expanded into equivalent Lisp code which is simple and efficient. For example, the forms

```
(incf i n)
(push x (car p))
```

are expanded at compile-time to the Lisp forms

```
(setq i (+ i n))
(setcar p (cons x (car p)))
```

which are the most efficient ways of doing these respective operations in Lisp. Thus, there is no performance penalty for using the more readable `incf` and `push` forms in your compiled code.

Interpreted code, on the other hand, must expand these macros every time they are executed. For this reason it is strongly recommended that code making heavy use of macros be compiled. (The features labelled "Special Form" instead of "Function" in this manual are macros.) A loop using `incf` a hundred times will execute considerably faster if compiled, and will also garbage-collect less because the macro expansion will not have to be generated, used, and thrown away a hundred times.

You can find out how a macro expands by using the `cl-prettyexpand` function.

Function: **cl-prettyexpand** *form &optional full*

This function takes a single Lisp form as an argument and inserts a nicely formatted copy of it in the current buffer (which must be in Lisp mode so that indentation works properly). It also expands all Lisp macros which appear in the form. The easiest way to use this function is to go to the `*scratch*` buffer and type, say,

```
(cl-prettyexpand '(loop for x below 10 collect x))
```

and type C-x C-e immediately after the closing parenthesis; the expansion

```
(block nil
  (let* ((x 0)
         (G1004 nil))
    (while (< x 10)
      (setq G1004 (cons x G1004))
      (setq x (+ x 1)))
    (nreverse G1004)))
```

will be inserted into the buffer. (The `block` macro is expanded differently in the interpreter and compiler, so

`cl-prettyexpand` just leaves it alone. The temporary variable `G1004` was created by `gensym`.)

If the optional argument full is true, then *all* macros are expanded, including `block`, `eval-when`, and compiler macros. Expansion is done as if form were a top-level form in a file being compiled. For example,

```
(cl-prettyexpand '(pushnew 'x list))
     -| (setq list (adjoin 'x list))
(cl-prettyexpand '(pushnew 'x list) t)
     -| (setq list (if (memq 'x list) list (cons 'x list)))
(cl-prettyexpand '(caddr (member* 'a list)) t)
     -| (car (cdr (cdr (memq 'a list))))
```

Note that `adjoin`, `caddr`, and `member*` all have built-in compiler macros to optimize them in common cases.

Error Checking

Common Lisp compliance has in general not been sacrificed for the sake of efficiency. A few exceptions have been made for cases where substantial gains were possible at the expense of marginal incompatibility. One example is the use of `memq` (which is treated very efficiently by the byte-compiler) to scan for keyword arguments; this can become confused in rare cases when keyword symbols are used as both keywords and data values at once. This is extremely unlikely to occur in practical code, and the use of `memq` allows functions with keyword arguments to be nearly as fast as functions that use `&optional` arguments.

The Common Lisp standard (as embodied in Steele's book) uses the phrase "it is an error if" to indicate a situation which is not supposed to arise in complying programs; implementations are strongly encouraged but not required to signal an error in these situations. This package sometimes omits such error checking in the interest of compactness and efficiency. For example, `do` variable specifiers are supposed to be lists of one, two, or three forms; extra forms are ignored by this package rather than signalling a syntax error. The `endp` function is simply a synonym for `null` in this package. Functions taking keyword arguments will accept an odd number of arguments, treating the trailing keyword as if it were followed by the value `nil`.

Argument lists (as processed by `defun*` and friends) *are* checked rigorously except for the minor point just mentioned; in particular, keyword arguments are checked for validity, and `&allow-other-keys` and `:allow-other-keys` are fully implemented. Keyword validity checking is slightly time consuming (though not too bad in byte-compiled code); you can use `&allow-other-keys` to omit this check. Functions defined in this package such as `find` and `member*` do check their keyword arguments for validity.

Optimizing Compiler

The byte-compiler that comes with Emacs 18 normally fails to expand macros that appear in top-level positions in the file (i.e., outside of `defun`s or other enclosing forms). This would have disastrous consequences to programs that used such top-level macros as `defun*`, `eval-when`, and `defstruct`. To work around this problem, the CL package patches the Emacs 18 compiler to expand top-level macros. This patch will apply to your own macros, too, if they are used in a top-level context. The patch will not harm versions of the Emacs 18 compiler which have already had a similar patch applied, nor will it affect the

optimizing Emacs 19 byte-compiler written by Jamie Zawinski and Hallvard Furuseth. The patch is appl to the byte compiler's code in Emacs' memory, *not* to the `bytecomp.elc' file stored on disk.

The Emacs 19 compiler (for Emacs 18) is available from various Emacs Lisp archive sites such as `archive.cis.ohio-state.edu`. Its use is highly recommended; many of the Common Lisp macros emit code which can be improved by optimization. In particular, `block`s (whether explicit or implicit in constructs like `defun*` and `loop`) carry a fair run-time penalty; the optimizing compiler removes `block`s which are not actually referenced by `return` or `return-from` inside the block.

Common Lisp Compatibility

Following is a list of all known incompatibilities between this package and Common Lisp as documented in Steele (2nd edition).

Certain function names, such as `member`, `assoc`, and `floor`, were already taken by (incompatible) Emacs Lisp functions; this package appends `*' to the names of its Common Lisp versions of these functions.

The word `defun*` is required instead of `defun` in order to use extended Common Lisp argument lists in a function. Likewise, `defmacro*` and `function*` are versions of those forms which understand full-featured argument lists. The `&whole` keyword does not work in `defmacro` argument lists (except inside recursive argument lists).

In order to allow an efficient implementation, keyword arguments use a slightly cheesy parser which may be confused if a keyword symbol is passed as the *value* of another keyword argument. (Specifically, `(memq :keyword rest-of-arguments)` is used to scan for `:keyword` among the supplied keyword arguments.)

The `eql` and `equal` predicates do not distinguish between IEEE floating-point plus and minus zero. The `equalp` predicate has several differences with Common Lisp; see section Predicates.

The `setf` mechanism is entirely compatible, except that setf-methods return a list of five values rather than five values directly. Also, the new "`setf` function" concept (typified by `(defun (setf foo) ...)`) is not implemented.

The `do-all-symbols` form is the same as `do-symbols` with no obarray argument. In Common Lisp, this form would iterate over all symbols in all packages. Since Emacs obarrays are not a first-class package mechanism, there is no way for `do-all-symbols` to locate any but the default obarray.

The `loop` macro is complete except that `loop-finish` and type specifiers are unimplemented.

The multiple-value return facility treats lists as multiple values, since Emacs Lisp cannot support multiple return values directly. The macros will be compatible with Common Lisp if `values` or `values-list` is always used to return to a `multiple-value-bind` or other multiple-value receiver; if `values` is used without `multiple-value-...` or vice-versa the effect will be different from Common Lisp.

Many Common Lisp declarations are ignored, and others match the Common Lisp standard in concept but not in detail. For example, local `special` declarations, which are purely advisory in Emacs Lisp, do not rigorously obey the scoping rules set down in Steele's book.

The variable `*gensym-counter*` starts out with a pseudo-random value rather than with zero. This is to

cope with the fact that generated symbols become interned when they are written to and loaded back fro file.

The `defstruct` facility is compatible, except that structures are of type `:type vector :named` by default rather than some special, distinct type. Also, the `:type` slot option is ignored.

The second argument of `check-type` is treated differently.

Old CL Compatibility

Following is a list of all known incompatibilities between this package and the older Quiroz `` `cl.el' `` package.

This package's emulation of multiple return values in functions is incompatible with that of the older package. That package attempted to come as close as possible to true Common Lisp multiple return values; unfortunately, it could not be 100% reliable and so was prone to occasional surprises if used freely. This package uses a simpler method, namely replacing multiple values with lists of values, which is more predictable though more noticeably different from Common Lisp.

The `defkeyword` form and `keywordp` function are not implemented in this package.

The `member`, `floor`, `ceiling`, `truncate`, `round`, `mod`, and `rem` functions are suffixed by `*' in this package to avoid collision with existing functions in Emacs 18 or Emacs 19. The older package simply redefined these functions, overwriting the built-in meanings and causing serious portability problems with Emacs 19. (Some more recent versions of the Quiroz package changed the names to `cl-member`, etc.; this package defines the latter names as aliases for `member*`, etc.)

Certain functions in the old package which were buggy or inconsistent with the Common Lisp standard are incompatible with the conforming versions in this package. For example, `eql` and `member` were synonyms for `eq` and `memq` in that package, `setf` failed to preserve correct order of evaluation of its arguments, etc.

Finally, unlike the older package, this package is careful to prefix all of its internal names with `cl-`. Except for a few functions which are explicitly defined as additional features (such as `floatp-safe` and `letf`), this package does not export any non-`cl-' symbols which are not also part of Common Lisp.

The `cl-compat` package

The CL package includes emulations of some features of the old `` `cl.el' ``, in the form of a compatibility package `cl-compat`. To use it, put `(require 'cl-compat)` in your program.

The old package defined a number of internal routines without `cl-` prefixes or other annotations. Call to these routines may have crept into existing Lisp code. `cl-compat` provides emulations of the following internal routines: `pair-with-newsyms`, `zip-lists`, `unzip-lists`, `reassemble-arglists`, `duplicate-symbols-p`, `safe-idiv`.

Some `setf` forms translated into calls to internal functions that user code might call directly. The functions `setnth`, `setnthcdr`, and `setelt` fall in this category; they are defined by `cl-compat`, but the best fix is to change to use `setf` properly.

The `cl-compat` file defines the keyword functions `keywordp`, `keyword-of`, and `defkeyword`, which are not defined by the new CL package because the use of keywords as data is discouraged.

The `build-klist` mechanism for parsing keyword arguments is emulated by `cl-compat`; the `with-keyword-args` macro is not, however, and in any case it's best to change to use the more natural keyword argument processing offered by `defun*`.

Multiple return values are treated differently by the two Common Lisp packages. The old package's method was more compatible with true Common Lisp, though it used heuristics that caused it to report spurious multiple return values in certain cases. The `cl-compat` package defines a set of multiple-value macros that are compatible with the old CL package; again, they are heuristic in nature, but they are guaranteed to work in any case where the old package's macros worked. To avoid name collision with the "official" multiple-value facilities, the ones in `cl-compat` have capitalized names: `Values`, `Values-list`, `Multiple-value-bind`, etc.

The functions `cl-floor`, `cl-ceiling`, `cl-truncate`, and `cl-round` are defined by `cl-compat` to use the old-style multiple-value mechanism, just as they did in the old package. The newer `floor*` and friends return their two results in a list rather than as multiple values. Note that older versions of the old package used the unadorned names `floor`, `ceiling`, etc.; `cl-compat` cannot use these names because they conflict with Emacs 19 built-ins.

Porting Common Lisp

This package is meant to be used as an extension to Emacs Lisp, not as an Emacs implementation of true Common Lisp. Some of the remaining differences between Emacs Lisp and Common Lisp make it difficult to port large Common Lisp applications to Emacs. For one, some of the features in this package are not fully compliant with ANSI or Steele; see section Common Lisp Compatibility. But there are also quite a few features that this package does not provide at all. Here are some major omissions that you will want watch out for when bringing Common Lisp code into Emacs.

- Case-insensitivity. Symbols in Common Lisp are case-insensitive by default. Some programs refer to a function or variable as `foo` in one place and `Foo` or `FOO` in another. Emacs Lisp will treat these as three distinct symbols.

  Some Common Lisp code is written in all upper-case. While Emacs is happy to let the program's own functions and variables use this convention, calls to Lisp builtins like `if` and `defun` will have to be changed to lower-case.

- Lexical scoping. In Common Lisp, function arguments and `let` bindings apply only to references physically within their bodies (or within macro expansions in their bodies). Emacs Lisp, by contrast, uses dynamic scoping wherein a binding to a variable is visible even inside functions called from the body.

  Variables in Common Lisp can be made dynamically scoped by declaring them `special` or using `defvar`. In Emacs Lisp it is as if all variables were declared `special`.

  Often you can use code that was written for lexical scoping even in a dynamically scoped Lisp, but not always. Here is an example of a Common Lisp code fragment that would fail in Emacs Lisp:


```
(defun map-odd-elements (func list)
  (loop for x in list
        for flag = t then (not flag)
        collect (if flag x (funcall func x))))

(defun add-odd-elements (list x)
  (map-odd-elements (function (lambda (a) (+ a x))) list))
```

In Common Lisp, the two functions' usages of `x` are completely independent. In Emacs Lisp, the binding to `x` made by `add-odd-elements` will have been hidden by the binding in `map-odd-elements` by the time the `(+ a x)` function is called.

(This package avoids such problems in its own mapping functions by using names like `cl-x` instead of `x` internally; as long as you don't use the `cl-` prefix for your own variables no collision can occur.)

See section Lexical Bindings, for a description of the `lexical-let` form which establishes a Common Lisp-style lexical binding, and some examples of how it differs from Emacs' regular `let`.

- Common Lisp allows the shorthand `#'x` to stand for `(function x)`, just as `'x` stands for `(quote x)`. In Common Lisp, one traditionally uses `#'` notation when referring to the name of a function. In Emacs Lisp, it works just as well to use a regular quote:

```
(loop for x in y by #'cddr collect (mapcar #'plusp x))  ; Common Lisp
(loop for x in y by 'cddr collect (mapcar 'plusp x))    ; Emacs Lisp
```

When `#'` introduces a `lambda` form, it is best to write out `(function ...)` longhand in Emacs Lisp. You can use a regular quote, but then the byte-compiler won't know that the `lambda` expression is code that can be compiled.

```
(mapcar #'(lambda (x) (* x 2)) list)            ; Common Lisp
(mapcar (function (lambda (x) (* x 2))) list)   ; Emacs Lisp
```

Lucid Emacs supports `#'` notation starting with version 19.8.

- The "backquote" feature uses a different syntax in Emacs Lisp.

```
(defmacro foo (v &rest body) `(let ((,v 0)) @,body))  ; Common Lisp
(defmacro foo (v &rest body) (` (let (((, v) 0)) (@, body)))  ; Emacs
```

- Reader macros. Common Lisp includes a second type of macro that works at the level of individual characters. For example, Common Lisp implements the quote notation by a reader macro called `'`, whereas Emacs Lisp's parser just treats quote as a special case. Some Lisp packages use reader macros to create special syntaxes for themselves, which the Emacs parser is incapable of reading.

  The lack of reader macros, incidentally, is the reason behind Emacs Lisp's unusual backquote syntax. Since backquotes are implemented as a Lisp package and not built-in to the Emacs parser, they are forced to use a regular macro named ` which is used with the standard function/macro call notation.

- Other syntactic features. Common Lisp provides a number of notations beginning with # that the Emacs Lisp parser won't understand. For example, `#| ... |#' is an alternate comment notation, and `#+lucid (foo)' tells the parser to ignore the `(foo)` except in Lucid Common Lisp.

- Packages. In Common Lisp, symbols are divided into packages. Symbols that are Lisp built-ins are typically stored in one package; symbols that are vendor extensions are put in another, and each application program would have a package for its own symbols. Certain symbols are "exported" by a package and others are internal; certain packages "use" or import the exported symbols of other packages. To access symbols that would not normally be visible due to this importing and exporting, Common Lisp provides a syntax like `package:symbol` or `package::symbol`.

  Emacs Lisp has a single namespace for all interned symbols, and then uses a naming convention of putting a prefix like `cl-` in front of the name. Some Emacs packages adopt the Common Lisp-like convention of using `cl:` or `cl::` as the prefix. However, the Emacs parser does not understand colons and just treats them as part of the symbol name. Thus, while `mapcar` and `lisp:mapcar` may refer to the same symbol in Common Lisp, they are totally distinct in Emacs Lisp. Common Lisp programs which refer to a symbol by the full name sometimes and the short name other times will not port cleanly to Emacs.

  Emacs Lisp does have a concept of "obarrays," which are package-like collections of symbols, but this feature is not strong enough to be used as a true package mechanism.

- Keywords. The notation `:test-not` in Common Lisp really is a shorthand for `keyword:test-not`; keywords are just symbols in a built-in `keyword` package with the special property that all its symbols are automatically self-evaluating. Common Lisp programs often use keywords liberally to avoid having to use quotes.

  In Emacs Lisp a keyword is just a symbol whose name begins with a colon; since the Emacs parser does not treat them specially, they have to be explicitly made self-evaluating by a statement like `(setq :test-not ':test-not)`. This package arranges to execute such a statement whenever `defun*` or some other form sees a keyword being used as an argument. Common Lisp code that assumes that a symbol `:mumble` will be self-evaluating even though it was never introduced by a `defun*` will have to be fixed.

- The `format` function is quite different between Common Lisp and Emacs Lisp. It takes an additional "destination" argument before the format string. A destination of `nil` means to format to a string as in Emacs Lisp; a destination of `t` means to write to the terminal (similar to `message` in Emacs). Also, format control strings are utterly different; `~` is used instead of `%` to introduce format codes, and the set of available codes is much richer. There are no notations like `\n` for string literals; instead, `format` is used with the "newline" format code, `~%`. More advanced formatting codes provide such features as paragraph filling, case conversion, and even loops and conditionals.

  While it would have been possible to implement most of Common Lisp `format` in this package (under the name `format*`, of course), it was not deemed worthwhile. It would have required a huge amount of code to implement even a decent subset of `format*`, yet the functionality it would provide over Emacs Lisp's `format` would rarely be useful.

- Vector constants use square brackets in Emacs Lisp, but `#(a b c)` notation in Common Lisp. To further complicate matters, Emacs 19 introduces its own `#(` notation for something entirely different--strings with properties.

- Characters are distinct from integers in Common Lisp. The notation for character constants is also different: `#\A` instead of `?A`. Also, `string=` and `string-equal` are synonyms in Emacs Lisp whereas the latter is case-insensitive in Common Lisp.

- Data types. Some Common Lisp data types do not exist in Emacs Lisp. Rational numbers and complex

numbers are not present, nor are large integers (all integers are "fixnums"). All arrays are one-dimensional. There are no readtables or pathnames; streams are a set of existing data types rat than a new data type of their own. Hash tables, random-states, structures, and packages (obarrays) ar built from Lisp vectors or lists rather than being distinct types.

- The Common Lisp Object System (CLOS) is not implemented, nor is the Common Lisp Condition System.
- Common Lisp features that are completely redundant with Emacs Lisp features of a different name generally have not been implemented. For example, Common Lisp writes `defconstant` where Emacs Lisp uses `defconst`. Similarly, `make-list` takes its arguments in different ways in the two Lisps but does exactly the same thing, so this package has not bothered to implement a Common Lisp-style `make-list`.
- A few more notable Common Lisp features not included in this package: `compiler-let`, `tagbody`, `prog`, `ldb/dpb`, `parse-integer`, `cerror`.
- Recursion. While recursion works in Emacs Lisp just like it does in Common Lisp, various details of the Emacs Lisp system and compiler make recursion much less efficient than it is in most Lisps. Some schools of thought prefer to use recursion in Lisp over other techniques; they would sum a list of numbers using something like

```
(defun sum-list (list)
  (if list
      (+ (car list) (sum-list (cdr list)))
    0))
```

where a more iteratively-minded programmer might write one of these forms:

```
(let ((total 0)) (dolist (x my-list) (incf total x)) total)
(loop for x in my-list sum x)
```

While this would be mainly a stylistic choice in most Common Lisps, in Emacs Lisp you should be aware that the iterative forms are much faster than recursion. Also, Lisp programmers will want to note that the current Emacs Lisp compiler does not optimize tail recursion.

Function Index

a

b

c

d

e

f

g

h

i

l

m

n

o

p

r

s

t

u

w

Variable Index

*

f

l

m

Go to the next section.

Overview

Common Lisp is a huge language, and Common Lisp systems tend to be massive and extremely complex. Emacs Lisp, by contrast, is rather minimalist in the choice of Lisp features it offers the programmer. As Emacs Lisp programmers have grown in number, and the applications they write have grown more ambitious, it has become clear that Emacs Lisp could benefit from many of the conveniences of Common Lisp.

The CL package adds a number of Common Lisp functions and control structures to Emacs Lisp. While not a 100% complete implementation of Common Lisp, CL adds enough functionality to make Emacs Lisp programming significantly more convenient.

Some Common Lisp features have been omitted from this package for various reasons:

- Some features are too complex or bulky relative to their benefit to Emacs Lisp programmers. CLOS and Common Lisp streams are fine examples of this group.
- Other features cannot be implemented without modification to the Emacs Lisp interpreter itself, such as multiple return values, lexical scoping, case-insensitive symbols, and complex numbers. The CL package generally makes no attempt to emulate these features.
- Some features conflict with existing things in Emacs Lisp. For example, Emacs' `assoc` function is incompatible with the Common Lisp `assoc`. In such cases, this package usually adds the suffix `*' to the function name of the Common Lisp version of the function (e.g., `assoc*`).

The package described here was written by Dave Gillespie, `daveg@synaptics.com'. It is a total rewrite of the original 1986 `cl.el' package by Cesar Quiroz. Most features of the the Quiroz package have been retained; any incompatibilities are noted in the descriptions below. Care has been taken in this version to ensure that each function is defined efficiently, concisely, and with minimal impact on the rest of the Emacs environment.

Usage

Lisp code that uses features from the CL package should include at the beginning:

```
(require 'cl)
```

If you want to ensure that the new (Gillespie) version of CL is the one that is present, add an additional `(require 'cl-19)` call:

```
(require 'cl)
(require 'cl-19)
```

The second call will fail (with "`cl-19.el' not found") if the old `cl.el' package was in use.

It is safe to arrange to load CL at all times, e.g., in your `.emacs' file. But it's a good idea, for

portability, to `(require 'cl)` in your code even if you do this.

Organization

The Common Lisp package is organized into four files:

`cl.el'
: This is the "main" file, which contains basic functions and information about the package. This file is relatively compact--about 700 lines.

`cl-extra.el'
: This file contains the larger, more complex or unusual functions. It is kept separate so that packages which only want to use Common Lisp fundamentals like the `cadr` function won't need to pay the overhead of loading the more advanced functions.

`cl-seq.el'
: This file contains most of the advanced functions for operating on sequences or lists, such as `delete-if` and `assoc*`.

`cl-macs.el'
: This file contains the features of the packages which are macros instead of functions. Macros expand when the caller is compiled, not when it is run, so the macros generally only need to be present when the byte-compiler is running (or when the macros are used in uncompiled code such as a `.emacs' file). Most of the macros of this package are isolated in `cl-macs.el' so that they won't take up memory unless you are compiling.

The file `cl.el' includes all necessary `autoload` commands for the functions and macros in the other three files. All you have to do is `(require 'cl)`, and `cl.el' will take care of pulling in the other files when they are needed.

There is another file, `cl-compat.el', which defines some routines from the older `cl.el' package that are no longer present in the new package. This includes internal routines like `setelt` and `zip-lists`, deprecated features like `defkeyword`, and an emulation of the old-style multiple-values feature. See section Old CL Compatibility.

Installation

Installation of the CL package is simple: Just put the byte-compiled files `cl.elc', `cl-extra.elc', `cl-seq.elc', `cl-macs.elc', and `cl-compat.elc' into a directory on your `load-path`.

There are no special requirements to compile this package: The files do not have to be loaded before they are compiled, nor do they need to be compiled in any particular order.

You may choose to put the files into your main `lisp/' directory, replacing the original `cl.el' file there. Or, you could put them into a directory that comes before `lisp/' on your `load-path` so that the old `cl.el' is effectively hidden.

Also, format the `cl.texinfo' file and put the resulting Info files in the `info/' directory or another suitable place.

You may instead wish to leave this package's components all in their own directory, and then add this directory to your `load-path' and (Emacs 19 only) `Info-directory-list'. Add the directory to the front of the list so the old CL package and its documentation are hidden.

Naming Conventions

Except where noted, all functions defined by this package have the same names and calling conventions as their Common Lisp counterparts.

Following is a complete list of functions whose names were changed from Common Lisp, usually to avoid conflicts with Emacs. In each case, a `*' has been appended to the Common Lisp name to obtain the Emacs name:

```
defun*          defsubst*       defmacro*       function*
member*         assoc*          rassoc*         get*
remove*         delete*         mapcar*         sort*
floor*          ceiling*        truncate*       round*
mod*            rem*            random*
```

Internal function and variable names in the package are prefixed by `cl-'. Here is a complete list of functions *not* prefixed by `cl-' which were not taken from Common Lisp:

```
member          delete          remove          remq
rassoc          floatp-safe     lexical-let     lexical-let*
callf           callf2          letf            letf*
defsubst*       defalias        add-hook        eval-when-compile
```

(Most of these are Emacs 19 features provided to Emacs 18 users, or introduced, like `remq', for reasons of symmetry with similar features.)

The following simple functions and macros are defined in `cl.el'; they do not cause other components like `cl-extra' to be loaded.

```
eql             floatp-safe     abs             endp
evenp           oddp            plusp           minusp
last            butlast         nbutlast        caar .. cddddr
list*           ldiff           rest            first .. tenth
member [1]      copy-list       subst           mapcar* [2]
adjoin [3]      acons           pairlis         when
unless          pop [4]         push [4]        pushnew [3,4]
incf [4]        decf [4]        proclaim        declaim
add-hook
```

[1] This is the Emacs 19-compatible function, not `member*`.

[2] Only for one sequence argument or two list arguments.

[3] Only if `:test` is `eq`, `equal`, or unspecified, and `:key` is not used.

[4] Only when place is a plain variable name.

@chapno=4

Program Structure

This section describes features of the CL package which have to do with programs as a whole: advanced argument lists for functions, and the `eval-when` construct.

@secno=1

Argument Lists

Emacs Lisp's notation for argument lists of functions is a subset of the Common Lisp notation. As well as the familiar `&optional` and `&rest` markers, Common Lisp allows you to specify default values for optional arguments, and it provides the additional markers `&key` and `&aux`.

Since argument parsing is built-in to Emacs, there is no way for this package to implement Common Lisp argument lists seamlessly. Instead, this package defines alternates for several Lisp forms which you must use if you need Common Lisp argument lists.

Special Form: **defun*** *name arglist body...*

This form is identical to the regular `defun` form, except that arglist is allowed to be a full Common Lisp argument list. Also, the function body is enclosed in an implicit block called name; see section Blocks and Exits.

Special Form: **defsubst*** *name arglist body...*

This is just like `defun*`, except that the function that is defined is automatically proclaimed `inline`, i.e., calls to it may be expanded into in-line code by the byte compiler. This is analogous to the `defsubst` form in Emacs 19; `defsubst*` uses a different method (compiler macros) which works in all version of Emacs, and also generates somewhat more efficient inline expansions. In particular, `defsubst*` arranges for the processing of keyword arguments, default values, etc., to be done at compile-time whenever possible.

Special Form: **defmacro*** *name arglist body...*

This is identical to the regular `defmacro` form, except that arglist is allowed to be a full Common Lisp argument list. The `&environment` keyword is supported as described in Steele. The `&whole` keyword is supported only within destructured lists (see below); top-level `&whole` cannot be implemented with the current Emacs Lisp interpreter. The macro expander body is enclosed in an implicit block called

name.

Special Form: **function*** *symbol-or-lambda*

This is identical to the regular `function` form, except that if the argument is a `lambda` form then that form may use a full Common Lisp argument list.

Also, all forms (such as `defsetf` and `flet`) defined in this package that include arglists in their syntax allow full Common Lisp argument lists.

Note that it is *not* necessary to use `defun*` in order to have access to most CL features in your function. These features are always present; `defun*`'s only difference from `defun` is its more flexible argument lists and its implicit block.

The full form of a Common Lisp argument list is

```
(var...
 &optional (var initform svar)...
 &rest var
 &key ((keyword var) initform svar)...
 &aux (var initform)...)
```

Each of the five argument list sections is optional. The svar, initform, and keyword parts are optional; if they are omitted, then `(var)' may be written simply `var'.

The first section consists of zero or more required arguments. These arguments must always be specified in a call to the function; there is no difference between Emacs Lisp and Common Lisp as far as required arguments are concerned.

The second section consists of optional arguments. These arguments may be specified in the function call; if they are not, initform specifies the default value used for the argument. (No initform means to use `nil` as the default.) The initform is evaluated with the bindings for the preceding arguments already established; `(a &optional (b (1+ a)))` matches one or two arguments, with the second argument defaulting to one plus the first argument. If the svar is specified, it is an auxiliary variable which is bound to `t` if the optional argument was specified, or to `nil` if the argument was omitted. If you don't use an svar, then there will be no way for your function to tell whether it was called with no argument, or with the default value passed explicitly as an argument.

The third section consists of a single rest argument. If more arguments were passed to the function than are accounted for by the required and optional arguments, those extra arguments are collected into a list and bound to the "rest" argument variable. Common Lisp's `&rest` is equivalent to that of Emacs Lisp. Common Lisp accepts `&body` as a synonym for `&rest` in macro contexts; this package accepts it all the time.

The fourth section consists of keyword arguments. These are optional arguments which are specified by name rather than positionally in the argument list. For example,

```
(defun* foo (a &optional b &key c d (e 17)))
```

defines a function which may be called with one, two, or more arguments. The first two arguments are bound to `a` and `b` in the usual way. The remaining arguments must be pairs of the form `:c`, `:d`, or `:e` followed by the value to be bound to the corresponding argument variable. (Symbols whose names begin with a colon are called keywords, and they are self-quoting in the same way as `nil` and `t`.)

For example, the call `(foo 1 2 :d 3 :c 4)` sets the five arguments to 1, 2, 4, 3, and 17, respectively. If the same keyword appears more than once in the function call, the first occurrence takes precedence over the later ones. Note that it is not possible to specify keyword arguments without specifying the optional argument `b` as well, since `(foo 1 :c 2)` would bind `b` to the keyword `:c`, then signal an error because `2` is not a valid keyword.

If a keyword symbol is explicitly specified in the argument list as shown in the above diagram, then that keyword will be used instead of just the variable name prefixed with a colon. You can specify a keyword symbol which does not begin with a colon at all, but such symbols will not be self-quoting; you will have to quote them explicitly with an apostrophe in the function call.

Ordinarily it is an error to pass an unrecognized keyword to a function, e.g., `(foo 1 2 :c 3 :goober 4)`. You can ask Lisp to ignore unrecognized keywords, either by adding the marker `&allow-other-keys` after the keyword section of the argument list, or by specifying an `:allow-other-keys` argument in the call whose value is non-`nil`. If the function uses both `&rest` and `&key` at the same time, the "rest" argument is bound to the keyword list as it appears in the call. For example:

```
(defun* find-thing (thing &rest rest &key need &allow-other-keys)
  (or (apply 'member* thing thing-list :allow-other-keys t rest)
      (if need (error "Thing not found"))))
```

This function takes a `:need` keyword argument, but also accepts other keyword arguments which are passed on to the `member*` function. `allow-other-keys` is used to keep both `find-thing` and `member*` from complaining about each others' keywords in the arguments.

In Common Lisp, keywords are recognized by the Lisp parser itself and treated as special entities. In Emacs, keywords are just symbols whose names begin with colons, which `defun*` has arranged to set equal to themselves so that they will essentially be self-quoting.

As a (significant) performance optimization, this package implements the scan for keyword arguments by calling `memq` to search for keywords in a "rest" argument. Technically speaking, this is incorrect, since `memq` looks at the odd-numbered values as well as the even-numbered keywords. The net effect is that if you happen to pass a keyword symbol as the *value* of another keyword argument, where that keyword symbol happens to equal the name of a valid keyword argument of the same function, then the keyword parser will become confused. This minor bug can only affect you if you use keyword symbols as general-purpose data in your program; this practice is strongly discouraged in Emacs Lisp.

The fifth section of the argument list consists of auxiliary variables. These are not really arguments at all, but simply variables which are bound to `nil` or to the specified initforms during execution of the function. There is no difference between the following two functions, except for a matter of stylistic taste:


```
(defun* foo (a b &aux (c (+ a b)) d)
  body)

(defun* foo (a b)
  (let ((c (+ a b)) d)
    body))
```

Argument lists support destructuring. In Common Lisp, destructuring is only allowed with `defmacro`; this package allows it with `defun*` and other argument lists as well. In destructuring, any argument variable (var in the above diagram) can be replaced by a list of variables, or more generally, a recursive argument list. The corresponding argument value must be a list whose elements match this recursive argument list. For example:

```
(defmacro* dolist ((var listform &optional resultform)
                   &rest body)
  ...)
```

This says that the first argument of `dolist` must be a list of two or three items; if there are other arguments as well as this list, they are stored in `body`. All features allowed in regular argument lists are allowed in these recursive argument lists. In addition, the clause `&whole var' is allowed at the front of a recursive argument list. It binds var to the whole list being matched; thus `(&whole all a b)` matches a list of two things, with `a` bound to the first thing, `b` bound to the second thing, and `all` bound to the list itself. (Common Lisp allows `&whole` in top-level `defmacro` argument lists as well, but Emacs Lisp does not support this usage.)

One last feature of destructuring is that the argument list may be dotted, so that the argument list `(a b . c)` is functionally equivalent to `(a b &rest c)`.

If the optimization quality `safety` is set to 0 (see section Declarations), error checking for wrong number of arguments and invalid keyword arguments is disabled. By default, argument lists are rigorously checked.

Time of Evaluation

Normally, the byte-compiler does not actually execute the forms in a file it compiles. For example, if a file contains `(setq foo t)`, the act of compiling it will not actually set `foo` to `t`. This is true even if the `setq` was a top-level form (i.e., not enclosed in a `defun` or other form). Sometimes, though, you would like to have certain top-level forms evaluated at compile-time. For example, the compiler effectively evaluates `defmacro` forms at compile-time so that later parts of the file can refer to the macros that are defined.

Special Form: **eval-when** *(situations...) forms...*

This form controls when the body forms are evaluated. The situations list may contain any set of the symbols `compile`, `load`, and `eval` (or their long-winded ANSI equivalents, `:compile-toplevel`, `:load-toplevel`, and `:execute`).

The `eval-when` form is handled differently depending on whether or not it is being compiled as a top-level form. Specifically, it gets special treatment if it is being compiled by a command such as `byte-compile-file` which compiles files or buffers of code, and it appears either literally at the top level of the file or inside a top-level `progn`.

For compiled top-level `eval-when`s, the body forms are executed at compile-time if `compile` is in the situations list, and the forms are written out to the file (to be executed at load-time) if `load` is in the situations list.

For non-compiled-top-level forms, only the `eval` situation is relevant. (This includes forms executed by the interpreter, forms compiled with `byte-compile` rather than `byte-compile-file`, and non-top-level forms.) The `eval-when` acts like a `progn` if `eval` is specified, and like `nil` (ignoring the body forms) if not.

The rules become more subtle when `eval-when`s are nested; consult Steele (second edition) for the gruesome details (and some gruesome examples).

Some simple examples:

```
;; Top-level forms in foo.el:
(eval-when (compile)           (setq foo1 'bar))
(eval-when (load)              (setq foo2 'bar))
(eval-when (compile load)      (setq foo3 'bar))
(eval-when (eval)              (setq foo4 'bar))
(eval-when (eval compile)      (setq foo5 'bar))
(eval-when (eval load)         (setq foo6 'bar))
(eval-when (eval compile load) (setq foo7 'bar))
```

When `` `foo.el' `` is compiled, these variables will be set during the compilation itself:

```
foo1  foo3  foo5  foo7      ; `compile'
```

When `` `foo.elc' `` is loaded, these variables will be set:

```
foo2  foo3  foo6  foo7      ; `load'
```

And if `` `foo.el' `` is loaded uncompiled, these variables will be set:

```
foo4  foo5  foo6  foo7      ; `eval'
```

If these seven `eval-when`s had been, say, inside a `defun`, then the first three would have been equivalent to `nil` and the last four would have been equivalent to the corresponding `setq`s.

Note that `(eval-when (load eval) ...)` is equivalent to `(progn ...)` in all contexts. The compiler treats certain top-level forms, like `defmacro` (sort-of) and `require`, as if they were wrapped in `(eval-when (compile load eval) ...)`.

Emacs 19 includes two special forms related to `eval-when`. One of these, `eval-when-compile`, is

not quite equivalent to any `eval-when` construct and is described below. This package defines a version of `eval-when-compile` for the benefit of Emacs 18 users.

The other form, `(eval-and-compile ...)`, is exactly equivalent to `(eval-when (compile load eval) ...)' and so is not itself defined by this package.

Special Form: **eval-when-compile** *forms...*

The forms are evaluated at compile-time; at execution time, this form acts like a quoted constant of the resulting value. Used at top-level, `eval-when-compile` is just like `eval-when (compile eval)'. In other contexts, `eval-when-compile` allows code to be evaluated once at compile-time for efficiency or other reasons.

This form is similar to the `#.' syntax of true Common Lisp.

Special Form: **load-time-value** *form*

The form is evaluated at load-time; at execution time, this form acts like a quoted constant of the resulting value.

Early Common Lisp had a `#,' syntax that was similar to this, but ANSI Common Lisp replaced it with `load-time-value` and gave it more well-defined semantics.

In a compiled file, `load-time-value` arranges for form to be evaluated when the `.elc' file is loaded and then used as if it were a quoted constant. In code compiled by `byte-compile` rather than `byte-compile-file`, the effect is identical to `eval-when-compile`. In uncompiled code, both `eval-when-compile` and `load-time-value` act exactly like `progn`.

```
(defun report ()
  (insert "This function was executed on: "
          (current-time-string)
          ", compiled on: "
          (eval-when-compile (current-time-string))
          ;; or '#.(current-time-string) in real Common Lisp
          ", and loaded on: "
          (load-time-value (current-time-string))))
```

Byte-compiled, the above defun will result in the following code (or its compiled equivalent, of course) in the `.elc' file:

```
(setq --temp-- (current-time-string))
(defun report ()
  (insert "This function was executed on: "
          (current-time-string)
          ", compiled on: "
          '"Wed Jun 23 18:33:43 1993"
          ", and loaded on: "
          --temp--))
```

Function Aliases

This section describes a feature from GNU Emacs 19 which this package makes available in other versions of Emacs.

Function: **defalias** *symbol function*

This function sets symbol's function cell to function. It is equivalent to `fset`, except that in GNU Emacs 19 it also records the setting in `load-history` so that it can be undone by a later `unload-feature`.

In other versions of Emacs, `defalias` is a synonym for `fset`.

Go to the next section.

Go to the previous, next section.

Predicates

This section describes functions for testing whether various facts are true or false.

Type Predicates

The CL package defines a version of the Common Lisp `typep` predicate.

Function: **typep** *object type*

Check if object is of type type, where type is a (quoted) type name of the sort used by Common Lisp. For example, `(typep foo 'integer)` is equivalent to `(integerp foo)`.

The type argument to the above function is either a symbol or a list beginning with a symbol.

- If the type name is a symbol, Emacs appends `-p' to the symbol name to form the name of a predicate function for testing the type. (Built-in predicates whose names end in `p' rather than `-p' are used when appropriate.)
- The type symbol `t` stands for the union of all types. `(typep object t)` is always true. Likewise, the type symbol `nil` stands for nothing at all, and `(typep object nil)` is always false.
- The type symbol `null` represents the symbol `nil`. Thus `(typep object 'null)` is equivalent to `(null object)`.
- The type symbol `real` is a synonym for `number`, and `fixnum` is a synonym for `integer`.
- The type symbols `character` and `string-char` match integers in the range from 0 to 255.
- The type symbol `float` uses the `floatp-safe` predicate defined by this package rather than `floatp`, so it will work correctly even in Emacs versions without floating-point support.
- The type list `(integer low high)` represents all integers between low and high, inclusive. Either bound may be a list of a single integer to specify an exclusive limit, or a `*` to specify no limit. The type `(integer * *)` is thus equivalent to `integer`.
- Likewise, lists beginning with `float`, `real`, or `number` represent numbers of that type falling in a particular range.
- Lists beginning with `and`, `or`, and `not` form combinations of types. For example, `(or integer (float 0 *))` represents all objects that are integers or non-negative floats.
- Lists beginning with `member` or `member*` represent objects `eql` to any of the following values. For example, `(member 1 2 3 4)` is equivalent to `(integer 1 4)`, and `(member nil)` is equivalent to `null`.
- Lists of the form `(satisfies predicate)` represent all objects for which predicate returns true when called with that object as an argument.

The following function and macro (not technically predicates) are related to `typep`.

Function: **coerce** *object type*

This function attempts to convert object to the specified type. If object is already of that type as determined by `typep`, it is simply returned. Otherwise, certain types of conversions will be made: If type is any sequence type (`string`, `list`, etc.) then object will be converted to that type if possible. If type is `character`, then strings of length one and symbols with one-character names can be coerced. If type is `float`, then integers can be coerced in versions of Emacs that support floats. In all other circumstances, `coerce` signals an error.

Special Form: **deftype** *name arglist forms...*

This macro defines a new type called name. It is similar to `defmacro` in many ways; when name is encountered as a type name, the body forms are evaluated and should return a type specifier that is equivalent to the type. The arglist is a Common Lisp argument list of the sort accepted by `defmacro*`. The type specifier `(name args...)' is expanded by calling the expander with those arguments; the type symbol `name' is expanded by calling the expander with no arguments. The arglist is processed the same as for `defmacro*` except that optional arguments without explicit defaults use `*` instead of `nil` as the "default" default. Some examples:

```
(deftype null () '(satisfies null))    ; predefined
(deftype list () '(or null cons))      ; predefined
(deftype unsigned-byte (&optional bits)
  (list 'integer 0 (if (eq bits '*) bits (1- (lsh 1 bits)))))
(unsigned-byte 8)  ==  (integer 0 255)
(unsigned-byte)  ==  (integer 0 *)
unsigned-byte  ==  (integer 0 *)
```

The last example shows how the Common Lisp `unsigned-byte` type specifier could be implemented if desired; this package does not implement `unsigned-byte` by default.

The `typecase` and `check-type` macros also use type names. See section Conditionals. See section Assertions and Errors. The `map`, `concatenate`, and `merge` functions take type-name arguments to specify the type of sequence to return. See section Sequences.

Equality Predicates

This package defines two Common Lisp predicates, `eql` and `equalp`.

Function: **eql** *a b*

This function is almost the same as `eq`, except that if a and b are numbers of the same type, it compares them for numeric equality (as if by `equal` instead of `eq`). This makes a difference only for versions of Emacs that are compiled with floating-point support, such as Emacs 19. Emacs floats are allocated objects just like cons cells, which means that `(eq 3.0 3.0)` will not necessarily be true--if the two `3.0`s were allocated separately, the pointers will be different even though the numbers are the same. But `(eql 3.0 3.0)` will always be true.

The types of the arguments must match, so `(eql 3 3.0)` is still false.

Note that Emacs integers are "direct" rather than allocated, which basically means `(eq 3 3)` will always be true. Thus `eq` and `eql` behave differently only if floating-point numbers are involved, and are indistinguishable on Emacs versions that don't support floats.

There is a slight inconsistency with Common Lisp in the treatment of positive and negative zeros. Some machines, notably those with IEEE standard arithmetic, represent `+0` and `-0` as distinct values. Normally this doesn't matter because the standard specifies that `(= 0.0 -0.0)` should always be true, and this is indeed what Emacs Lisp and Common Lisp do. But the Common Lisp standard states that `(eql 0.0 -0.0)` and `(equal 0.0 -0.0)` should be false on IEEE-like machines; Emacs Lisp does not do this, and in fact the only known way to distinguish between the two zeros in Emacs Lisp is to `format` them and check for a minus sign.

Function: **equalp** *a b*

This function is a more flexible version of `equal`. In particular, it compares strings case-insensitively, and it compares numbers without regard to type (so that `(equalp 3 3.0)` is true). Vectors and conses are compared recursively. All other objects are compared as if by `equal`.

This function differs from Common Lisp `equalp` in several respects. First, Common Lisp's `equalp` also compares *characters* case-insensitively, which would be impractical in this package since Emacs does not distinguish between integers and characters. In keeping with the idea that strings are less vector-like in Emacs Lisp, this package's `equalp` also will not compare strings against vectors of integers. Finally, Common Lisp's `equalp` compares hash tables without regard to ordering, whereas this package simply compares hash tables in terms of their underlying structure (which means vectors for Lucid Emacs 19 hash tables, or lists for other hash tables).

Also note that the Common Lisp functions `member` and `assoc` use `eql` to compare elements, whereas Emacs Lisp follows the MacLisp tradition and uses `equal` for these two functions. In Emacs, use `member*` and `assoc*` to get functions which use `eql` for comparisons.

Go to the previous, next section.

Go to the previous, next section.

Control Structure

The features described in the following sections implement various advanced control structures, including the powerful `setf` facility and a number of looping and conditional constructs.

Assignment

The `psetq` form is just like `setq`, except that multiple assignments are done in parallel rather than sequentially.

Special Form: **psetq** *[symbol form]...*

This special form (actually a macro) is used to assign to several variables simultaneously. Given only one symbol and form, it has the same effect as `setq`. Given several symbol and form pairs, it evaluates all the forms in advance and then stores the corresponding variables afterwards.

```
(setq x 2 y 3)
(setq x (+ x y)  y (* x y))
x
     => 5
y                       ; y was computed after x was set.
     => 15
(setq x 2 y 3)
(psetq x (+ x y)  y (* x y))
x
     => 5
y                       ; y was computed before x was set.
     => 6
```

The simplest use of `psetq` is `(psetq x y y x)`, which exchanges the values of two variables. (The `rotatef` form provides an even more convenient way to swap two variables; see section Modify Macros.)

`psetq` always returns `nil`.

Generalized Variables

A "generalized variable" or "place form" is one of the many places in Lisp memory where values can be stored. The simplest place form is a regular Lisp variable. But the cars and cdrs of lists, elements of arrays, properties of symbols, and many other locations are also places where Lisp values are stored.

The `setf` form is like `setq`, except that it accepts arbitrary place forms on the left side rather than just

symbols. For example, `(setf (car a) b)` sets the car of `a` to `b`, doing the same operation as `(setcar a b)` but without having to remember two separate functions for setting and accessing each type of place.

Generalized variables are analogous to "lvalues" in the C language, where `x = a[i]' gets an element from an array and `a[i] = x' stores an element using the same notation. Just as certain forms like `a[i]` can be lvalues in C, there is a set of forms that can be generalized variables in Lisp.

Basic Setf

The `setf` macro is the most basic way to operate on generalized variables.

Special Form: **setf** *[place form]...*

This macro evaluates form and stores it in place, which must be a valid generalized variable form. If there are several place and form pairs, the assignments are done sequentially just as with `setq`. `setf` returns the value of the last form.

The following Lisp forms will work as generalized variables, and so may legally appear in the place argument of `setf`:

- A symbol naming a variable. In other words, `(setf x y)` is exactly equivalent to `(setq x y)`, and `setq` itself is strictly speaking redundant now that `setf` exists. Many programmers continue to prefer `setq` for setting simple variables, though, purely for stylistic or historical reasons. The macro `(setf x y)` actually expands to `(setq x y)`, so there is no performance penalty for using it in compiled code.
- A call to any of the following Lisp functions:

```
car                 cdr                 caar .. cddddr
nth                 rest                first .. tenth
aref                elt                 nthcdr
symbol-function     symbol-value        symbol-plist
get                 get*                getf
gethash             subseq
```

  Note that for `nthcdr` and `getf`, the list argument of the function must itself be a valid place form. For example, `(setf (nthcdr 0 foo) 7)` will set `foo` itself to 7. Note that `push` and `pop` on an `nthcdr` place can be used to insert or delete at any position in a list. The use of `nthcdr` as a place form is an extension to standard Common Lisp.
- The following Emacs-specific functions are also `setf`-able. (Some of these are defined only in Emacs 19 or only in Lucid Emacs.)

```
buffer-file-name                marker-position
buffer-modified-p               match-data
buffer-name                     mouse-position
buffer-string                   overlay-end
buffer-substring                overlay-get
```


```
current-buffer                  overlay-start
current-case-table              point
current-column                  point-marker
current-global-map              point-max
current-input-mode              point-min
current-local-map               process-buffer
current-window-configuration    process-filter
default-file-modes              process-sentinel
default-value                   read-mouse-position
documentation-property          screen-height
extent-data                     screen-menubar
extent-end-position             screen-width
extent-start-position           selected-window
face-background                 selected-screen
face-background-pixmap          selected-frame
face-font                       standard-case-table
face-foreground                 syntax-table
face-underline-p                window-buffer
file-modes                      window-dedicated-p
frame-height                    window-display-table
frame-parameters                window-height
frame-visible-p                 window-hscroll
frame-width                     window-point
get-register                    window-start
getenv                          window-width
global-key-binding              x-get-cut-buffer
keymap-parent                   x-get-cutbuffer
local-key-binding               x-get-secondary-selection
mark                            x-get-selection
mark-marker
```

Most of these have directly corresponding "set" functions, like `use-local-map` for `current-local-map`, or `goto-char` for `point`. A few, like `point-min`, expand to longer sequences of code when they are `setf`'d (`(narrow-to-region x (point-max))` in this case).

- A call of the form `(substring subplace n [m])`, where subplace is itself a legal generalized variable whose current value is a string, and where the value stored is also a string. The new string is spliced into the specified part of the destination string. For example:

```
(setq a (list "hello" "world"))
     => ("hello" "world")
(cadr a)
     => "world"
(substring (cadr a) 2 4)
     => "rl"
```


```
(setf (substring (cadr a) 2 4) "o")
     => "o"
(cadr a)
     => "wood"
a
     => ("hello" "wood")
```

The generalized variable `buffer-substring`, listed above, also works in this way by replacing a portion of the current buffer.

- A call of the form `(apply 'func ...)` or `(apply (function func) ...)`, where func is a `setf`-able function whose store function is "suitable" in the sense described in Steele's book; since none of the standard Emacs place functions are suitable in this sense, this feature is only interesting when used with places you define yourself with `define-setf-method` or the long form of `defsetf`.
- A macro call, in which case the macro is expanded and `setf` is applied to the resulting form.
- Any form for which a `defsetf` or `define-setf-method` has been made.

Using any forms other than these in the place argument to `setf` will signal an error.

The `setf` macro takes care to evaluate all subforms in the proper left-to-right order; for example,

```
(setf (aref vec (incf i)) i)
```

looks like it will evaluate `(incf i)` exactly once, before the following access to `i`; the `setf` expander will insert temporary variables as necessary to ensure that it does in fact work this way no matter what setf-method is defined for `aref`. (In this case, `aset` would be used and no such steps would be necessary since `aset` takes its arguments in a convenient order.)

However, if the place form is a macro which explicitly evaluates its arguments in an unusual order, this unusual order will be preserved. Adapting an example from Steele, given

```
(defmacro wrong-order (x y) (list 'aref y x))
```

the form `(setf (wrong-order a b) 17)` will evaluate b first, then a, just as in an actual call to `wrong-order`.

Modify Macros

This package defines a number of other macros besides `setf` that operate on generalized variables. Many are interesting and useful even when the place is just a variable name.

Special Form: **psetf** *[place form]...*

This macro is to `setf` what `psetq` is to `setq`: When several places and forms are involved, the assignments take place in parallel rather than sequentially. Specifically, all subforms are evaluated from left to right, then all the assignments are done (in an undefined order).

Special Form: **incf** *place &optional x*

This macro increments the number stored in place by one, or by x if specified. The incremented value is returned. For example, `(incf i)` is equivalent to `(setq i (1+ i))`, and `(incf (car x) 2)` is equivalent to `(setcar x (+ (car x) 2))`.

Once again, care is taken to preserve the "apparent" order of evaluation. For example,

```
(incf (aref vec (incf i)))
```

appears to increment `i` once, then increment the element of `vec` addressed by `i`; this is indeed exactly what it does, which means the above form is *not* equivalent to the "obvious" expansion,

```
(setf (aref vec (incf i)) (1+ (aref vec (incf i))))   ; Wrong!
```

but rather to something more like

```
(let ((temp (incf i)))
  (setf (aref vec temp) (1+ (aref vec temp))))
```

Again, all of this is taken care of automatically by `incf` and the other generalized-variable macros.

As a more Emacs-specific example of `incf`, the expression `(incf (point) n)` is essentially equivalent to `(forward-char n)`.

Special Form: **decf** *place &optional x*

This macro decrements the number stored in place by one, or by x if specified.

Special Form: **pop** *place*

This macro removes and returns the first element of the list stored in place. It is analogous to `(prog1 (car place) (setf place (cdr place)))`, except that it takes care to evaluate all subforms only once.

Special Form: **push** *x place*

This macro inserts x at the front of the list stored in place. It is analogous to `(setf place (cons x place))`, except for evaluation of the subforms.

Special Form: **pushnew** *x place* `&key :test :test-not :key`

This macro inserts x at the front of the list stored in place, but only if x was not `eql` to any existing element of the list. The optional keyword arguments are interpreted in the same way as for `adjoin`. See section Lists as Sets.

Special Form: **shiftf** *place... newvalue*

This macro shifts the places left by one, shifting in the value of newvalue (which may be any Lisp expression, not just a generalized variable), and returning the value shifted out of the first place. Thus, `(shiftf a b c d)` is equivalent to


```
(prog1
    a
  (psetf a b
         b c
         c d))
```

except that the subforms of a, b, and c are actually evaluated only once each and in the apparent order.

Special Form: **rotatef** *place...*

This macro rotates the places left by one in circular fashion. Thus, `(rotatef a b c d)` is equivalent to

```
(psetf a b
       b c
       c d
       d a)
```

except for the evaluation of subforms. `rotatef` always returns `nil`. Note that `(rotatef a b)` conveniently exchanges a and b.

The following macros were invented for this package; they have no analogues in Common Lisp.

Special Form: **letf** *(bindings...) forms...*

This macro is analogous to `let`, but for generalized variables rather than just symbols. Each binding should be of the form `(place value)`; the original contents of the places are saved, the values are stored in them, and then the body forms are executed. Afterwards, the places are set back to their original saved contents. This cleanup happens even if the forms exit irregularly due to a `throw` or an error.

For example,

```
(letf (((point) (point-min))
       (a 17))
  ...)
```

moves "point" in the current buffer to the beginning of the buffer, and also binds `a` to 17 (as if by a normal `let`, since `a` is just a regular variable). After the body exits, `a` is set back to its original value and point is moved back to its original position.

Note that `letf` on `(point)` is not quite like a `save-excursion`, as the latter effectively saves a marker which tracks insertions and deletions in the buffer. Actually, a `letf` of `(point-marker)` is much closer to this behavior. (`point` and `point-marker` are equivalent as `setf` places; each will accept either an integer or a marker as the stored value.)

Since generalized variables look like lists, `let`'s shorthand of using `foo' for `(foo nil)' as a binding would be ambiguous in `letf` and is not allowed.

However, a binding specifier may be a one-element list `(place)', which is similar to `(place place)'. In other words, the place is not disturbed on entry to the body, and the only effect of the `letf' is to restore the original value of place afterwards. (The redundant access-and-store suggested by the `(place place)' example does not actually occur.)

In most cases, the place must have a well-defined value on entry to the `letf' form. The only exceptions are plain variables and calls to `symbol-value' and `symbol-function'. If the symbol is not bound on entry, it is simply made unbound by `makunbound' or `fmakunbound' on exit.

Special Form: **letf*** *(bindings...) forms...*

This macro is to `letf' what `let*' is to `let': It does the bindings in sequential rather than parallel order.

Special Form: **callf** *function place args...*

This is the "generic" modify macro. It calls function, which should be an unquoted function name, macro name, or lambda. It passes place and args as arguments, and assigns the result back to place. For example, `(incf place n)' is the same as `(callf + place n)'. Some more examples:

```
(callf abs my-number)
(callf concat (buffer-name) "<" (int-to-string n) ">")
(callf union happy-people (list joe bob) :test 'same-person)
```

See section Customizing Setf, for `define-modify-macro', a way to create even more concise notations for modify macros. Note again that `callf' is an extension to standard Common Lisp.

Special Form: **callf2** *function arg1 place args...*

This macro is like `callf', except that place is the *second* argument of function rather than the first. For example, `(push x place)' is equivalent to `(callf2 cons x place)'.

The `callf' and `callf2' macros serve as building blocks for other macros like `incf', `pushnew', and `define-modify-macro'. The `letf' and `letf*' macros are used in the processing of symbol macros; see section Macro Bindings.

Customizing Setf

Common Lisp defines three macros, `define-modify-macro', `defsetf', and `define-setf-method', that allow the user to extend generalized variables in various ways.

Special Form: **define-modify-macro** *name arglist function [doc-string]*

This macro defines a "read-modify-write" macro similar to `incf' and `decf'. The macro name is defined to take a place argument followed by additional arguments described by arglist. The call

```
(name place args...)
```

will be expanded to


```
(callf func place args...)
```

which in turn is roughly equivalent to

```
(setf place (func place args...))
```

For example:

```
(define-modify-macro incf (&optional (n 1)) +)
(define-modify-macro concatf (&rest args) concat)
```

Note that `&key` is not allowed in arglist, but `&rest` is sufficient to pass keywords on to the function.

Most of the modify macros defined by Common Lisp do not exactly follow the pattern of `define-modify-macro`. For example, `push` takes its arguments in the wrong order, and `pop` is completely irregular. You can define these macros "by hand" using `get-setf-method`, or consult the source file `` `cl-macs.el' `` to see how to use the internal `setf` building blocks.

Special Form: **defsetf** *access-fn update-fn*

This is the simpler of two `defsetf` forms. Where access-fn is the name of a function which accesses a place, this declares update-fn to be the corresponding store function. From now on,

```
(setf (access-fn arg1 arg2 arg3) value)
```

will be expanded to

```
(update-fn arg1 arg2 arg3 value)
```

The update-fn is required to be either a true function, or a macro which evaluates its arguments in a function-like way. Also, the update-fn is expected to return value as its result. Otherwise, the above expansion would not obey the rules for the way `setf` is supposed to behave.

As a special (non-Common-Lisp) extension, a third argument of `t` to `defsetf` says that the `update-fn`'s return value is not suitable, so that the above `setf` should be expanded to something more like

```
(let ((temp value))
  (update-fn arg1 arg2 arg3 temp)
  temp)
```

Some examples of the use of `defsetf`, drawn from the standard suite of setf methods, are:

```
(defsetf car setcar)
(defsetf symbol-value set)
(defsetf buffer-name rename-buffer t)
```

Special Form: **defsetf** *access-fn arglist (store-var) forms...*

This is the second, more complex, form of `defsetf`. It is rather like `defmacro` except for the additional store-var argument. The forms should return a Lisp form which stores the value of store-var into the generalized variable formed by a call to access-fn with arguments described by arglist. The forms may begin with a string which documents the `setf` method (analogous to the doc string that appears at the front of a function).

For example, the simple form of `defsetf` is shorthand for

```
(defsetf access-fn (&rest args) (store)
  (append '(update-fn) args (list store)))
```

The Lisp form that is returned can access the arguments from arglist and store-var in an unrestricted fashion; macros like `setf` and `incf` which invoke this setf-method will insert temporary variables as needed to make sure the apparent order of evaluation is preserved.

Another example drawn from the standard package:

```
(defsetf nth (n x) (store)
  (list 'setcar (list 'nthcdr n x) store))
```

Special Form: **define-setf-method** *access-fn arglist forms...*

This is the most general way to create new place forms. When a `setf` to access-fn with arguments described by arglist is expanded, the forms are evaluated and must return a list of five items:

1. A list of temporary variables.
2. A list of value forms corresponding to the temporary variables above. The temporary variables will be bound to these value forms as the first step of any operation on the generalized variable.
3. A list of exactly one store variable (generally obtained from a call to `gensym`).
4. A Lisp form which stores the contents of the store variable into the generalized variable, assuming the temporaries have been bound as described above.
5. A Lisp form which accesses the contents of the generalized variable, assuming the temporaries have been bound.

This is exactly like the Common Lisp macro of the same name, except that the method returns a list of five values rather than the five values themselves, since Emacs Lisp does not support Common Lisp's notion of multiple return values.

Once again, the forms may begin with a documentation string.

A setf-method should be maximally conservative with regard to temporary variables. In the setf-methods generated by `defsetf`, the second return value is simply the list of arguments in the place form, and the first return value is a list of a corresponding number of temporary variables generated by `gensym`. Macros like `setf` and `incf` which use this setf-method will optimize away most temporaries that turn out to be unnecessary, so there is little reason for the setf-method itself to optimize.

Function: **get-setf-method** *place &optional env*

This function returns the setf-method for place, by invoking the definition previously recorded by `defsetf` or `define-setf-method`. The result is a list of five values as described above. You can use this function to build your own `incf`-like modify macros. (Actually, it is better to use the internal functions `cl-setf-do-modify` and `cl-setf-do-store`, which are a bit easier to use and which also do a number of optimizations; consult the source code for the `incf` function for a simple example.)

The argument env specifies the "environment" to be passed on to `macroexpand` if `get-setf-method` should need to expand a macro in place. It should come from an `&environment` argument to the macro or setf-method that called `get-setf-method`.

See also the source code for the setf-methods for `apply` and `substring`, each of which works by calling `get-setf-method` on a simpler case, then massaging the result in various ways.

Modern Common Lisp defines a second, independent way to specify the `setf` behavior of a function, namely "`setf` functions" whose names are lists `(setf name)` rather than symbols. For example, `(defun (setf foo) ...)` defines the function that is used when `setf` is applied to `foo`. This package does not currently support `setf` functions. In particular, it is a compile-time error to use `setf` on a form which has not already been `defsetf`'d or otherwise declared; in newer Common Lisps, this would not be an error since the function `(setf func)` might be defined later.

@secno=4

Variable Bindings

These Lisp forms make bindings to variables and function names, analogous to Lisp's built-in `let` form.

See section Modify Macros, for the `letf` and `letf*` forms which are also related to variable bindings.

Dynamic Bindings

The standard `let` form binds variables whose names are known at compile-time. The `progv` form provides an easy way to bind variables whose names are computed at run-time.

Special Form: **progv** *symbols values forms...*

This form establishes `let`-style variable bindings on a set of variables computed at run-time. The expressions symbols and values are evaluated, and must return lists of symbols and values, respectively. The symbols are bound to the corresponding values for the duration of the body forms. If values is shorter than symbols, the last few symbols are made unbound (as if by `makunbound`) inside the body. If symbols is shorter than values, the excess values are ignored.

Lexical Bindings

The CL package defines the following macro which more closely follows the Common Lisp `let` form:

Special Form: **lexical-let** *(bindings...) forms...*

This form is exactly like `let` except that the bindings it establishes are purely lexical. Lexical bindings

are similar to local variables in a language like C: Only the code physically within the body of the `lexical-let` (after macro expansion) may refer to the bound variables.

```
(setq a 5)
(defun foo (b) (+ a b))
(let ((a 2)) (foo a))
     => 4
(lexical-let ((a 2)) (foo a))
     => 7
```

In this example, a regular `let` binding of `a` actually makes a temporary change to the global variable `a`, so `foo` is able to see the binding of `a` to 2. But `lexical-let` actually creates a distinct local variable `a` for use within its body, without any effect on the global variable of the same name.

The most important use of lexical bindings is to create closures. A closure is a function object that refers to an outside lexical variable. For example:

```
(defun make-adder (n)
  (lexical-let ((n n))
    (function (lambda (m) (+ n m)))))
(setq add17 (make-adder 17))
(funcall add17 4)
     => 21
```

The call `(make-adder 17)` returns a function object which adds 17 to its argument. If `let` had been used instead of `lexical-let`, the function object would have referred to the global `n`, which would have been bound to 17 only during the call to `make-adder` itself.

```
(defun make-counter ()
  (lexical-let ((n 0))
    (function* (lambda (&optional (m 1)) (incf n m)))))
(setq count-1 (make-counter))
(funcall count-1 3)
     => 3
(funcall count-1 14)
     => 17
(setq count-2 (make-counter))
(funcall count-2 5)
     => 5
(funcall count-1 2)
     => 19
(funcall count-2)
     => 6
```

Here we see that each call to `make-counter` creates a distinct local variable `n`, which serves as a private counter for the function object that is returned.

Closed-over lexical variables persist until the last reference to them goes away, just like all other Lisp objects. For example, `count-2` refers to a function object which refers to an instance of the variable `n`; this is the only reference to that variable, so after `(setq count-2 nil)` the garbage collector would be able to delete this instance of `n`. Of course, if a `lexical-let` does not actually create any closures, then the lexical variables are free as soon as the `lexical-let` returns.

Many closures are used only during the extent of the bindings they refer to; these are known as "downward funargs" in Lisp parlance. When a closure is used in this way, regular Emacs Lisp dynamic bindings suffice and will be more efficient than `lexical-let` closures:

```
(defun add-to-list (x list)
  (mapcar (function (lambda (y) (+ x y))) list))
(add-to-list 7 '(1 2 5))
     => (8 9 12)
```

Since this lambda is only used while `x` is still bound, it is not necessary to make a true closure out of it.

You can use `defun` or `flet` inside a `lexical-let` to create a named closure. If several closures are created in the body of a single `lexical-let`, they all close over the same instance of the lexical variable.

The `lexical-let` form is an extension to Common Lisp. In true Common Lisp, all bindings are lexical unless declared otherwise.

Special Form: **lexical-let*** *(bindings...) forms...*

This form is just like `lexical-let`, except that the bindings are made sequentially in the manner of `let*`.

Function Bindings

These forms make `let`-like bindings to functions instead of variables.

Special Form: **flet** *(bindings...) forms...*

This form establishes `let`-style bindings on the function cells of symbols rather than on the value cells. Each binding must be a list of the form `(name arglist forms...)', which defines a function exactly as if it were a `defun*` form. The function name is defined accordingly for the duration of the body of the `flet`; then the old function definition, or lack thereof, is restored.

While `flet` in Common Lisp establishes a lexical binding of name, Emacs Lisp `flet` makes a dynamic binding. The result is that `flet` affects indirect calls to a function as well as calls directly inside the `flet` form itself.

You can use `flet` to disable or modify the behavior of a function in a temporary fashion. This will even work on Emacs primitives, although note that some calls to primitive functions internal to Emacs are made without going through the symbol's function cell, and so will not be affected by `flet`. For example,


```
(flet ((message (&rest args) (push args saved-msgs)))
  (do-something))
```

This code attempts to replace the built-in function `message` with a function that simply saves the messages in a list rather than displaying them. The original definition of `message` will be restored after `do-something` exits. This code will work fine on messages generated by other Lisp code, but messages generated directly inside Emacs will not be caught since they make direct C-language calls to the message routines rather than going through the Lisp `message` function.

Functions defined by `flet` may use the full Common Lisp argument notation supported by `defun*`; also, the function body is enclosed in an implicit block as if by `defun*`. See section Program Structure.

Special Form: **labels** *(bindings...) forms...*

The `labels` form is like `flet`, except that it makes lexical bindings of the function names rather than dynamic bindings. (In true Common Lisp, both `flet` and `labels` make lexical bindings of slightly different sorts; since Emacs Lisp is dynamically bound by default, it seemed more appropriate for `flet` also to use dynamic binding. The `labels` form, with its lexical binding, is fully compatible with Common Lisp.)

Lexical scoping means that all references to the named functions must appear physically within the body of the `labels` form. References may appear both in the body forms of `labels` itself, and in the bodies of the functions themselves. Thus, `labels` can define local recursive functions, or mutually-recursive sets of functions.

A "reference" to a function name is either a call to that function, or a use of its name quoted by `quote` or `function` to be passed on to, say, `mapcar`.

Macro Bindings

These forms create local macros and "symbol macros."

Special Form: **macrolet** *(bindings...) forms...*

This form is analogous to `flet`, but for macros instead of functions. Each binding is a list of the same form as the arguments to `defmacro*` (i.e., a macro name, argument list, and macro-expander forms). The macro is defined accordingly for use within the body of the `macrolet`.

Because of the nature of macros, `macrolet` is lexically scoped even in Emacs Lisp: The `macrolet` binding will affect only calls that appear physically within the body forms, possibly after expansion of other macros in the body.

Special Form: **symbol-macrolet** *(bindings...) forms...*

This form creates symbol macros, which are macros that look like variable references rather than function calls. Each binding is a list `(var expansion)'; any reference to var within the body forms is replaced by expansion.

```
(setq bar '(5 . 9))
```


```
(symbol-macrolet ((foo (car bar)))
  (incf foo))
bar
     => (6 . 9)
```

A `setq` of a symbol macro is treated the same as a `setf`. I.e., `(setq foo 4)` in the above would be equivalent to `(setf foo 4)`, which in turn expands to `(setf (car bar) 4)`.

Likewise, a `let` or `let*` binding a symbol macro is treated like a `letf` or `letf*`. This differs from true Common Lisp, where the rules of lexical scoping cause a `let` binding to shadow a `symbol-macrolet` binding. In this package, only `lexical-let` and `lexical-let*` will shadow a symbol macro.

There is no analogue of `defmacro` for symbol macros; all symbol macros are local. A typical use of `symbol-macrolet` is in the expansion of another macro:

```
(defmacro* my-dolist ((x list) &rest body)
  (let ((var (gensym)))
    (list 'loop 'for var 'on list 'do
          (list* 'symbol-macrolet (list (list x (list 'car var)))
                 body))))

(setq mylist '(1 2 3 4))
(my-dolist (x mylist) (incf x))
mylist
     => (2 3 4 5)
```

In this example, the `my-dolist` macro is similar to `dolist` (see section Iteration) except that the variable `x` becomes a true reference onto the elements of the list. The `my-dolist` call shown here expands to

```
(loop for G1234 on mylist do
      (symbol-macrolet ((x (car G1234)))
        (incf x)))
```

which in turn expands to

```
(loop for G1234 on mylist do (incf (car G1234)))
```

See section Loop Facility, for a description of the `loop` macro. This package defines a nonstandard `in-ref` loop clause that works much like `my-dolist`.

Conditionals

These conditional forms augment Emacs Lisp's simple `if`, `and`, `or`, and `cond` forms.

Special Form: **when** *test forms...*

This is a variant of `if` where there are no "else" forms, and possibly several "then" forms. In particular,

```
(when test a b c)
```

is entirely equivalent to

```
(if test (progn a b c) nil)
```

Special Form: **unless** *test forms...*

This is a variant of `if` where there are no "then" forms, and possibly several "else" forms:

```
(unless test a b c)
```

is entirely equivalent to

```
(when (not test) a b c)
```

Special Form: **case** *keyform clause...*

This macro evaluates keyform, then compares it with the key values listed in the various clauses. Whichever clause matches the key is executed; comparison is done by `eql`. If no clause matches, the `case` form returns `nil`. The clauses are of the form

```
(keylist body-forms...)
```

where keylist is a list of key values. If there is exactly one value, and it is not a cons cell or the symbol `nil` or `t`, then it can be used by itself as a keylist without being enclosed in a list. All key values in the `case` form must be distinct. The final clauses may use `t` in place of a keylist to indicate a default clause that should be taken if none of the other clauses match. (The symbol `otherwise` is also recognized in place of `t`. To make a clause that matches the actual symbol `t`, `nil`, or `otherwise`, enclose the symbol in a list.)

For example, this expression reads a keystroke, then does one of four things depending on whether it is an `a', a `b', a RET or LFD, or anything else.

```
(case (read-char)
  (?a (do-a-thing))
  (?b (do-b-thing))
  ((?\r ?\n) (do-ret-thing))
  (t (do-other-thing)))
```

Special Form: **ecase** *keyform clause...*

This macro is just like `case`, except that if the key does not match any of the clauses, an error is signaled rather than simply returning `nil`.

Special Form: **typecase** *keyform clause...*

This macro is a version of `case` that checks for types rather than values. Each clause is of the form `(type body...)'. See section Type Predicates, for a description of type specifiers. For example,

```
(typecase x
  (integer (munch-integer x))
  (float (munch-float x))
  (string (munch-integer (string-to-int x)))
  (t (munch-anything x)))
```

The type specifier `t` matches any type of object; the word `otherwise` is also allowed. To make one clause match any of several types, use an `(or ...)` type specifier.

Special Form: **etypecase** *keyform clause...*

This macro is just like `typecase`, except that if the key does not match any of the clauses, an error is signaled rather than simply returning `nil`.

Blocks and Exits

Common Lisp blocks provide a non-local exit mechanism very similar to `catch` and `throw`, but lexically rather than dynamically scoped. This package actually implements `block` in terms of `catch`; however, the lexical scoping allows the optimizing byte-compiler to omit the costly `catch` step if the body of the block does not actually `return-from` the block.

Special Form: **block** *name forms...*

The forms are evaluated as if by a `progn`. However, if any of the forms execute `(return-from name)`, they will jump out and return directly from the `block` form. The `block` returns the result of the last form unless a `return-from` occurs.

The `block`/`return-from` mechanism is quite similar to the `catch`/`throw` mechanism. The main differences are that block names are unevaluated symbols, rather than forms (such as quoted symbols) which evaluate to a tag at run-time; and also that blocks are lexically scoped whereas `catch`/`throw` are dynamically scoped. This means that functions called from the body of a `catch` can also `throw` to the `catch`, but the `return-from` referring to a block name must appear physically within the forms that make up the body of the block. They may not appear within other called functions, although they may appear within macro expansions or `lambda`s in the body. Block names and `catch` names form independent name-spaces.

In true Common Lisp, `defun` and `defmacro` surround the function or expander bodies with implicit blocks with the same name as the function or macro. This does not occur in Emacs Lisp, but this package

provides `defun*` and `defmacro*` forms which do create the implicit block.

The Common Lisp looping constructs defined by this package, such as `loop` and `dolist`, also create implicit blocks just as in Common Lisp.

Because they are implemented in terms of Emacs Lisp `catch` and `throw`, blocks have the same overhead as actual `catch` constructs (roughly two function calls). However, Zawinski and Furuseth's optimizing byte compiler (standard in Emacs 19) will optimize away the `catch` if the block does not in fact contain any `return` or `return-from` calls that jump to it. This means that `do` loops and `defun*` functions which don't use `return` don't pay the overhead to support it.

Special Form: **return-from** *name [result]*

This macro returns from the block named name, which must be an (unevaluated) symbol. If a result form is specified, it is evaluated to produce the result returned from the `block`. Otherwise, `nil` is returned.

Special Form: **return** *[result]*

This macro is exactly like `(return-from nil result)`. Common Lisp loops like `do` and `dolist` implicitly enclose themselves in `nil` blocks.

Iteration

The macros described here provide more sophisticated, high-level looping constructs to complement Emacs Lisp's basic `while` loop.

Special Form: **loop** *forms...*

The CL package supports both the simple, old-style meaning of `loop` and the extremely powerful and flexible feature known as the Loop Facility or Loop Macro. This more advanced facility is discussed in the following section; see section Loop Facility. The simple form of `loop` is described here.

If `loop` is followed by zero or more Lisp expressions, then `(loop exprs...)` simply creates an infinite loop executing the expressions over and over. The loop is enclosed in an implicit `nil` block. Thus,

```
(loop (foo)  (if (no-more) (return 72))  (bar))
```

is exactly equivalent to

```
(block nil (while t (foo)  (if (no-more) (return 72))  (bar)))
```

If any of the expressions are plain symbols, the loop is instead interpreted as a Loop Macro specification as described later. (This is not a restriction in practice, since a plain symbol in the above notation would simply access and throw away the value of a variable.)

Special Form: **do** *(spec...) (end-test [result...]) forms...*

This macro creates a general iterative loop. Each spec is of the form


```
(var [init [step]])
```

The loop works as follows: First, each var is bound to the associated init value as if by a `let` form. Then, in each iteration of the loop, the end-test is evaluated; if true, the loop is finished. Otherwise, the body forms are evaluated, then each var is set to the associated step expression (as if by a `psetq` form) and the next iteration begins. Once the end-test becomes true, the result forms are evaluated (with the vars still bound to their values) to produce the result returned by `do`.

The entire `do` loop is enclosed in an implicit `nil` block, so that you can use `(return)` to break out of the loop at any time.

If there are no result forms, the loop returns `nil`. If a given var has no step form, it is bound to its init value but not otherwise modified during the `do` loop (unless the code explicitly modifies it); this case is just a shorthand for putting a `(let ((var init)) ...)` around the loop. If init is also omitted it defaults to `nil`, and in this case a plain `var' can be used in place of `(var)', again following the analogy with `let`.

This example (from Steele) illustrates a loop which applies the function `f` to successive pairs of values from the lists `foo` and `bar`; it is equivalent to the call `(mapcar* 'f foo bar)`. Note that this loop has no body forms at all, performing all its work as side effects of the rest of the loop.

```
(do ((x foo (cdr x))
     (y bar (cdr y))
     (z nil (cons (f (car x) (car y)) z)))
  ((or (null x) (null y))
   (nreverse z)))
```

Special Form: **do*** *(spec...) (end-test [result...]) forms...*

This is to `do` what `let*` is to `let`. In particular, the initial values are bound as if by `let*` rather than `let`, and the steps are assigned as if by `setq` rather than `psetq`.

Here is another way to write the above loop:

```
(do* ((xp foo (cdr xp))
      (yp bar (cdr yp))
      (x (car xp) (car xp))
      (y (car yp) (car yp))
      z)
  ((or (null xp) (null yp))
   (nreverse z))
  (push (f x y) z))
```

Special Form: **dolist** *(var list [result]) forms...*

This is a more specialized loop which iterates across the elements of a list. list should evaluate to a list; the body forms are executed with var bound to each element of the list in turn. Finally, the result form (or

`nil`) is evaluated with var bound to `nil` to produce the result returned by the loop. The loop is surrounded by an implicit `nil` block.

Special Form: **dotimes** *(var count [result]) forms...*

This is a more specialized loop which iterates a specified number of times. The body is executed with var bound to the integers from zero (inclusive) to count (exclusive), in turn. Then the `result` form is evaluated with var bound to the total number of iterations that were done (i.e., `(max 0 count)`) to get the return value for the loop form. The loop is surrounded by an implicit `nil` block.

Special Form: **do-symbols** *(var [obarray [result]]) forms...*

This loop iterates over all interned symbols. If obarray is specified and is not `nil`, it loops over all symbols in that obarray. For each symbol, the body forms are evaluated with var bound to that symbol. The symbols are visited in an unspecified order. Afterward the result form, if any, is evaluated (with var bound to `nil`) to get the return value. The loop is surrounded by an implicit `nil` block.

Special Form: **do-all-symbols** *(var [result]) forms...*

This is identical to `do-symbols` except that the obarray argument is omitted; it always iterates over the default obarray.

See section Mapping over Sequences, for some more functions for iterating over vectors or lists.

Loop Facility

A common complaint with Lisp's traditional looping constructs is that they are either too simple and limited, such as Common Lisp's `dotimes` or Emacs Lisp's `while`, or too unreadable and obscure, like Common Lisp's `do` loop.

To remedy this, recent versions of Common Lisp have added a new construct called the "Loop Facility" or "`loop` macro," with an easy-to-use but very powerful and expressive syntax.

Loop Basics

The `loop` macro essentially creates a mini-language within Lisp that is specially tailored for describing loops. While this language is a little strange-looking by the standards of regular Lisp, it turns out to be very easy to learn and well-suited to its purpose.

Since `loop` is a macro, all parsing of the loop language takes place at byte-compile time; compiled `loop`s are just as efficient as the equivalent `while` loops written longhand.

Special Form: **loop** *clauses...*

A loop construct consists of a series of clauses, each introduced by a symbol like `for` or `do`. Clauses are simply strung together in the argument list of `loop`, with minimal extra parentheses. The various types of clauses specify initializations, such as the binding of temporary variables, actions to be taken in the loop, stepping actions, and final cleanup.

Common Lisp specifies a certain general order of clauses in a loop:

```
(loop name-clause
      var-clauses...
      action-clauses...)
```

The name-clause optionally gives a name to the implicit block that surrounds the loop. By default, the implicit block is named `nil`. The var-clauses specify what variables should be bound during the loop, and how they should be modified or iterated throughout the course of the loop. The action-clauses are things to be done during the loop, such as computing, collecting, and returning values.

The Emacs version of the `loop` macro is less restrictive about the order of clauses, but things will behave most predictably if you put the variable-binding clauses `with`, `for`, and `repeat` before the action clauses. As in Common Lisp, `initially` and `finally` clauses can go anywhere.

Loops generally return `nil` by default, but you can cause them to return a value by using an accumulation clause like `collect`, an end-test clause like `always`, or an explicit `return` clause to jump out of the implicit block. (Because the loop body is enclosed in an implicit block, you can also use regular Lisp `return` or `return-from` to break out of the loop.)

The following sections give some examples of the Loop Macro in action, and describe the particular loop clauses in great detail. Consult the second edition of Steele's Common Lisp, the Language, for additional discussion and examples of the `loop` macro.

Loop Examples

Before listing the full set of clauses that are allowed, let's look at a few example loops just to get a feel for the `loop` language.

```
(loop for buf in (buffer-list)
      collect (buffer-file-name buf))
```

This loop iterates over all Emacs buffers, using the list returned by `buffer-list`. For each buffer `buf`, it calls `buffer-file-name` and collects the results into a list, which is then returned from the `loop` construct. The result is a list of the file names of all the buffers in Emacs' memory. The words `for`, `in`, and `collect` are reserved words in the `loop` language.

```
(loop repeat 20 do (insert "Yowsa\n"))
```

This loop inserts the phrase "Yowsa" twenty times in the current buffer.

```
(loop until (eobp) do (munch-line) (forward-line 1))
```

This loop calls `munch-line` on every line until the end of the buffer. If point is already at the end of the buffer, the loop exits immediately.

```
(loop do (munch-line) until (eobp) do (forward-line 1))
```

This loop is similar to the above one, except that `munch-line` is always called at least once.

```
(loop for x from 1 to 100
      for y = (* x x)
      until (>= y 729)
      finally return (list x (= y 729)))
```

This more complicated loop searches for a number `x` whose square is 729. For safety's sake it only examines `x` values up to 100; dropping the phrase `to 100' would cause the loop to count upwards with no limit. The second `for` clause defines `y` to be the square of `x` within the loop; the expression after the `=` sign is reevaluated each time through the loop. The `until` clause gives a condition for terminating the loop, and the `finally` clause says what to do when the loop finishes. (This particular example was written less concisely than it could have been, just for the sake of illustration.)

Note that even though this loop contains three clauses (two `for`s and an `until`) that would have been enough to define loops all by themselves, it still creates a single loop rather than some sort of triple-nested loop. You must explicitly nest your `loop` constructs if you want nested loops.

For Clauses

Most loops are governed by one or more `for` clauses. A `for` clause simultaneously describes variables to be bound, how those variables are to be stepped during the loop, and usually an end condition based on those variables.

The word `as` is a synonym for the word `for`. This word is followed by a variable name, then a word like `from` or `across` that describes the kind of iteration desired. In Common Lisp, the phrase `being the` sometimes precedes the type of iteration; in this package both `being` and `the` are optional. The word `each` is a synonym for `the`, and the word that follows it may be singular or plural: `for x being the elements of y' or `for x being each element of y'. Which form you use is purely a matter of style.

The variable is bound around the loop as if by `let`:

```
(setq i 'happy)
(loop for i from 1 to 10 do (do-something-with i))
i
     => happy
```

`for var from expr1 to expr2 by expr3`

This type of `for` clause creates a counting loop. Each of the three sub-terms is optional, though there must be at least one term so that the clause is marked as a counting clause.

The three expressions are the starting value, the ending value, and the step value, respectively, of the variable. The loop counts upwards by default (expr3 must be positive), from expr1 to expr2 inclusively. If you omit the `from` term, the loop counts from zero; if you omit the `to` term, the loop counts forever without stopping (unless stopped by some other loop clause, of course); if you omit the `by` term, the loop counts in steps of one.

You can replace the word `from` with `upfrom` or `downfrom` to indicate the direction of the l Likewise, you can replace `to` with `upto` or `downto`. For example, `for x from 5 downto 1' executes five times with `x` taking on the integers from 5 down to 1 in turn. Also, you can replace `to` with `below` or `above`, which are like `upto` and `downto` respectively except that they are exclusive rather than inclusive limits:

```
(loop for x to 10 collect x)
     => (0 1 2 3 4 5 6 7 8 9 10)
(loop for x below 10 collect x)
     => (0 1 2 3 4 5 6 7 8 9)
```

The `by` value is always positive, even for downward-counting loops. Some sort of `from` value is required for downward loops; `for x downto 5' is not a legal loop clause all by itself.

`for var in list by function`

This clause iterates var over all the elements of list, in turn. If you specify the `by` term, then function is used to traverse the list instead of `cdr`; it must be a function taking one argument. For example:

```
(loop for x in '(1 2 3 4 5 6) collect (* x x))
     => (1 4 9 16 25 36)
(loop for x in '(1 2 3 4 5 6) by 'cddr collect (* x x))
     => (1 9 25)
```

`for var on list by function`

This clause iterates var over all the cons cells of list.

```
(loop for x on '(1 2 3 4) collect x)
     => ((1 2 3 4) (2 3 4) (3 4) (4))
```

With `by`, there is no real reason that the `on` expression must be a list. For example:

```
(loop for x on first-animal by 'next-animal collect x)
```

where `(next-animal x)` takes an "animal" x and returns the next in the (assumed) sequence of animals, or `nil` if x was the last animal in the sequence.

`for var in-ref list by function`

This is like a regular `in` clause, but var becomes a `setf`-able "reference" onto the elements of the list rather than just a temporary variable. For example,

```
(loop for x in-ref my-list do (incf x))
```

increments every element of `my-list` in place. This clause is an extension to standard Common Lisp.

`for var across array`

This clause iterates var over all the elements of array, which may be a vector or a string.


```
(loop for x across "aeiou"
      do (use-vowel (char-to-string x)))
```

`for var across-ref array`

This clause iterates over an array, with var a `setf`-able reference onto the elements; see `in-ref` above.

`for var being the elements of sequence`

This clause iterates over the elements of sequence, which may be a list, vector, or string. Since the type must be determined at run-time, this is somewhat less efficient than `in` or `across`. The clause may be followed by the additional term `using (index var2)' to cause var2 to be bound to the successive indices (starting at 0) of the elements.

This clause type is taken from older versions of the `loop` macro, and is not present in modern Common Lisp. The `using (sequence ...)' term of the older macros is not supported.

`for var being the elements of-ref sequence`

This clause iterates over a sequence, with var a `setf`-able reference onto the elements; see `in-ref` above.

`for var being the symbols [of obarray]`

This clause iterates over symbols, either over all interned symbols or over all symbols in obarray. The loop is executed with var bound to each symbol in turn. The symbols are visited in an unspecified order.

As an example,

```
(loop for sym being the symbols
      when (fboundp sym)
      when (string-match "^map" (symbol-name sym))
      collect sym)
```

returns a list of all the functions whose names begin with `map'.

The Common Lisp words `external-symbols` and `present-symbols` are also recognized but are equivalent to `symbols` in Emacs Lisp.

Due to a minor implementation restriction, it will not work to have more than one `for` clause iterating over symbols, hash tables, keymaps, overlays, or intervals in a given `loop`. Fortunately, it would rarely if ever be useful to do so. It *is* legal to mix one of these types of clauses with other clauses like `for ... to` or `while`.

`for var being the hash-keys of hash-table`

This clause iterates over the entries in hash-table. For each hash table entry, var is bound to the entry's key. If you write `the hash-values' instead, var is bound to the values of the entries. The clause may be followed by the additional term `using (hash-values var2)' (where `hash-values` is the opposite word of the word following `the`) to cause var and var2 to be bound to the two parts of each hash table entry.

`for var being the key-codes of keymap`

This clause iterates over the entries in keymap. In GNU Emacs 18 and 19, keymaps are either alists or vectors, and key-codes are integers or symbols. In Lucid Emacs 19, keymaps are a special new data type, and key-codes are symbols or lists of symbols. The iteration does not enter nested keymaps or inherited (parent) keymaps. You can use `the key-bindings' to access the commands bound to the keys rather than the key codes, and you can add a `using` clause to access both the codes and the bindings together.

`for var being the key-seqs of keymap`

This clause iterates over all key sequences defined by keymap and its nested keymaps, where var takes on values which are strings in Emacs 18 or vectors in Emacs 19. The strings or vectors are reused for each iteration, so you must copy them if you wish to keep them permanently. You can add a `using (key-bindings ...)' clause to get the command bindings as well.

`for var being the overlays [of buffer] ...`

This clause iterates over the Emacs 19 "overlays" or Lucid Emacs "extents" of a buffer (the clause `extents` is synonymous with `overlays`). Under Emacs 18, this clause iterates zero times. If the `of` term is omitted, the current buffer is used. This clause also accepts optional `from pos' and `to pos' terms, limiting the clause to overlays which overlap the specified region.

`for var being the intervals [of buffer] ...`

This clause iterates over all intervals of a buffer with constant text properties. The variable var will be bound to conses of start and end positions, where one start position is always equal to the previous end position. The clause allows `of`, `from`, `to`, and `property` terms, where the latter term restricts the search to just the specified property. The `of` term may specify either a buffer or a string. This clause is useful only in GNU Emacs 19; in other versions, all buffers and strings consist of a single interval.

`for var being the frames`

This clause iterates over all frames, i.e., X window system windows open on Emacs files. This clause works only under Emacs 19. The clause `screens` is a synonym for `frames`. The frames are visited in `next-frame` order starting from `selected-frame`.

`for var being the windows [of frame]`

This clause iterates over the windows (in the Emacs sense) of the current frame, or of the specified frame. (In Emacs 18 there is only ever one frame, and the `of` term is not allowed there.)

`for var being the buffers`

This clause iterates over all buffers in Emacs. It is equivalent to `for var in (buffer-list)'.

`for var = expr1 then expr2`

This clause does a general iteration. The first time through the loop, var will be bound to expr1. On the second and successive iterations it will be set by evaluating expr2 (which may refer to the old value of var). For example, these two loops are effectively the same:

```
(loop for x on my-list by 'cddr do ...)
(loop for x = my-list then (cddr x) while x do ...)
```

Note that this type of `for` clause does not imply any sort of terminating condition; the above

example combines it with a `while` clause to tell when to end the loop.

If you omit the `then` term, expr1 is used both for the initial setting and for successive settings:

```
(loop for x = (random) when (> x 0) return x)
```

This loop keeps taking random numbers from the `(random)` function until it gets a positive one, which it then returns.

If you include several `for` clauses in a row, they are treated sequentially (as if by `let*` and `setq`). You can instead use the word `and` to link the clauses, in which case they are processed in parallel (as if by `let` and `psetq`).

```
(loop for x below 5 for y = nil then x collect (list x y))
     => ((0 nil) (1 1) (2 2) (3 3) (4 4))
(loop for x below 5 and y = nil then x collect (list x y))
     => ((0 nil) (1 0) (2 1) (3 2) (4 3))
```

In the first loop, `y` is set based on the value of `x` that was just set by the previous clause; in the second loop, `x` and `y` are set simultaneously so `y` is set based on the value of `x` left over from the previous time through the loop.

Another feature of the `loop` macro is destructuring, similar in concept to the destructuring provided by `defmacro`. The var part of any `for` clause can be given as a list of variables instead of a single variable. The values produced during loop execution must be lists; the values in the lists are stored in the corresponding variables.

```
(loop for (x y) in '((2 3) (4 5) (6 7)) collect (+ x y))
     => (5 9 13)
```

In loop destructuring, if there are more values than variables the trailing values are ignored, and if there are more variables than values the trailing variables get the value `nil`. If `nil` is used as a variable name, the corresponding values are ignored. Destructuring may be nested, and dotted lists of variables like `(x . y)` are allowed.

Iteration Clauses

Aside from `for` clauses, there are several other loop clauses that control the way the loop operates. They might be used by themselves, or in conjunction with one or more `for` clauses.

`repeat integer`

This clause simply counts up to the specified number using an internal temporary variable. The loops

```
(loop repeat n do ...)
(loop for temp to n do ...)
```

are identical except that the second one forces you to choose a name for a variable you aren't actually going to use.

`while condition`

This clause stops the loop when the specified condition (any Lisp expression) becomes `nil`. For example, the following two loops are equivalent, except for the implicit `nil` block that surrounds the second one:

```
(while cond forms...)
(loop while cond do forms...)
```

`until condition`

This clause stops the loop when the specified condition is true, i.e., non-`nil`.

`always condition`

This clause stops the loop when the specified condition is `nil`. Unlike `while`, it stops the loop using `return nil` so that the `finally` clauses are not executed. If all the conditions were non-`nil`, the loop returns `t`:

```
(if (loop for size in size-list always (> size 10))
    (some-big-sizes)
  (no-big-sizes))
```

`never condition`

This clause is like `always`, except that the loop returns `t` if any conditions were false, or `nil` otherwise.

`thereis condition`

This clause stops the loop when the specified form is non-`nil`; in this case, it returns that non-`nil` value. If all the values were `nil`, the loop returns `nil`.

Accumulation Clauses

These clauses cause the loop to accumulate information about the specified Lisp form. The accumulated result is returned from the loop unless overridden, say, by a `return` clause.

`collect form`

This clause collects the values of form into a list. Several examples of `collect` appear elsewhere in this manual.

The word `collecting` is a synonym for `collect`, and likewise for the other accumulation clauses.

`append form`

This clause collects lists of values into a result list using `append`.

`nconc form`

This clause collects lists of values into a result list by destructively modifying the lists rather than copying them.

`concat form`

This clause concatenates the values of the specified form into a string. (It and the following clause are extensions to standard Common Lisp.)

`vconcat form`

This clause concatenates the values of the specified form into a vector.

`count form`

This clause counts the number of times the specified form evaluates to a non-`nil` value.

`sum form`

This clause accumulates the sum of the values of the specified form, which must evaluate to a number.

`maximize form`

This clause accumulates the maximum value of the specified form, which must evaluate to a number. The return value is undefined if `maximize` is executed zero times.

`minimize form`

This clause accumulates the minimum value of the specified form.

Accumulation clauses can be followed by `into var' to cause the data to be collected into variable var (which is automatically `let`-bound during the loop) rather than an unnamed temporary variable. Also, `into` accumulations do not automatically imply a return value. The loop must use some explicit mechanism, such as `finally return`, to return the accumulated result.

It is legal for several accumulation clauses of the same type to accumulate into the same place. From Steele:

```
(loop for name in '(fred sue alice joe june)
      for kids in '((bob ken) () () (kris sunshine) ())
      collect name
      append kids)
     => (fred bob ken sue alice joe kris sunshine june)
```

Other Clauses

This section describes the remaining loop clauses.

`with var = value`

This clause binds a variable to a value around the loop, but otherwise leaves the variable alone during the loop. The following loops are basically equivalent:

```
(loop with x = 17 do ...)
(let ((x 17)) (loop do ...))
(loop for x = 17 then x do ...)
```

Naturally, the variable var might be used for some purpose in the rest of the loop. For example:


```
(loop for x in my-list  with res = nil  do (push x res)
      finally return res)
```

This loop inserts the elements of `my-list` at the front of a new list being accumulated in `res`, then returns the list `res` at the end of the loop. The effect is similar to that of a `collect` clause, but the list gets reversed by virtue of the fact that elements are being pushed onto the front of `res` rather than the end.

If you omit the `=` term, the variable is initialized to `nil`. (Thus the `= nil' in the above example is unnecessary.)

Bindings made by `with` are sequential by default, as if by `let*`. Just like `for` clauses, `with` clauses can be linked with `and` to cause the bindings to be made by `let` instead.

`if condition clause`

This clause executes the following loop clause only if the specified condition is true. The following clause should be an accumulation, `do`, `return`, `if`, or `unless` clause. Several clauses may be linked by separating them with `and`. These clauses may be followed by `else` and a clause or clauses to execute if the condition was false. The whole construct may optionally be followed by the word `end` (which may be used to disambiguate an `else` or `and` in a nested `if`).

The actual non-`nil` value of the condition form is available by the name `it` in the "then" part. For example:

```
(setq funny-numbers '(6 13 -1))
     => (6 13 -1)
(loop for x below 10
      if (oddp x)
        collect x into odds
        and if (memq x funny-numbers) return (cdr it) end
      else
        collect x into evens
      finally return (vector odds evens))
     => [(1 3 5 7 9) (0 2 4 6 8)]
(setq funny-numbers '(6 7 13 -1))
     => (6 7 13 -1)
(loop <same thing again>)
     => (13 -1)
```

Note the use of `and` to put two clauses into the "then" part, one of which is itself an `if` clause. Note also that `end`, while normally optional, was necessary here to make it clear that the `else` refers to the outermost `if` clause. In the first case, the loop returns a vector of lists of the odd and even values of x. In the second case, the odd number 7 is one of the `funny-numbers` so the loop returns early; the actual returned value is based on the result of the `memq` call.

`when condition clause`

This clause is just a synonym for `if`.

`unless condition clause`

The `unless` clause is just like `if` except that the sense of the condition is reversed.

`named name`

This clause gives a name other than `nil` to the implicit block surrounding the loop. The name is the symbol to be used as the block name.

`initially [do] forms...`

This keyword introduces one or more Lisp forms which will be executed before the loop itself begins (but after any variables requested by `for` or `with` have been bound to their initial values). `initially` clauses can appear anywhere; if there are several, they are executed in the order they appear in the loop. The keyword `do` is optional.

`finally [do] forms...`

This introduces Lisp forms which will be executed after the loop finishes (say, on request of a `for` or `while`). `initially` and `finally` clauses may appear anywhere in the loop construct, but they are executed (in the specified order) at the beginning or end, respectively, of the loop.

`finally return form`

This says that form should be executed after the loop is done to obtain a return value. (Without this, or some other clause like `collect` or `return`, the loop will simply return `nil`.) Variables bound by `for`, `with`, or `into` will still contain their final values when form is executed.

`do forms...`

The word `do` may be followed by any number of Lisp expressions which are executed as an implicit `progn` in the body of the loop. Many of the examples in this section illustrate the use of `do`.

`return form`

This clause causes the loop to return immediately. The following Lisp form is evaluated to give the return value of the `loop` form. The `finally` clauses, if any, are not executed. Of course, `return` is generally used inside an `if` or `unless`, as its use in a top-level loop clause would mean the loop would never get to "loop" more than once.

The clause `return form' is equivalent to `do (return form)' (or `return-from` if the loop was named). The `return` clause is implemented a bit more efficiently, though.

While there is no high-level way to add user extensions to `loop` (comparable to `defsetf` for `setf`, say), this package does offer two properties called `cl-loop-handler` and `cl-loop-for-handler` which are functions to be called when a given symbol is encountered as a top-level loop clause or `for` clause, respectively. Consult the source code in file `` `cl-macs.el' `` for details.

This package's `loop` macro is compatible with that of Common Lisp, except that a few features are not implemented: `loop-finish` and data-type specifiers. Naturally, the `for` clauses which iterate over keymaps, overlays, intervals, frames, windows, and buffers are Emacs-specific extensions.

Multiple Values

Common Lisp functions can return zero or more results. Emacs Lisp functions, by contrast, always return exactly one result. This package makes no attempt to emulate Common Lisp multiple return values; Emacs versions of Common Lisp functions that return more than one value either return just the first value (as in `compiler-macroexpand`) or return a list of values (as in `get-setf-method`). This package *does* define placeholders for the Common Lisp functions that work with multiple values, but in Emacs Lisp these functions simply operate on lists instead. The `values` form, for example, is a synonym for `list` in Emacs.

Special Form: **multiple-value-bind** *(var...) values-form forms...*

This form evaluates values-form, which must return a list of values. It then binds the vars to these respective values, as if by `let`, and then executes the body forms. If there are more vars than values, the extra vars are bound to `nil`. If there are fewer vars than values, the excess values are ignored.

Special Form: **multiple-value-setq** *(var...) form*

This form evaluates form, which must return a list of values. It then sets the vars to these respective values, as if by `setq`. Extra vars or values are treated the same as in `multiple-value-bind`.

The older Quiroz package attempted a more faithful (but still imperfect) emulation of Common Lisp multiple values. The old method "usually" simulated true multiple values quite well, but under certain circumstances would leave spurious return values in memory where a later, unrelated `multiple-value-bind` form would see them.

Since a perfect emulation is not feasible in Emacs Lisp, this package opts to keep it as simple and predictable as possible.

Go to the previous, next section.

Go to the previous, next section.

Macros

This package implements the various Common Lisp features of `defmacro`, such as destructuring, `&environment`, and `&body`. Top-level `&whole` is not implemented for `defmacro` due to technical difficulties. See section Argument Lists.

Destructuring is made available to the user by way of the following macro:

Special Form: **destructuring-bind** *arglist expr forms...*

This macro expands to code which executes forms, with the variables in arglist bound to the list of values returned by expr. The arglist can include all the features allowed for `defmacro` argument lists, including destructuring. (The `&environment` keyword is not allowed.) The macro expansion will signal an error if expr returns a list of the wrong number of arguments or with incorrect keyword arguments.

This package also includes the Common Lisp `define-compiler-macro` facility, which allows you to define compile-time expansions and optimizations for your functions.

Special Form: **define-compiler-macro** *name arglist forms...*

This form is similar to `defmacro`, except that it only expands calls to name at compile-time; calls processed by the Lisp interpreter are not expanded, nor are they expanded by the `macroexpand` function.

The argument list may begin with a `&whole` keyword and a variable. This variable is bound to the macro-call form itself, i.e., to a list of the form `(name args...)'. If the macro expander returns this form unchanged, then the compiler treats it as a normal function call. This allows compiler macros to work as optimizers for special cases of a function, leaving complicated cases alone.

For example, here is a simplified version of a definition that appears as a standard part of this package:

```
(define-compiler-macro member* (&whole form a list &rest keys)
  (if (and (null keys)
           (eq (car-safe a) 'quote)
           (not (floatp-safe (cadr a))))
      (list 'memq a list)
    form))
```

This definition causes `(member* a list)` to change to a call to the faster `memq` in the common case where a is a non-floating-point constant; if a is anything else, or if there are any keyword arguments in the call, then the original `member*` call is left intact. (The actual compiler macro for `member*` optimizes a number of other cases, including common `:test` predicates.)

Function: **compiler-macroexpand** *form*

This function is analogous to `macroexpand`, except that it expands compiler macros rather than re macros. It returns form unchanged if it is not a call to a function for which a compiler macro has bee defined, or if that compiler macro decided to punt by returning its `&whole` argument. Like `macroexpand`, it expands repeatedly until it reaches a form for which no further expansion is possible.

See section Macro Bindings, for descriptions of the `macrolet` and `symbol-macrolet` forms for making "local" macro definitions.

Go to the previous, next section.

Go to the previous, next section.

Declarations

Common Lisp includes a complex and powerful "declaration" mechanism that allows you to give the compiler special hints about the types of data that will be stored in particular variables, and about the ways those variables and functions will be used. This package defines versions of all the Common Lisp declaration forms: `declare`, `locally`, `proclaim`, `declaim`, and `the`.

Most of the Common Lisp declarations are not currently useful in Emacs Lisp, as the byte-code system provides little opportunity to benefit from type information, and `special` declarations are redundant in a fully dynamically-scoped Lisp. A few declarations are meaningful when the optimizing Emacs 19 byte compiler is being used, however. Under the earlier non-optimizing compiler, these declarations will effectively be ignored.

Function: **proclaim** *decl-spec*

This function records a "global" declaration specified by decl-spec. Since `proclaim` is a function, decl-spec is evaluated and thus should normally be quoted.

Special Form: **declaim** *decl-specs...*

This macro is like `proclaim`, except that it takes any number of decl-spec arguments, and the arguments are unevaluated and unquoted. The `declaim` macro also puts an `(eval-when (compile load eval) ...)` around the declarations so that they will be registered at compile-time as well as at run-time. (This is vital, since normally the declarations are meant to influence the way the compiler treats the rest of the file that contains the `declaim` form.)

Special Form: **declare** *decl-specs...*

This macro is used to make declarations within functions and other code. Common Lisp allows declarations in various locations, generally at the beginning of any of the many "implicit `progn`s" throughout Lisp syntax, such as function bodies, `let` bodies, etc. Currently the only declaration understood by `declare` is `special`.

Special Form: **locally** *declarations... forms...*

In this package, `locally` is no different from `progn`.

Special Form: **the** *type form*

Type information provided by `the` is ignored in this package; in other words, `(the type form)` is equivalent to form. Future versions of the optimizing byte-compiler may make use of this information.

For example, `mapcar` can map over both lists and arrays. It is hard for the compiler to expand `mapcar` into an in-line loop unless it knows whether the sequence will be a list or an array ahead of time. With `(mapcar 'car (the vector foo))`, a future compiler would have enough information to expand the loop in-line. For now, Emacs Lisp will treat the above code as exactly equivalent to


```
(mapcar 'car foo).
```

Each decl-spec in a `proclaim`, `declaim`, or `declare` should be a list beginning with a symbol that says what kind of declaration it is. This package currently understands `special`, `inline`, `notinline`, `optimize`, and `warn` declarations. (The `warn` declaration is an extension of standard Common Lisp.) Other Common Lisp declarations, such as `type` and `ftype`, are silently ignored.

`special`

Since all variables in Emacs Lisp are "special" (in the Common Lisp sense), `special` declarations are only advisory. They simply tell the optimizing byte compiler that the specified variables are intentionally being referred to without being bound in the body of the function. The compiler normally emits warnings for such references, since they could be typographical errors for references to local variables.

The declaration `(declare (special var1 var2))` is equivalent to `(defvar var1)` `(defvar var2)` in the optimizing compiler, or to nothing at all in older compilers (which do not warn for non-local references).

In top-level contexts, it is generally better to write `(defvar var)` than `(declaim (special var))`, since `defvar` makes your intentions clearer. But the older byte compilers can not handle `defvar`s appearing inside of functions, while `(declare (special var))` takes care to work correctly with all compilers.

`inline`

The `inline` decl-spec lists one or more functions whose bodies should be expanded "in-line" into calling functions whenever the compiler is able to arrange for it. For example, the Common Lisp function `cadr` is declared `inline` by this package so that the form `(cadr x)` will expand directly into `(car (cdr x))` when it is called in user functions, for a savings of one (relatively expensive) function call.

The following declarations are all equivalent. Note that the `defsubst` form is a convenient way to define a function and declare it inline all at once, but it is available only in Emacs 19.

```
(declaim (inline foo bar))
(eval-when (compile load eval) (proclaim '(inline foo bar)))
(proclaim-inline foo bar)      ; Lucid Emacs only
(defsubst foo (...) ...)       ; instead of defun; Emacs 19 only
```

Note: This declaration remains in effect after the containing source file is done. It is correct to use it to request that a function you have defined should be inlined, but it is impolite to use it to request inlining of an external function.

In Common Lisp, it is possible to use `(declare (inline ...))` before a particular call to a function to cause just that call to be inlined; the current byte compilers provide no way to implement this, so `(declare (inline ...))` is currently ignored by this package.

`notinline`

The `notinline` declaration lists functions which should not be inlined after all; it cancels a previous `inline` declaration.

`optimize`

This declaration controls how much optimization is performed by the compiler. Naturally, it is ignored by the earlier non-optimizing compilers.

The word `optimize` is followed by any number of lists like `(speed 3)` or `(safety 2)`. Common Lisp defines several optimization "qualities"; this package ignores all but `speed` and `safety`. The value of a quality should be an integer from 0 to 3, with 0 meaning "unimportant" and 3 meaning "very important." The default level for both qualities is 1.

In this package, with the Emacs 19 optimizing compiler, the `speed` quality is tied to the `byte-compile-optimize` flag, which is set to `nil` for `(speed 0)` and to `t` for higher settings; and the `safety` quality is tied to the `byte-compile-delete-errors` flag, which is set to `t` for `(safety 3)` and to `nil` for all lower settings. (The latter flag controls whether the compiler is allowed to optimize out code whose only side-effect could be to signal an error, e.g., rewriting `(progn foo bar)` to `bar` when it is not known whether `foo` will be bound at run-time.)

Note that even compiling with `(safety 0)`, the Emacs byte-code system provides sufficient checking to prevent real harm from being done. For example, barring serious bugs in Emacs itself, Emacs will not crash with a segmentation fault just because of an error in a fully-optimized Lisp program.

The `optimize` declaration is normally used in a top-level `proclaim` or `declaim` in a file; Common Lisp allows it to be used with `declare` to set the level of optimization locally for a given form, but this will not work correctly with the current version of the optimizing compiler. (The `declare` will set the new optimization level, but that level will not automatically be unset after the enclosing form is done.)

`warn`

This declaration controls what sorts of warnings are generated by the byte compiler. Again, only the optimizing compiler generates warnings. The word `warn` is followed by any number of "warning qualities," similar in form to optimization qualities. The currently supported warning types are `redefine`, `callargs`, `unresolved`, and `free-vars`; in the current system, a value of 0 will disable these warnings and any higher value will enable them. See the documentation for the optimizing byte compiler for details.

Go to the previous, next section.

Go to the previous, next section.

Symbols

This package defines several symbol-related features that were missing from Emacs Lisp.

Property Lists

These functions augment the standard Emacs Lisp functions `get` and `put` for operating on properties attached to symbols. There are also functions for working with property lists as first-class data structures not attached to particular symbols.

Function: **get*** *symbol property &optional default*

This function is like `get`, except that if the property is not found, the default argument provides the return value. (The Emacs Lisp `get` function always uses `nil` as the default; this package's `get*` is equivalent to Common Lisp's `get`.)

The `get*` function is `setf`-able; when used in this fashion, the default argument is allowed but ignored.

Function: **remprop** *symbol property*

This function removes the entry for property from the property list of symbol. It returns a true value if the property was indeed found and removed, or `nil` if there was no such property. (This function was probably omitted from Emacs originally because, since `get` did not allow a default, it was very difficult to distinguish between a missing property and a property whose value was `nil`; thus, setting a property to `nil` was close enough to `remprop` for most purposes.)

Function: **getf** *place property &optional default*

This function scans the list place as if it were a property list, i.e., a list of alternating property names and values. If an even-numbered element of place is found which is `eq` to property, the following odd-numbered element is returned. Otherwise, default is returned (or `nil` if no default is given).

In particular,

```
(get sym prop)  ==  (getf (symbol-plist sym) prop)
```

It is legal to use `getf` as a `setf` place, in which case its place argument must itself be a legal `setf` place. The default argument, if any, is ignored in this context. The effect is to change (via `setcar`) the value cell in the list that corresponds to property, or to cons a new property-value pair onto the list if the property is not yet present.

```
(put sym prop val)  ==  (setf (getf (symbol-plist sym) prop) val)
```

The `get` and `get*` functions are also `setf`-able. The fact that `default` is ignored can sometimes be

useful:

```
(incf (get* 'foo 'usage-count 0))
```

Here, symbol `foo`'s `usage-count` property is incremented if it exists, or set to 1 (an incremented 0) otherwise.

When not used as a `setf` form, `getf` is just a regular function and its place argument can actually be any Lisp expression.

Special Form: **remf** *place property*

This macro removes the property-value pair for property from the property list stored at place, which is any `setf`-able place expression. It returns true if the property was found. Note that if property happens to be first on the list, this will effectively do a `(setf place (cddr place))`, whereas if it occurs later, this simply uses `setcdr` to splice out the property and value cells.

@secno=2

Creating Symbols

These functions create unique symbols, typically for use as temporary variables.

Function: **gensym** *&optional x*

This function creates a new, uninterned symbol (using `make-symbol`) with a unique name. (The name of an uninterned symbol is relevant only if the symbol is printed.) By default, the name is generated from an increasing sequence of numbers, `G1000', `G1001', `G1002', etc. If the optional argument x is a string, that string is used as a prefix instead of `G'. Uninterned symbols are used in macro expansions for temporary variables, to ensure that their names will not conflict with "real" variables in the user's code.

Variable: ***gensym-counter***

This variable holds the counter used to generate `gensym` names. It is incremented after each use by `gensym`. In Common Lisp this is initialized with 0, but this package initializes it with a random (time-dependent) value to avoid trouble when two files that each used `gensym` in their compilation are loaded together. (Uninterned symbols become interned when the compiler writes them out to a file and the Emacs loader loads them, so their names have to be treated a bit more carefully than in Common Lisp where uninterned symbols remain uninterned after loading.)

Function: **gentemp** *&optional x*

This function is like `gensym`, except that it produces a new *interned* symbol. If the symbol that is generated already exists, the function keeps incrementing the counter and trying again until a new symbol is generated.

The Quiroz `` `cl.el' `` package also defined a `defkeyword` form for creating self-quoting keyword symbols. This package automatically creates all keywords that are called for by `&key` argument specifiers, and discourages the use of keywords as data unrelated to keyword arguments, so the

`defkeyword` form has been discontinued.

@chapno=11

Go to the previous, next section.

Go to the previous, next section.

Numbers

This section defines a few simple Common Lisp operations on numbers which were left out of Emacs Lisp.

@secno=1

Predicates on Numbers

These functions return `t` if the specified condition is true of the numerical argument, or `nil` otherwise.

Function: **plusp** *number*

This predicate tests whether number is positive. It is an error if the argument is not a number.

Function: **minusp** *number*

This predicate tests whether number is negative. It is an error if the argument is not a number.

Function: **oddp** *integer*

This predicate tests whether integer is odd. It is an error if the argument is not an integer.

Function: **evenp** *integer*

This predicate tests whether integer is even. It is an error if the argument is not an integer.

Function: **floatp-safe** *object*

This predicate tests whether object is a floating-point number. On systems that support floating-point, this is equivalent to `floatp`. On other systems, this always returns `nil`.

@secno=3

Numerical Functions

These functions perform various arithmetic operations on numbers.

Function: **abs** *number*

This function returns the absolute value of number. (Newer versions of Emacs provide this as a built-in function; this package defines `abs` only for Emacs 18 versions which don't provide it as a primitive.)

Function: **expt** *base power*

This function returns base raised to the power of number. (Newer versions of Emacs provide this as a

built-in function; this package defines `expt` only for Emacs 18 versions which don't provide it as a primitive.)

Function: **gcd** *&rest integers*

This function returns the Greatest Common Divisor of the arguments. For one argument, it returns the absolute value of that argument. For zero arguments, it returns zero.

Function: **lcm** *&rest integers*

This function returns the Least Common Multiple of the arguments. For one argument, it returns the absolute value of that argument. For zero arguments, it returns one.

Function: **isqrt** *integer*

This function computes the "integer square root" of its integer argument, i.e., the greatest integer less than or equal to the true square root of the argument.

Function: **floor*** *number &optional divisor*

This function implements the Common Lisp `floor` function. It is called `floor*` to avoid name conflicts with the simpler `floor` function built-in to Emacs 19.

With one argument, `floor*` returns a list of two numbers: The argument rounded down (toward minus infinity) to an integer, and the "remainder" which would have to be added back to the first return value to yield the argument again. If the argument is an integer x, the result is always the list `(x 0)`. If the argument is an Emacs 19 floating-point number, the first result is a Lisp integer and the second is a Lisp float between 0 (inclusive) and 1 (exclusive).

With two arguments, `floor*` divides number by divisor, and returns the floor of the quotient and the corresponding remainder as a list of two numbers. If `(floor* x y)` returns `(q r)`, then `q*y + r = x`, with r between 0 (inclusive) and r (exclusive). Also, note that `(floor* x)` is exactly equivalent to `(floor* x 1)`.

This function is entirely compatible with Common Lisp's `floor` function, except that it returns the two results in a list since Emacs Lisp does not support multiple-valued functions.

Function: **ceiling*** *number &optional divisor*

This function implements the Common Lisp `ceiling` function, which is analogous to `floor` except that it rounds the argument or quotient of the arguments up toward plus infinity. The remainder will be between 0 and minus r.

Function: **truncate*** *number &optional divisor*

This function implements the Common Lisp `truncate` function, which is analogous to `floor` except that it rounds the argument or quotient of the arguments toward zero. Thus it is equivalent to `floor*` if the argument or quotient is positive, or to `ceiling*` otherwise. The remainder has the same sign as number.

Function: **round*** *number &optional divisor*

This function implements the Common Lisp `round` function, which is analogous to `floor` except that it rounds the argument or quotient of the arguments to the nearest integer. In the case of a tie (the argument or quotient is exactly halfway between two integers), it rounds to the even integer.

Function: **mod*** *number divisor*

This function returns the same value as the second return value of `floor`.

Function: **rem*** *number divisor*

This function returns the same value as the second return value of `truncate`.

These definitions are compatible with those in the Quiroz `` `cl.el' `` package, except that this package appends `` `*' `` to certain function names to avoid conflicts with existing Emacs 19 functions, and that the mechanism for returning multiple values is different.

@secno=8

Random Numbers

This package also provides an implementation of the Common Lisp random number generator. It uses its own additive-congruential algorithm, which is much more likely to give statistically clean random numbers than the simple generators supplied by many operating systems.

Function: **random*** *number &optional state*

This function returns a random nonnegative number less than number, and of the same type (either integer or floating-point). The state argument should be a `random-state` object which holds the state of the random number generator. The function modifies this state object as a side effect. If state is omitted, it defaults to the variable `*random-state*`, which contains a pre-initialized `random-state` object.

Variable: ***random-state***

This variable contains the system "default" `random-state` object, used for calls to `random*` that do not specify an alternative state object. Since any number of programs in the Emacs process may be accessing `*random-state*` in interleaved fashion, the sequence generated from this variable will be irreproducible for all intents and purposes.

Function: **make-random-state** *&optional state*

This function creates or copies a `random-state` object. If state is omitted or `nil`, it returns a new copy of `*random-state*`. This is a copy in the sense that future sequences of calls to `(random* n)` and `(random* n s)` (where s is the new random-state object) will return identical sequences of random numbers.

If state is a `random-state` object, this function returns a copy of that object. If state is `t`, this function returns a new `random-state` object seeded from the date and time. As an extension to Common Lisp, state may also be an integer in which case the new object is seeded from that integer; each different integer seed will result in a completely different sequence of random numbers.

It is legal to print a `random-state` object to a buffer or file and later read it back with `read`. If a program wishes to use a sequence of pseudo-random numbers which can be reproduced later for debugging, it can call `(make-random-state t)` to get a new sequence, then print this sequence to a file. When the program is later rerun, it can read the original run's random-state from the file.

Function: **random-state-p** *object*

This predicate returns `t` if object is a `random-state` object, or `nil` otherwise.

Implementation Parameters

This package defines several useful constants having to with numbers.

Variable: **most-positive-fixnum**

This constant equals the largest value a Lisp integer can hold. It is typically `2^23-1` or `2^25-1`.

Variable: **most-negative-fixnum**

This constant equals the smallest (most negative) value a Lisp integer can hold.

The following parameters have to do with floating-point numbers. This package determines their values by exercising the computer's floating-point arithmetic in various ways. Because this operation might be slow, the code for initializing them is kept in a separate function that must be called before the parameters can be used.

Function: **cl-float-limits**

This function makes sure that the Common Lisp floating-point parameters like `most-positive-float` have been initialized. Until it is called, these parameters will be `nil`. If this version of Emacs does not support floats (e.g., most versions of Emacs 18), the parameters will remain `nil`. If the parameters have already been initialized, the function returns immediately.

The algorithm makes assumptions that will be valid for most modern machines, but will fail if the machine's arithmetic is extremely unusual, e.g., decimal.

Since true Common Lisp supports up to four different floating-point precisions, it has families of constants like `most-positive-single-float`, `most-positive-double-float`, `most-positive-long-float`, and so on. Emacs has only one floating-point precision, so this package omits the precision word from the constants' names.

Variable: **most-positive-float**

This constant equals the largest value a Lisp float can hold. For those systems whose arithmetic supports infinities, this is the largest *finite* value. For IEEE machines, the value is approximately `1.79e+308`.

Variable: **most-negative-float**

This constant equals the most-negative value a Lisp float can hold. (It is assumed to be equal to `(-most-positive-float)`.)

Variable: **least-positive-float**

This constant equals the smallest Lisp float value greater than zero. For IEEE machines, it is about `4.94e-324` if denormals are supported or `2.22e-308` if not.

Variable: **least-positive-normalized-float**

This constant equals the smallest *normalized* Lisp float greater than zero, i.e., the smallest value for which IEEE denormalization will not result in a loss of precision. For IEEE machines, this value is about `2.22e-308`. For machines that do not support the concept of denormalization and gradual underflow, this constant will always equal `least-positive-float`.

Variable: **least-negative-float**

This constant is the negative counterpart of `least-positive-float`.

Variable: **least-negative-normalized-float**

This constant is the negative counterpart of `least-positive-normalized-float`.

Variable: **float-epsilon**

This constant is the smallest positive Lisp float that can be added to 1.0 to produce a distinct value. Adding a smaller number to 1.0 will yield 1.0 again due to roundoff. For IEEE machines, epsilon is about `2.22e-16`.

Variable: **float-negative-epsilon**

This is the smallest positive value that can be subtracted from 1.0 to produce a distinct value. For IEEE machines, it is about `1.11e-16`.

@chapno=13

Go to the previous, next section.

Go to the previous, next section.

Sequences

Common Lisp defines a number of functions that operate on sequences, which are either lists, strings, or vectors. Emacs Lisp includes a few of these, notably `elt` and `length`; this package defines most of the rest.

Sequence Basics

Many of the sequence functions take keyword arguments; see section Argument Lists. All keyword arguments are optional and, if specified, may appear in any order.

The `:key` argument should be passed either `nil`, or a function of one argument. This key function is used as a filter through which the elements of the sequence are seen; for example, `(find x y :key 'car)` is similar to `(assoc* x y)`: It searches for an element of the list whose `car` equals `x`, rather than for an element which equals `x` itself. If `:key` is omitted or `nil`, the filter is effectively the identity function.

The `:test` and `:test-not` arguments should be either `nil`, or functions of two arguments. The test function is used to compare two sequence elements, or to compare a search value with sequence elements. (The two values are passed to the test function in the same order as the original sequence function arguments from which they are derived, or, if they both come from the same sequence, in the same order as they appear in that sequence.) The `:test` argument specifies a function which must return true (non-`nil`) to indicate a match; instead, you may use `:test-not` to give a function which returns *false* to indicate a match. The default test function is `:test 'eql`.

Many functions which take item and `:test` or `:test-not` arguments also come in `-if` and `-if-not` varieties, where a predicate function is passed instead of item, and sequence elements match if the predicate returns true on them (or false in the case of `-if-not`). For example:

```
(remove* 0 seq :test '=)  ==  (remove-if 'zerop seq)
```

to remove all zeros from sequence `seq`.

Some operations can work on a subsequence of the argument sequence; these function take `:start` and `:end` arguments which default to zero and the length of the sequence, respectively. Only elements between start (inclusive) and end (exclusive) are affected by the operation. The end argument may be passed `nil` to signify the length of the sequence; otherwise, both start and end must be integers, with `0 <= start <= end <= (length seq)`. If the function takes two sequence arguments, the limits are defined by keywords `:start1` and `:end1` for the first, and `:start2` and `:end2` for the second.

A few functions accept a `:from-end` argument, which, if non-`nil`, causes the operation to go from right-to-left through the sequence instead of left-to-right, and a `:count` argument, which specifies an integer maximum number of elements to be removed or otherwise processed.

The sequence functions make no guarantees about the order in which the `:test`, `:test-not`, and `:key` functions are called on various elements. Therefore, it is a bad idea to depend on side effects of these functions. For example, `:from-end` may cause the sequence to be scanned actually in reverse, or it may be scanned forwards but computing a result "as if" it were scanned backwards. (Some functions, like `mapcar*` and `every`, *do* specify exactly the order in which the function is called so side effects are perfectly acceptable in those cases.)

Strings in GNU Emacs 19 may contain "text properties" as well as character data. Except as noted, it is undefined whether or not text properties are preserved by sequence functions. For example, `(remove* ?A str)` may or may not preserve the properties of the characters copied from str into the result.

Mapping over Sequences

These functions "map" the function you specify over the elements of lists or arrays. They are all variations on the theme of the built-in function `mapcar`.

Function: **mapcar*** *function seq &rest more-seqs*

This function calls function on successive parallel sets of elements from its argument sequences. Given a single seq argument it is equivalent to `mapcar`; given n sequences, it calls the function with the first elements of each of the sequences as the n arguments to yield the first element of the result list, then with the second elements, and so on. The mapping stops as soon as the shortest sequence runs out. The argument sequences may be any mixture of lists, strings, and vectors; the return sequence is always a list.

Common Lisp's `mapcar` accepts multiple arguments but works only on lists; Emacs Lisp's `mapcar` accepts a single sequence argument. This package's `mapcar*` works as a compatible superset of both.

Function: **map** *result-type function seq &rest more-seqs*

This function maps function over the argument sequences, just like `mapcar*`, but it returns a sequence of type result-type rather than a list. result-type must be one of the following symbols: `vector`, `string`, `list` (in which case the effect is the same as for `mapcar*`), or `nil` (in which case the results are thrown away and `map` returns `nil`).

Function: **maplist** *function list &rest more-lists*

This function calls function on each of its argument lists, then on the `cdr`s of those lists, and so on, until the shortest list runs out. The results are returned in the form of a list. Thus, `maplist` is like `mapcar*` except that it passes in the list pointers themselves rather than the `car`s of the advancing pointers.

Function: **mapc** *function seq &rest more-seqs*

This function is like `mapcar*`, except that the values returned by function are ignored and thrown away rather than being collected into a list. The return value of `mapc` is seq, the first sequence.

Function: **mapl** *function list &rest more-lists*

This function is like `maplist`, except that it throws away the values returned by function.

Function: **mapcan** *function seq &rest more-seqs*

This function is like `mapcar*`, except that it concatenates the return values (which must be lists) using `nconc`, rather than simply collecting them into a list.

Function: **mapcon** *function list &rest more-lists*

This function is like `maplist`, except that it concatenates the return values using `nconc`.

Function: **some** *predicate seq &rest more-seqs*

This function calls predicate on each element of seq in turn; if predicate returns a non-`nil` value, `some` returns that value, otherwise it returns `nil`. Given several sequence arguments, it steps through the sequences in parallel until the shortest one runs out, just as in `mapcar*`. You can rely on the left-to-right order in which the elements are visited, and on the fact that mapping stops immediately as soon as predicate returns non-`nil`.

Function: **every** *predicate seq &rest more-seqs*

This function calls predicate on each element of the sequence(s) in turn; it returns `nil` as soon as predicate returns `nil` for any element, or `t` if the predicate was true for all elements.

Function: **notany** *predicate seq &rest more-seqs*

This function calls predicate on each element of the sequence(s) in turn; it returns `nil` as soon as predicate returns a non-`nil` value for any element, or `t` if the predicate was `nil` for all elements.

Function: **notevery** *predicate seq &rest more-seqs*

This function calls predicate on each element of the sequence(s) in turn; it returns a non-`nil` value as soon as predicate returns `nil` for any element, or `t` if the predicate was true for all elements.

Function: **reduce** *function seq* `&key :from-end :start :end :initial-value :key`

This function combines the elements of seq using an associative binary operation. Suppose function is `*` and seq is the list `(2 3 4 5)`. The first two elements of the list are combined with `(* 2 3) = 6`; this is combined with the next element, `(* 6 4) = 24`, and that is combined with the final element: `(* 24 5) = 120`. Note that the `*` function happens to be self-reducing, so that `(* 2 3 4 5)` has the same effect as an explicit call to `reduce`.

If `:from-end` is true, the reduction is right-associative instead of left-associative:

```
(reduce '- '(1 2 3 4))
     == (- (- (- 1 2) 3) 4) => -8
(reduce '- '(1 2 3 4) :from-end t)
     == (- 1 (- 2 (- 3 4))) => -2
```

If `:key` is specified, it is a function of one argument which is called on each of the sequence elements in turn.

If `:initial-value` is specified, it is effectively added to the front (or rear in the case of

`:from-end`) of the sequence. The `:key` function is *not* applied to the initial value.

If the sequence, including the initial value, has exactly one element then that element is returned without ever calling function. If the sequence is empty (and there is no initial value), then function is called with no arguments to obtain the return value.

All of these mapping operations can be expressed conveniently in terms of the `loop` macro. In compiled code, `loop` will be faster since it generates the loop as in-line code with no function calls.

Sequence Functions

This section describes a number of Common Lisp functions for operating on sequences.

Function: **subseq** *sequence start &optional end*

This function returns a given subsequence of the argument sequence, which may be a list, string, or vector. The indices start and end must be in range, and start must be no greater than end. If end is omitted, it defaults to the length of the sequence. The return value is always a copy; it does not share structure with sequence.

As an extension to Common Lisp, start and/or end may be negative, in which case they represent a distance back from the end of the sequence. This is for compatibility with Emacs' `substring` function. Note that `subseq` is the *only* sequence function that allows negative start and end.

You can use `setf` on a `subseq` form to replace a specified range of elements with elements from another sequence. The replacement is done as if by `replace`, described below.

Function: **concatenate** *result-type &rest seqs*

This function concatenates the argument sequences together to form a result sequence of type result-type, one of the symbols `vector`, `string`, or `list`. The arguments are always copied, even in cases such as `(concatenate 'list '(1 2 3))` where the result is identical to an argument.

Function: **fill** *seq item* `&key :start :end`

This function fills the elements of the sequence (or the specified part of the sequence) with the value item.

Function: **replace** *seq1 seq2* `&key :start1 :end1 :start2 :end2`

This function copies part of seq2 into part of seq1. The sequence seq1 is not stretched or resized; the amount of data copied is simply the shorter of the source and destination (sub)sequences. The function returns seq1.

If seq1 and seq2 are `eq`, then the replacement will work correctly even if the regions indicated by the start and end arguments overlap. However, if seq1 and seq2 are lists which share storage but are not `eq`, and the start and end arguments specify overlapping regions, the effect is undefined.

Function: **remove*** *item seq* `&key :test :test-not :key :count :start :end :from-end`

This returns a copy of seq with all elements matching item removed. The result may share storage with or be `eq` to seq in some circumstances, but the original seq will not be modified. The `:test`, `:test-not`, and `:key` arguments define the matching test that is used; by default, elements `eql` to item are removed. The `:count` argument specifies the maximum number of matching elements that can be removed (only the leftmost count matches are removed). The `:start` and `:end` arguments specify a region in seq in which elements will be removed; elements outside that region are not matched or removed. The `:from-end` argument, if true, says that elements should be deleted from the end of the sequence rather than the beginning (this matters only if count was also specified).

Function: **delete*** *item seq* `&key :test :test-not :key :count :start :end :from-end`

This deletes all elements of seq which match item. It is a destructive operation. Since Emacs Lisp does not support stretchable strings or vectors, this is the same as `remove*` for those sequence types. On lists, `remove*` will copy the list if necessary to preserve the original list, whereas `delete*` will splice out parts of the argument list. Compare `append` and `nconc`, which are analogous non-destructive and destructive list operations in Emacs Lisp.

The predicate-oriented functions `remove-if`, `remove-if-not`, `delete-if`, and `delete-if-not` are defined similarly.

Function: **delete** *item list*

This MacLisp-compatible function deletes from list all elements which are `equal` to item. The `delete` function is built-in to Emacs 19; this package defines it equivalently in Emacs 18.

Function: **remove** *item list*

This function removes from list all elements which are `equal` to item. This package defines it for symmetry with `delete`, even though `remove` is not built-in to Emacs 19.

Function: **remq** *item list*

This function removes from list all elements which are `eq` to item. This package defines it for symmetry with `delq`, even though `remq` is not built-in to Emacs 19.

Function: **remove-duplicates** *seq* `&key :test :test-not :key :start :end :from-end`

This function returns a copy of seq with duplicate elements removed. Specifically, if two elements from the sequence match according to the `:test`, `:test-not`, and `:key` arguments, only the rightmost one is retained. If `:from-end` is true, the leftmost one is retained instead. If `:start` or `:end` is specified, only elements within that subsequence are examined or removed.

Function: **delete-duplicates** *seq* `&key :test :test-not :key :start :end :from-end`

This function deletes duplicate elements from seq. It is a destructive version of `remove-duplicates`.

Function: **substitute** *new old seq* `&key :test :test-not :key :count :start :end :from-end`

This function returns a copy of seq, with all elements matching old replaced with new. The `:count`,

`:start`, `:end`, and `:from-end` arguments may be used to limit the number of substitutions made.

Function: **nsubstitute** *new old seq* `&key :test :test-not :key :count :start :end :from-end`

This is a destructive version of `substitute`; it performs the substitution using `setcar` or `aset` rather than by returning a changed copy of the sequence.

The `substitute-if`, `substitute-if-not`, `nsubstitute-if`, and `nsubstitute-if-not` functions are defined similarly. For these, a predicate is given in place of the old argument.

Searching Sequences

These functions search for elements or subsequences in a sequence. (See also `member*` and `assoc*`; see section Lists.)

Function: **find** *item seq* `&key :test :test-not :key :start :end :from-end`

This function searches seq for an element matching item. If it finds a match, it returns the matching element. Otherwise, it returns `nil`. It returns the leftmost match, unless `:from-end` is true, in which case it returns the rightmost match. The `:start` and `:end` arguments may be used to limit the range of elements that are searched.

Function: **position** *item seq* `&key :test :test-not :key :start :end :from-end`

This function is like `find`, except that it returns the integer position in the sequence of the matching item rather than the item itself. The position is relative to the start of the sequence as a whole, even if `:start` is non-zero. The function returns `nil` if no matching element was found.

Function: **count** *item seq* `&key :test :test-not :key :start :end`

This function returns the number of elements of seq which match item. The result is always a nonnegative integer.

The `find-if`, `find-if-not`, `position-if`, `position-if-not`, `count-if`, and `count-if-not` functions are defined similarly.

Function: **mismatch** *seq1 seq2* `&key :test :test-not :key :start1 :end1 :start2 :end2 :from-end`

This function compares the specified parts of seq1 and seq2. If they are the same length and the corresponding elements match (according to `:test`, `:test-not`, and `:key`), the function returns `nil`. If there is a mismatch, the function returns the index (relative to seq1) of the first mismatching element. This will be the leftmost pair of elements which do not match, or the position at which the shorter of the two otherwise-matching sequences runs out.

If `:from-end` is true, then the elements are compared from right to left starting at `(1- end1)` and `(1- end2)`. If the sequences differ, then one plus the index of the rightmost difference (relative to seq1) is returned.

An interesting example is `(mismatch str1 str2 :key 'upcase)`, which compares two str case-insensitively.

Function: **search** *seq1 seq2* `&key :test :test-not :key :from-end :start1 :end1 :start2 :end2`

This function searches seq2 for a subsequence that matches seq1 (or part of it specified by `:start1` and `:end1`.) Only matches which fall entirely within the region defined by `:start2` and `:end2` will be considered. The return value is the index of the leftmost element of the leftmost match, relative to the start of seq2, or `nil` if no matches were found. If `:from-end` is true, the function finds the *rightmost* matching subsequence.

Sorting Sequences

Function: **sort*** *seq predicate* `&key :key`

This function sorts seq into increasing order as determined by using predicate to compare pairs of elements. predicate should return true (non-`nil`) if and only if its first argument is less than (not equal to) its second argument. For example, `<` and `string-lessp` are suitable predicate functions for sorting numbers and strings, respectively; `>` would sort numbers into decreasing rather than increasing order.

This function differs from Emacs' built-in `sort` in that it can operate on any type of sequence, not just lists. Also, it accepts a `:key` argument which is used to preprocess data fed to the predicate function. For example,

```
(setq data (sort data 'string-lessp :key 'downcase))
```

sorts data, a sequence of strings, into increasing alphabetical order without regard to case. A `:key` function of `car` would be useful for sorting association lists.

The `sort*` function is destructive; it sorts lists by actually rearranging the `cdr` pointers in suitable fashion.

Function: **stable-sort** *seq predicate* `&key :key`

This function sorts seq stably, meaning two elements which are equal in terms of predicate are guaranteed not to be rearranged out of their original order by the sort.

In practice, `sort*` and `stable-sort` are equivalent in Emacs Lisp because the underlying `sort` function is stable by default. However, this package reserves the right to use non-stable methods for `sort*` in the future.

Function: **merge** *type seq1 seq2 predicate* `&key :key`

This function merges two sequences seq1 and seq2 by interleaving their elements. The result sequence, of type type (in the sense of `concatenate`), has length equal to the sum of the lengths of the two input sequences. The sequences may be modified destructively. Order of elements within seq1 and seq2 is preserved in the interleaving; elements of the two sequences are compared by predicate (in the sense of

`sort`) and the lesser element goes first in the result. When elements are equal, those from seq1 prec those from seq2 in the result. Thus, if seq1 and seq2 are both sorted according to predicate, then the result will be a merged sequence which is (stably) sorted according to predicate.

Go to the previous, next section.

Go to the previous, next section.

Lists

The functions described here operate on lists.

List Functions

This section describes a number of simple operations on lists, i.e., chains of cons cells.

Function: **caddr** *x*

This function is equivalent to `(car (cdr (cdr x)))`. Likewise, this package defines all 28 `cxxxr` functions where xxx is up to four `a's and/or `d's. All of these functions are `setf`-able, and calls to them are expanded inline by the byte-compiler for maximum efficiency.

Function: **first** *x*

This function is a synonym for `(car x)`. Likewise, the functions `second`, `third`, ..., through `tenth` return the given element of the list x.

Function: **rest** *x*

This function is a synonym for `(cdr x)`.

Function: **endp** *x*

Common Lisp defines this function to act like `null`, but signaling an error if `x` is neither a `nil` nor a cons cell. This package simply defines `endp` as a synonym for `null`.

Function: **list-length** *x*

This function returns the length of list x, exactly like `(length x)`, except that if x is a circular list (where the cdr-chain forms a loop rather than terminating with `nil`), this function returns `nil`. (The regular `length` function would get stuck if given a circular list.)

Function: **last** *x &optional n*

This function returns the last cons, or the nth-to-last cons, of the list x. If n is omitted it defaults to 1. The "last cons" means the first cons cell of the list whose `cdr` is not another cons cell. (For normal lists, the `cdr` of the last cons will be `nil`.) This function returns `nil` if x is `nil` or shorter than n. Note that the last *element* of the list is `(car (last x))`.

Function: **butlast** *x &optional n*

This function returns the list x with the last element, or the last n elements, removed. If n is greater than zero it makes a copy of the list so as not to damage the original list. In general, `(append (butlast x n) (last x n))` will return a list equal to x.

Function: **nbutlast** *x &optional n*

This is a version of `butlast` that works by destructively modifying the `cdr` of the appropriate element, rather than making a copy of the list.

Function: **list*** *arg &rest others*

This function constructs a list of its arguments. The final argument becomes the `cdr` of the last cell constructed. Thus, `(list* a b c)` is equivalent to `(cons a (cons b c))`, and `(list* a b nil)` is equivalent to `(list a b)`.

(Note that this function really is called `list*` in Common Lisp; it is not a name invented for this package like `member*` or `defun*`.)

Function: **ldiff** *list sublist*

If sublist is a sublist of list, i.e., is `eq` to one of the cons cells of list, then this function returns a copy of the part of list up to but not including sublist. For example, `(ldiff x (cddr x))` returns the first two elements of the list `x`. The result is a copy; the original list is not modified. If sublist is not a sublist of list, a copy of the entire list is returned.

Function: **copy-list** *list*

This function returns a copy of the list list. It copies dotted lists like `(1 2 . 3)` correctly.

Function: **copy-tree** *x &optional vecp*

This function returns a copy of the tree of cons cells x. Unlike `copy-sequence` (and its alias `copy-list`), which copies only along the `cdr` direction, this function copies (recursively) along both the `car` and the `cdr` directions. If x is not a cons cell, the function simply returns x unchanged. If the optional vecp argument is true, this function copies vectors (recursively) as well as cons cells.

Function: **tree-equal** *x y* `&key :test :test-not :key`

This function compares two trees of cons cells. If x and y are both cons cells, their `car`s and `cdr`s are compared recursively. If neither x nor y is a cons cell, they are compared by `eql`, or according to the specified test. The `:key` function, if specified, is applied to the elements of both trees. See section Sequences.

@secno=3

Substitution of Expressions

These functions substitute elements throughout a tree of cons cells. (See section Sequence Functions, for the `substitute` function, which works on just the top-level elements of a list.)

Function: **subst** *new old tree* `&key :test :test-not :key`

This function substitutes occurrences of old with new in tree, a tree of cons cells. It returns a substituted tree, which will be a copy except that it may share storage with the argument tree in parts where no

substitutions occurred. The original tree is not modified. This function recurses on, and compares ag
old, both `car`s and `cdr`s of the component cons cells. If old is itself a cons cell, then matching cells
the tree are substituted as usual without recursively substituting in that cell. Comparisons with old a
done according to the specified test (`eql` by default). The `:key` function is applied to the elements of the tree but not to old.

Function: **nsubst** *new old tree* `&key :test :test-not :key`

This function is like `subst`, except that it works by destructive modification (by `setcar` or `setcdr`) rather than copying.

The `subst-if`, `subst-if-not`, `nsubst-if`, and `nsubst-if-not` functions are defined similarly.

Function: **sublis** *alist tree* `&key :test :test-not :key`

This function is like `subst`, except that it takes an association list alist of old-new pairs. Each element of the tree (after applying the `:key` function, if any), is compared with the `car`s of alist; if it matches, it is replaced by the corresponding `cdr`.

Function: **nsublis** *alist tree* `&key :test :test-not :key`

This is a destructive version of `sublis`.

Lists as Sets

These functions perform operations on lists which represent sets of elements.

Function: **member** *item list*

This MacLisp-compatible function searches list for an element which is `equal` to item. The `member` function is built-in to Emacs 19; this package defines it equivalently in Emacs 18. See the following function for a Common-Lisp compatible version.

Function: **member*** *item list* `&key :test :test-not :key`

This function searches list for an element matching item. If a match is found, it returns the cons cell whose `car` was the matching element. Otherwise, it returns `nil`. Elements are compared by `eql` by default; you can use the `:test`, `:test-not`, and `:key` arguments to modify this behavior. See section Sequences.

Note that this function's name is suffixed by `*' to avoid the incompatible `member` function defined in Emacs 19. (That function uses `equal` for comparisons; it is equivalent to `(member* item list :test 'equal)`.)

The `member-if` and `member-if-not` functions analogously search for elements which satisfy a given predicate.

Function: **tailp** *sublist list*

This function returns `t` if sublist is a sublist of list, i.e., if sublist is `eql` to list or to any of its `cdr`s.

Function: **adjoin** *item list* `&key :test :test-not :key`

This function conses item onto the front of list, like `(cons item list)`, but only if item is not already present on the list (as determined by `member*`). If a `:key` argument is specified, it is applied to item as well as to the elements of list during the search, on the reasoning that item is "about" to become part of the list.

Function: **union** *list1 list2* `&key :test :test-not :key`

This function combines two lists which represent sets of items, returning a list that represents the union of those two sets. The result list will contain all items which appear in list1 or list2, and no others. If an item appears in both list1 and list2 it will be copied only once. If an item is duplicated in list1 or list2, it is undefined whether or not that duplication will survive in the result list. The order of elements in the result list is also undefined.

Function: **nunion** *list1 list2* `&key :test :test-not :key`

This is a destructive version of `union`; rather than copying, it tries to reuse the storage of the argument lists if possible.

Function: **intersection** *list1 list2* `&key :test :test-not :key`

This function computes the intersection of the sets represented by list1 and list2. It returns the list of items which appear in both list1 and list2.

Function: **nintersection** *list1 list2* `&key :test :test-not :key`

This is a destructive version of `intersection`. It tries to reuse storage of list1 rather than copying. It does *not* reuse the storage of list2.

Function: **set-difference** *list1 list2* `&key :test :test-not :key`

This function computes the "set difference" of list1 and list2, i.e., the set of elements that appear in list1 but *not* in list2.

Function: **nset-difference** *list1 list2* `&key :test :test-not :key`

This is a destructive `set-difference`, which will try to reuse list1 if possible.

Function: **set-exclusive-or** *list1 list2* `&key :test :test-not :key`

This function computes the "set exclusive or" of list1 and list2, i.e., the set of elements that appear in exactly one of list1 and list2.

Function: **nset-exclusive-or** *list1 list2* `&key :test :test-not :key`

This is a destructive `set-exclusive-or`, which will try to reuse list1 and list2 if possible.

Function: **subsetp** *list1 list2* `&key :test :test-not :key`

This function checks whether list1 represents a subset of list2, i.e., whether every element of list1 also

appears in list2.

Association Lists

An association list is a list representing a mapping from one set of values to another; any list whose elements are cons cells is an association list.

Function: **assoc*** *item a-list* `&key :test :test-not :key`

This function searches the association list a-list for an element whose `car` matches (in the sense of `:test`, `:test-not`, and `:key`, or by comparison with `eql`) a given item. It returns the matching element, if any, otherwise `nil`. It ignores elements of a-list which are not cons cells. (This corresponds to the behavior of `assq` and `assoc` in Emacs Lisp; Common Lisp's `assoc` ignores `nil`s but considers any other non-cons elements of a-list to be an error.)

Function: **rassoc*** *item a-list* `&key :test :test-not :key`

This function searches for an element whose `cdr` matches item. If a-list represents a mapping, this applies the inverse of the mapping to item.

Function: **rassoc** *item a-list*

This function searches like `rassoc*` with a `:test` argument of `equal`. It is analogous to Emacs Lisp's standard `assoc` function, which derives from the MacLisp rather than the Common Lisp tradition.

The `assoc-if`, `assoc-if-not`, `rassoc-if`, and `rassoc-if-not` functions are defined similarly.

Two simple functions for constructing association lists are:

Function: **acons** *key value alist*

This is equivalent to `(cons (cons key value) alist)`.

Function: **pairlis** *keys values &optional alist*

This is equivalent to `(nconc (mapcar* 'cons keys values) alist)`.

Go to the previous, next section.

Go to the previous, next section.

Hash Tables

A hash table is a data structure that maps "keys" onto "values." Keys and values can be arbitrary Lisp data objects. Hash tables have the property that the time to search for a given key is roughly constant; simpler data structures like association lists take time proportional to the number of entries in the list.

Function: **make-hash-table** *&key :test :size*

This function creates and returns a hash-table object whose function for comparing elements is `:test` (`eql` by default), and which is allocated to fit about `:size` elements. The `:size` argument is purely advisory; the table will stretch automatically if you store more elements in it. If `:size` is omitted, a reasonable default is used.

Common Lisp allows only `eq`, `eql`, `equal`, and `equalp` as legal values for the `:test` argument. In this package, any reasonable predicate function will work, though if you use something else you should check the details of the hashing function described below to make sure it is suitable for your predicate.

Some versions of Emacs (like Lucid Emacs 19) include a built-in hash table type; in these versions, `make-hash-table` with a test of `eq` will use these built-in hash tables. In all other cases, it will return a hash-table object which takes the form of a list with an identifying "tag" symbol at the front. All of the hash table functions in this package can operate on both types of hash table; normally you will never know which type is being used.

This function accepts the additional Common Lisp keywords `:rehash-size` and `:rehash-threshold`, but it ignores their values.

Function: **gethash** *key table &optional default*

This function looks up key in table. If key exists in the table, in the sense that it matches any of the existing keys according to the table's test function, then the associated value is returned. Otherwise, default (or `nil`) is returned.

To store new data in the hash table, use `setf` on a call to `gethash`. If key already exists in the table, the corresponding value is changed to the stored value. If key does not already exist, a new entry is added to the table and the table is reallocated to a larger size if necessary. The default argument is allowed but ignored in this case. The situation is exactly analogous to that of `get*`; see section Property Lists.

Function: **remhash** *key table*

This function removes the entry for key from table. If an entry was removed, it returns `t`. If key does not appear in the table, it does nothing and returns `nil`.

Function: **clrhash** *table*

This function removes all the entries from table, leaving an empty hash table.

Function: **maphash** *function table*

This function calls function for each entry in table. It passes two arguments to function, the key and the value of the given entry. The return value of function is ignored; maphash itself returns `nil`. See section Loop Facility, for an alternate way of iterating over hash tables.

Function: **hash-table-count** *table*

This function returns the number of entries in table. **Warning:** The current implementation of Lucid Emacs 19 hash-tables does not decrement the stored `count` when `remhash` removes an entry. Therefore, the return value of this function is not dependable if you have used `remhash` on the table and the table's test is `eq`. A slower, but reliable, way to count the entries is `(loop for x being the hash-keys of table count t)`.

Function: **hash-table-p** *object*

This function returns `t` if object is a hash table, `nil` otherwise. It recognizes both types of hash tables (both Lucid Emacs built-in tables and tables implemented with special lists.)

Sometimes when dealing with hash tables it is useful to know the exact "hash function" that is used. This package implements hash tables using Emacs Lisp "obarrays," which are the same data structure that Emacs Lisp uses to keep track of symbols. Each hash table includes an embedded obarray. Key values given to `gethash` are converted by various means into strings, which are then looked up in the obarray using `intern` and `intern-soft`. The symbol, or "bucket," corresponding to a given key string includes as its `symbol-value` an association list of all key-value pairs which hash to that string. Depending on the test function, it is possible for many entries to hash to the same bucket. For example, if the test is `eql`, then the symbol `foo` and two separately built strings `"foo"` will create three entries in the same bucket. Search time is linear within buckets, so hash tables will be most effective if you arrange not to store too many things that hash the same.

The following algorithm is used to convert Lisp objects to hash strings:

- Strings are used directly as hash strings. (However, if the test function is `equalp`, strings are `downcase`d first.)
- Symbols are hashed according to their `symbol-name`.
- Integers are hashed into one of 16 buckets depending on their value modulo 16. Floating-point numbers are truncated to integers and hashed modulo 16.
- Cons cells are hashed according to their `car`s; nonempty vectors are hashed according to their first element.
- All other types of objects hash into a single bucket named `"*"`.

Thus, for example, searching among many buffer objects in a hash table will devolve to a (still fairly fast) linear-time search through a single bucket, whereas searching for different symbols will be very fast since each symbol will, in general, hash into its own bucket.

The size of the obarray in a hash table is automatically adjusted as the number of elements increases.

As a special case, `make-hash-table` with a `:size` argument of 0 or 1 will create a hash-table object that uses a single association list rather than an obarray of many lists. For very small tables this structure will be more efficient since lookup does not require converting the key to a string or looking it up in an

TOC

obarray. However, such tables are guaranteed to take time proportional to their size to do a search.

@chapno=18

Go to the previous, next section.

Go to the previous, next section.

Structures

The Common Lisp structure mechanism provides a general way to define data types similar to C's `struct` types. A structure is a Lisp object containing some number of slots, each of which can hold any Lisp data object. Functions are provided for accessing and setting the slots, creating or copying structure objects, and recognizing objects of a particular structure type.

In true Common Lisp, each structure type is a new type distinct from all existing Lisp types. Since the underlying Emacs Lisp system provides no way to create new distinct types, this package implements structures as vectors (or lists upon request) with a special "tag" symbol to identify them.

Special Form: **defstruct** *name slots...*

The `defstruct` form defines a new structure type called name, with the specified slots. (The slots may begin with a string which documents the structure type.) In the simplest case, name and each of the slots are symbols. For example,

```
(defstruct person name age sex)
```

defines a struct type called `person` which contains three slots. Given a `person` object p, you can access those slots by calling `(person-name p)`, `(person-age p)`, and `(person-sex p)`. You can also change these slots by using `setf` on any of these place forms:

```
(incf (person-age birthday-boy))
```

You can create a new `person` by calling `make-person`, which takes keyword arguments `:name`, `:age`, and `:sex` to specify the initial values of these slots in the new object. (Omitting any of these arguments leaves the corresponding slot "undefined," according to the Common Lisp standard; in Emacs Lisp, such uninitialized slots are filled with `nil`.)

Given a `person`, `(copy-person p)` makes a new object of the same type whose slots are `eq` to those of p.

Given any Lisp object x, `(person-p x)` returns true if x looks like a `person`, false otherwise. (Again, in Common Lisp this predicate would be exact; in Emacs Lisp the best it can do is verify that x is a vector of the correct length which starts with the correct tag symbol.)

Accessors like `person-name` normally check their arguments (effectively using `person-p`) and signal an error if the argument is the wrong type. This check is affected by `(optimize (safety ...))` declarations. Safety level 1, the default, uses a somewhat optimized check that will detect all incorrect arguments, but may use an uninformative error message (e.g., "expected a vector" instead of "expected a `person`"). Safety level 0 omits all checks except as provided by the underlying `aref` call; safety levels 2 and 3 do rigorous checking that will always print a descriptive error message for incorrect inputs. See section Declarations.


```
(setq dave (make-person :name "Dave" :sex 'male))
     => [cl-struct-person "Dave" nil male]
(setq other (copy-person dave))
     => [cl-struct-person "Dave" nil male]
(eq dave other)
     => nil
(eq (person-name dave) (person-name other))
     => t
(person-p dave)
     => t
(person-p [1 2 3 4])
     => nil
(person-p "Bogus")
     => nil
(person-p '[cl-struct-person counterfeit person object])
     => t
```

In general, name is either a name symbol or a list of a name symbol followed by any number of struct options; each slot is either a slot symbol or a list of the form `(slot-name default-value slot-options...)'. The default-value is a Lisp form which is evaluated any time an instance of the structure type is created without specifying that slot's value.

Common Lisp defines several slot options, but the only one implemented in this package is `:read-only`. A non-`nil` value for this option means the slot should not be `setf`-able; the slot's value is determined when the object is created and does not change afterward.

```
(defstruct person
  (name nil :read-only t)
  age
  (sex 'unknown))
```

Any slot options other than `:read-only` are ignored.

For obscure historical reasons, structure options take a different form than slot options. A structure option is either a keyword symbol, or a list beginning with a keyword symbol possibly followed by arguments. (By contrast, slot options are key-value pairs not enclosed in lists.)

```
(defstruct (person (:constructor create-person)
                   (:type list)
                   :named)
  name age sex)
```

The following structure options are recognized.

@itemmax=0 in @advance@leftskip-.5@tableindent

`:conc-name`

The argument is a symbol whose print name is used as the prefix for the names of slot accesso functions. The default is the name of the struct type followed by a hyphen. The option `(:conc-name p-)` would change this prefix to `p-`. Specifying `nil` as an argument means no prefix, so that the slot names themselves are used to name the accessor functions.

`:constructor`

In the simple case, this option takes one argument which is an alternate name to use for the constructor function. The default is `make-name`, e.g., `make-person`. The above example changes this to `create-person`. Specifying `nil` as an argument means that no standard constructor should be generated at all.

In the full form of this option, the constructor name is followed by an arbitrary argument list. See section Program Structure, for a description of the format of Common Lisp argument lists. All options, such as `&rest` and `&key`, are supported. The argument names should match the slot names; each slot is initialized from the corresponding argument. Slots whose names do not appear in the argument list are initialized based on the default-value in their slot descriptor. Also, `&optional` and `&key` arguments which don't specify defaults take their defaults from the slot descriptor. It is legal to include arguments which don't correspond to slot names; these are useful if they are referred to in the defaults for optional, keyword, or `&aux` arguments which *do* correspond to slots.

You can specify any number of full-format `:constructor` options on a structure. The default constructor is still generated as well unless you disable it with a simple-format `:constructor` option.

```
(defstruct
 (person
  (:constructor nil)   ; no default constructor
  (:constructor new-person (name sex &optional (age 0)))
  (:constructor new-hound (&key (name "Rover")
                                (dog-years 0)
                          &aux (age (* 7 dog-years))
                               (sex 'canine))))
 name age sex)
```

The first constructor here takes its arguments positionally rather than by keyword. (In official Common Lisp terminology, constructors that work By Order of Arguments instead of by keyword are called "BOA constructors." No, I'm not making this up.) For example, `(new-person "Jane" 'female)` generates a person whose slots are `"Jane"`, 0, and `female`, respectively.

The second constructor takes two keyword arguments, `:name`, which initializes the `name` slot and defaults to `"Rover"`, and `:dog-years`, which does not itself correspond to a slot but which is used to initialize the `age` slot. The `sex` slot is forced to the symbol `canine` with no syntax for overriding it.

`:copier`

The argument is an alternate name for the copier function for this type. The default is `copy-name`. `nil` means not to generate a copier function. (In this implementation, all copier

functions are simply synonyms for `copy-sequence`.)

`:predicate`

The argument is an alternate name for the predicate which recognizes objects of this type. The default is `name-p`. `nil` means not to generate a predicate function. (If the `:type` option is used without the `:named` option, no predicate is ever generated.)

In true Common Lisp, `typep` is always able to recognize a structure object even if `:predicate` was used. In this package, `typep` simply looks for a function called `typename-p`, so it will work for structure types only if they used the default predicate name.

`:include`

This option implements a very limited form of C++-style inheritance. The argument is the name of another structure type previously created with `defstruct`. The effect is to cause the new structure type to inherit all of the included structure's slots (plus, of course, any new slots described by this struct's slot descriptors). The new structure is considered a "specialization" of the included one. In fact, the predicate and slot accessors for the included type will also accept objects of the new type.

If there are extra arguments to the `:include` option after the included-structure name, these options are treated as replacement slot descriptors for slots in the included structure, possibly with modified default values. Borrowing an example from Steele:

```
(defstruct person name (age 0) sex)
     => person
(defstruct (astronaut (:include person (age 45)))
  helmet-size
  (favorite-beverage 'tang))
     => astronaut

(setq joe (make-person :name "Joe"))
     => [cl-struct-person "Joe" 0 nil]
(setq buzz (make-astronaut :name "Buzz"))
     => [cl-struct-astronaut "Buzz" 45 nil nil tang]

(list (person-p joe) (person-p buzz))
     => (t t)
(list (astronaut-p joe) (astronaut-p buzz))
     => (nil t)

(person-name buzz)
     => "Buzz"
(astronaut-name joe)
     => error: "astronaut-name accessing a non-astronaut"
```

Thus, if `astronaut` is a specialization of `person`, then every `astronaut` is also a `person` (but not the other way around). Every `astronaut` includes all the slots of a `person`, plus extra

slots that are specific to astronauts. Operations that work on people (like `person-name`) wo astronauts just like other people.

`:print-function`

In full Common Lisp, this option allows you to specify a function which is called to print an instance of the structure type. The Emacs Lisp system offers no hooks into the Lisp printer which would allow for such a feature, so this package simply ignores `:print-function`.

`:type`

The argument should be one of the symbols `vector` or `list`. This tells which underlying Lisp data type should be used to implement the new structure type. Vectors are used by default, but `(:type list)` will cause structure objects to be stored as lists instead.

The vector representation for structure objects has the advantage that all structure slots can be accessed quickly, although creating vectors is a bit slower in Emacs Lisp. Lists are easier to create, but take a relatively long time accessing the later slots.

`:named`

This option, which takes no arguments, causes a characteristic "tag" symbol to be stored at the front of the structure object. Using `:type` without also using `:named` will result in a structure type stored as plain vectors or lists with no identifying features.

The default, if you don't specify `:type` explicitly, is to use named vectors. Therefore, `:named` is only useful in conjunction with `:type`.

```
(defstruct (person1) name age sex)
(defstruct (person2 (:type list) :named) name age sex)
(defstruct (person3 (:type list)) name age sex)

(setq p1 (make-person1))
     => [cl-struct-person1 nil nil nil]
(setq p2 (make-person2))
     => (person2 nil nil nil)
(setq p3 (make-person3))
     => (nil nil nil)

(person1-p p1)
     => t
(person2-p p2)
     => t
(person3-p p3)
     => error: function person3-p undefined
```

Since unnamed structures don't have tags, `defstruct` is not able to make a useful predicate for recognizing them. Also, accessors like `person3-name` will be generated but they will not be able to do any type checking. The `person3-name` function, for example, will simply be a synonym for `car` in this case. By contrast, `person2-name` is able to verify that its argument is indeed a `person2` object before proceeding.

`:initial-offset`

The argument must be a nonnegative integer. It specifies a number of slots to be left "empty" at the front of the structure. If the structure is named, the tag appears at the specified position in the list or vector; otherwise, the first slot appears at that position. Earlier positions are filled with `nil` by the constructors and ignored otherwise. If the type `:includes` another type, then `:initial-offset` specifies a number of slots to be skipped between the last slot of the included type and the first new slot.

Except as noted, the `defstruct` facility of this package is entirely compatible with that of Common Lisp.

@chapno=23

Go to the previous, next section.

Go to the previous, next section.

Assertions and Errors

This section describes two macros that test assertions, i.e., conditions which must be true if the program is operating correctly. Assertions never add to the behavior of a Lisp program; they simply make "sanity checks" to make sure everything is as it should be.

If the optimization property `speed` has been set to 3, and `safety` is less than 3, then the byte-compiler will optimize away the following assertions. Because assertions might be optimized away, it is a bad idea for them to include side-effects.

Special Form: **assert** *test-form [show-args string args...]*

This form verifies that test-form is true (i.e., evaluates to a non-`nil` value). If so, it returns `nil`. If the test is not satisfied, `assert` signals an error.

A default error message will be supplied which includes test-form. You can specify a different error message by including a string argument plus optional extra arguments. Those arguments are simply passed to `error` to signal the error.

If the optional second argument show-args is `t` instead of `nil`, then the error message (with or without string) will also include all non-constant arguments of the top-level form. For example:

```
(assert (> x 10) t "x is too small: %d")
```

This usage of show-args is an extension to Common Lisp. In true Common Lisp, the second argument gives a list of places which can be `setf`'d by the user before continuing from the error. Since Emacs Lisp does not support continuable errors, it makes no sense to specify places.

Special Form: **check-type** *form type [string]*

This form verifies that form evaluates to a value of type type. If so, it returns `nil`. If not, `check-type` signals a `wrong-type-argument` error. The default error message lists the erroneous value along with type and form themselves. If string is specified, it is included in the error message in place of type. For example:

```
(check-type x (integer 1 *) "a positive integer")
```

See section Type Predicates, for a description of the type specifiers that may be used for type.

Note that in Common Lisp, the first argument to `check-type` must be a place suitable for use by `setf`, because `check-type` signals a continuable error that allows the user to modify place.

The following error-related macro is also defined:

Special Form: **ignore-errors** *forms...*

This executes forms exactly like a `progn`, except that errors are ignored during the forms. More precisely, if an error is signaled then `ignore-errors` immediately aborts execution of the forms and returns `nil`. If the forms complete successfully, `ignore-errors` returns the result of the last form.

Go to the previous, next section.

Go to the previous, next section.

Efficiency Concerns

Macros

Many of the advanced features of this package, such as `defun*`, `loop`, and `setf`, are implemented as Lisp macros. In byte-compiled code, these complex notations will be expanded into equivalent Lisp code which is simple and efficient. For example, the forms

```
(incf i n)
(push x (car p))
```

are expanded at compile-time to the Lisp forms

```
(setq i (+ i n))
(setcar p (cons x (car p)))
```

which are the most efficient ways of doing these respective operations in Lisp. Thus, there is no performance penalty for using the more readable `incf` and `push` forms in your compiled code.

Interpreted code, on the other hand, must expand these macros every time they are executed. For this reason it is strongly recommended that code making heavy use of macros be compiled. (The features labeled "Special Form" instead of "Function" in this manual are macros.) A loop using `incf` a hundred times will execute considerably faster if compiled, and will also garbage-collect less because the macro expansion will not have to be generated, used, and thrown away a hundred times.

You can find out how a macro expands by using the `cl-prettyexpand` function.

Function: **cl-prettyexpand** *form &optional full*

This function takes a single Lisp form as an argument and inserts a nicely formatted copy of it in the current buffer (which must be in Lisp mode so that indentation works properly). It also expands all Lisp macros which appear in the form. The easiest way to use this function is to go to the `*scratch*` buffer and type, say,

```
(cl-prettyexpand '(loop for x below 10 collect x))
```

and type C-x C-e immediately after the closing parenthesis; the expansion

```
(block nil
  (let* ((x 0)
         (G1004 nil))
    (while (< x 10)
      (setq G1004 (cons x G1004))
```


```
    (setq x (+ x 1)))
  (nreverse G1004)))
```

will be inserted into the buffer. (The `block` macro is expanded differently in the interpreter and compiler, so `cl-prettyexpand` just leaves it alone. The temporary variable `G1004` was created by `gensym`.)

If the optional argument full is true, then *all* macros are expanded, including `block`, `eval-when`, and compiler macros. Expansion is done as if form were a top-level form in a file being compiled. For example,

```
(cl-prettyexpand '(pushnew 'x list))
     -| (setq list (adjoin 'x list))
(cl-prettyexpand '(pushnew 'x list) t)
     -| (setq list (if (memq 'x list) list (cons 'x list)))
(cl-prettyexpand '(caddr (member* 'a list)) t)
     -| (car (cdr (cdr (memq 'a list))))
```

Note that `adjoin`, `caddr`, and `member*` all have built-in compiler macros to optimize them in common cases.

Error Checking

Common Lisp compliance has in general not been sacrificed for the sake of efficiency. A few exceptions have been made for cases where substantial gains were possible at the expense of marginal incompatibility. One example is the use of `memq` (which is treated very efficiently by the byte-compiler) to scan for keyword arguments; this can become confused in rare cases when keyword symbols are used as both keywords and data values at once. This is extremely unlikely to occur in practical code, and the use of `memq` allows functions with keyword arguments to be nearly as fast as functions that use `&optional` arguments.

The Common Lisp standard (as embodied in Steele's book) uses the phrase "it is an error if" to indicate a situation which is not supposed to arise in complying programs; implementations are strongly encouraged but not required to signal an error in these situations. This package sometimes omits such error checking in the interest of compactness and efficiency. For example, `do` variable specifiers are supposed to be lists of one, two, or three forms; extra forms are ignored by this package rather than signaling a syntax error. The `endp` function is simply a synonym for `null` in this package. Functions taking keyword arguments will accept an odd number of arguments, treating the trailing keyword as if it were followed by the value `nil`.

Argument lists (as processed by `defun*` and friends) *are* checked rigorously except for the minor point just mentioned; in particular, keyword arguments are checked for validity, and `&allow-other-keys` and `:allow-other-keys` are fully implemented. Keyword validity checking is slightly time consuming (though not too bad in byte-compiled code); you can use `&allow-other-keys` to omit this check. Functions defined in this package such as `find` and `member*` do check their keyword arguments for validity.

Optimizing Compiler

The byte-compiler that comes with Emacs 18 normally fails to expand macros that appear in top-level positions in the file (i.e., outside of `defun`s or other enclosing forms). This would have disastrous consequences to programs that used such top-level macros as `defun*`, `eval-when`, and `defstruct`. To work around this problem, the CL package patches the Emacs 18 compiler to expand top-level macros. This patch will apply to your own macros, too, if they are used in a top-level context. The patch will not harm versions of the Emacs 18 compiler which have already had a similar patch applied, nor will it affect the optimizing Emacs 19 byte-compiler written by Jamie Zawinski and Hallvard Furuseth. The patch is applied to the byte compiler's code in Emacs' memory, *not* to the `` `bytecomp.elc' `` file stored on disk.

The Emacs 19 compiler (for Emacs 18) is available from various Emacs Lisp archive sites such as `archive.cis.ohio-state.edu`. Its use is highly recommended; many of the Common Lisp macros emit code which can be improved by optimization. In particular, `block`s (whether explicit or implicit in constructs like `defun*` and `loop`) carry a fair run-time penalty; the optimizing compiler removes `block`s which are not actually referenced by `return` or `return-from` inside the block.

Go to the previous, next section.

Go to the previous, next section.

Common Lisp Compatibility

Following is a list of all known incompatibilities between this package and Common Lisp as documented in Steele (2nd edition).

Certain function names, such as `member`, `assoc`, and `floor`, were already taken by (incompatible) Emacs Lisp functions; this package appends `*' to the names of its Common Lisp versions of these functions.

The word `defun*` is required instead of `defun` in order to use extended Common Lisp argument lists in a function. Likewise, `defmacro*` and `function*` are versions of those forms which understand full-featured argument lists. The `&whole` keyword does not work in `defmacro` argument lists (except inside recursive argument lists).

In order to allow an efficient implementation, keyword arguments use a slightly cheesy parser which may be confused if a keyword symbol is passed as the *value* of another keyword argument. (Specifically, `(memq :keyword rest-of-arguments)` is used to scan for `:keyword` among the supplied keyword arguments.)

The `eql` and `equal` predicates do not distinguish between IEEE floating-point plus and minus zero. The `equalp` predicate has several differences with Common Lisp; see section Predicates.

The `setf` mechanism is entirely compatible, except that setf-methods return a list of five values rather than five values directly. Also, the new "`setf` function" concept (typified by `(defun (setf foo) ...)`) is not implemented.

The `do-all-symbols` form is the same as `do-symbols` with no obarray argument. In Common Lisp, this form would iterate over all symbols in all packages. Since Emacs obarrays are not a first-class package mechanism, there is no way for `do-all-symbols` to locate any but the default obarray.

The `loop` macro is complete except that `loop-finish` and type specifiers are unimplemented.

The multiple-value return facility treats lists as multiple values, since Emacs Lisp cannot support multiple return values directly. The macros will be compatible with Common Lisp if `values` or `values-list` is always used to return to a `multiple-value-bind` or other multiple-value receiver; if `values` is used without `multiple-value-...` or vice-versa the effect will be different from Common Lisp.

Many Common Lisp declarations are ignored, and others match the Common Lisp standard in concept but not in detail. For example, local `special` declarations, which are purely advisory in Emacs Lisp, do not rigorously obey the scoping rules set down in Steele's book.

The variable `*gensym-counter*` starts out with a pseudo-random value rather than with zero. This is to cope with the fact that generated symbols become interned when they are written to and loaded back from a file.

The `defstruct` facility is compatible, except that structures are of type `:type vector :name` by default rather than some special, distinct type. Also, the `:type` slot option is ignored.

The second argument of `check-type` is treated differently.

Go to the previous, next section.

Go to the previous, next section.

Old CL Compatibility

Following is a list of all known incompatibilities between this package and the older Quiroz `cl.el' package.

This package's emulation of multiple return values in functions is incompatible with that of the older package. That package attempted to come as close as possible to true Common Lisp multiple return values; unfortunately, it could not be 100% reliable and so was prone to occasional surprises if used freely. This package uses a simpler method, namely replacing multiple values with lists of values, which is more predictable though more noticeably different from Common Lisp.

The `defkeyword` form and `keywordp` function are not implemented in this package.

The `member`, `floor`, `ceiling`, `truncate`, `round`, `mod`, and `rem` functions are suffixed by `*' in this package to avoid collision with existing functions in Emacs 18 or Emacs 19. The older package simply redefined these functions, overwriting the built-in meanings and causing serious portability problems with Emacs 19. (Some more recent versions of the Quiroz package changed the names to `cl-member`, etc.; this package defines the latter names as aliases for `member*`, etc.)

Certain functions in the old package which were buggy or inconsistent with the Common Lisp standard are incompatible with the conforming versions in this package. For example, `eql` and `member` were synonyms for `eq` and `memq` in that package, `setf` failed to preserve correct order of evaluation of its arguments, etc.

Finally, unlike the older package, this package is careful to prefix all of its internal names with `cl-`. Except for a few functions which are explicitly defined as additional features (such as `floatp-safe` and `letf`), this package does not export any non-`cl-' symbols which are not also part of Common Lisp.

The `cl-compat` package

The CL package includes emulations of some features of the old `cl.el', in the form of a compatibility package `cl-compat`. To use it, put `(require 'cl-compat)` in your program.

The old package defined a number of internal routines without `cl-` prefixes or other annotations. Call to these routines may have crept into existing Lisp code. `cl-compat` provides emulations of the following internal routines: `pair-with-newsyms`, `zip-lists`, `unzip-lists`, `reassemble-arglists`, `duplicate-symbols-p`, `safe-idiv`.

Some `setf` forms translated into calls to internal functions that user code might call directly. The functions `setnth`, `setnthcdr`, and `setelt` fall in this category; they are defined by `cl-compat`, but the best fix is to change to use `setf` properly.

The `cl-compat` file defines the keyword functions `keywordp`, `keyword-of`, and `defkeyword`, which are not defined by the new CL package because the use of keywords as data is discouraged.

The `build-klist` mechanism for parsing keyword arguments is emulated by `cl-compat`; the `with-keyword-args` macro is not, however, and in any case it's best to change to use the more natural keyword argument processing offered by `defun*`.

Multiple return values are treated differently by the two Common Lisp packages. The old package's method was more compatible with true Common Lisp, though it used heuristics that caused it to report spurious multiple return values in certain cases. The `cl-compat` package defines a set of multiple-value macros that are compatible with the old CL package; again, they are heuristic in nature, but they are guaranteed to work in any case where the old package's macros worked. To avoid name collision with the "official" multiple-value facilities, the ones in `cl-compat` have capitalized names: `Values`, `Values-list`, `Multiple-value-bind`, etc.

The functions `cl-floor`, `cl-ceiling`, `cl-truncate`, and `cl-round` are defined by `cl-compat` to use the old-style multiple-value mechanism, just as they did in the old package. The newer `floor*` and friends return their two results in a list rather than as multiple values. Note that older versions of the old package used the unadorned names `floor`, `ceiling`, etc.; `cl-compat` cannot use these names because they conflict with Emacs 19 built-ins.

Go to the previous, next section.

Go to the previous, next section.

Porting Common Lisp

This package is meant to be used as an extension to Emacs Lisp, not as an Emacs implementation of true Common Lisp. Some of the remaining differences between Emacs Lisp and Common Lisp make it difficult to port large Common Lisp applications to Emacs. For one, some of the features in this package are not fully compliant with ANSI or Steele; see section Common Lisp Compatibility. But there are also quite a few features that this package does not provide at all. Here are some major omissions that you will want watch out for when bringing Common Lisp code into Emacs.

- Case-insensitivity. Symbols in Common Lisp are case-insensitive by default. Some programs refer to a function or variable as `foo` in one place and `Foo` or `FOO` in another. Emacs Lisp will treat these as three distinct symbols.

  Some Common Lisp code is written in all upper-case. While Emacs is happy to let the program's own functions and variables use this convention, calls to Lisp builtins like `if` and `defun` will have to be changed to lower-case.

- Lexical scoping. In Common Lisp, function arguments and `let` bindings apply only to references physically within their bodies (or within macro expansions in their bodies). Emacs Lisp, by contrast, uses dynamic scoping wherein a binding to a variable is visible even inside functions called from the body.

  Variables in Common Lisp can be made dynamically scoped by declaring them `special` or using `defvar`. In Emacs Lisp it is as if all variables were declared `special`.

  Often you can use code that was written for lexical scoping even in a dynamically scoped Lisp, but not always. Here is an example of a Common Lisp code fragment that would fail in Emacs Lisp:

```
(defun map-odd-elements (func list)
  (loop for x in list
        for flag = t then (not flag)
        collect (if flag x (funcall func x))))

(defun add-odd-elements (list x)
  (map-odd-elements (function (lambda (a) (+ a x))) list))
```

  In Common Lisp, the two functions' usages of `x` are completely independent. In Emacs Lisp, the binding to `x` made by `add-odd-elements` will have been hidden by the binding in `map-odd-elements` by the time the `(+ a x)` function is called.

  (This package avoids such problems in its own mapping functions by using names like `cl-x` instead of `x` internally; as long as you don't use the `cl-` prefix for your own variables no collision can occur.)

  See section Lexical Bindings, for a description of the `lexical-let` form which establishes a Common Lisp-style lexical binding, and some examples of how it differs from Emacs' regular `let`.

- Common Lisp allows the shorthand `#'x` to stand for `(function x)`, just as `'x` stands for `(quote x)`. In Common Lisp, one traditionally uses `#'` notation when referring to the name of a function. In

Emacs Lisp, it works just as well to use a regular quote:

```
(loop for x in y by #'cddr collect (mapcar #'plusp x))  ; Common Lisp
(loop for x in y by 'cddr collect (mapcar 'plusp x))    ; Emacs Lisp
```

When `#'` introduces a `lambda` form, it is best to write out `(function ...)` longhand in Emacs Lisp. You can use a regular quote, but then the byte-compiler won't know that the `lambda` expression is code that can be compiled.

```
(mapcar #'(lambda (x) (* x 2)) list)            ; Common Lisp
(mapcar (function (lambda (x) (* x 2))) list)   ; Emacs Lisp
```

Lucid Emacs supports `#'` notation starting with version 19.8.

- The "backquote" feature uses a different syntax in Emacs Lisp.

```
(defmacro foo (v &rest body) `(let ((,v 0)) @,body))  ; Common Lisp
(defmacro foo (v &rest body) (` (let (((, v) 0)) (@, body)))  ; Emacs
```

- Reader macros. Common Lisp includes a second type of macro that works at the level of individual characters. For example, Common Lisp implements the quote notation by a reader macro called `'`, whereas Emacs Lisp's parser just treats quote as a special case. Some Lisp packages use reader macros to create special syntaxes for themselves, which the Emacs parser is incapable of reading.

  The lack of reader macros, incidentally, is the reason behind Emacs Lisp's unusual backquote syntax. Since backquotes are implemented as a Lisp package and not built-in to the Emacs parser, they are forced to use a regular macro named ` which is used with the standard function/macro call notation.

- Other syntactic features. Common Lisp provides a number of notations beginning with # that the Emacs Lisp parser won't understand. For example, `#| ... |#' is an alternate comment notation, and `#+lucid (foo)' tells the parser to ignore the `(foo)` except in Lucid Common Lisp.

- Packages. In Common Lisp, symbols are divided into packages. Symbols that are Lisp built-ins are typically stored in one package; symbols that are vendor extensions are put in another, and each application program would have a package for its own symbols. Certain symbols are "exported" by a package and others are internal; certain packages "use" or import the exported symbols of other packages. To access symbols that would not normally be visible due to this importing and exporting, Common Lisp provides a syntax like `package:symbol` or `package::symbol`.

  Emacs Lisp has a single namespace for all interned symbols, and then uses a naming convention of putting a prefix like `cl-` in front of the name. Some Emacs packages adopt the Common Lisp-like convention of using `cl:` or `cl::` as the prefix. However, the Emacs parser does not understand colons and just treats them as part of the symbol name. Thus, while `mapcar` and `lisp:mapcar` may refer to the same symbol in Common Lisp, they are totally distinct in Emacs Lisp. Common Lisp programs which refer to a symbol by the full name sometimes and the short name other times will not port cleanly to Emacs.

  Emacs Lisp does have a concept of "obarrays," which are package-like collections of symbols, but this feature is not strong enough to be used as a true package mechanism.

- Keywords. The notation `:test-not` in Common Lisp really is a shorthand for `keyword:test-not`; keywords are just symbols in a built-in `keyword` package with the special

property that all its symbols are automatically self-evaluating. Common Lisp programs often use keywords liberally to avoid having to use quotes.

In Emacs Lisp a keyword is just a symbol whose name begins with a colon; since the Emacs parser does not treat them specially, they have to be explicitly made self-evaluating by a statement like `(setq :test-not ':test-not)`. This package arranges to execute such a statement whenever `defun*` or some other form sees a keyword being used as an argument. Common Lisp code that assumes that a symbol `:mumble` will be self-evaluating even though it was never introduced by a `defun*` will have to be fixed.

- The `format` function is quite different between Common Lisp and Emacs Lisp. It takes an additional "destination" argument before the format string. A destination of `nil` means to format to a string as in Emacs Lisp; a destination of `t` means to write to the terminal (similar to `message` in Emacs). Also, format control strings are utterly different; `~` is used instead of `%` to introduce format codes, and the set of available codes is much richer. There are no notations like `\n` for string literals; instead, `format` is used with the "newline" format code, `~%`. More advanced formatting codes provide such features as paragraph filling, case conversion, and even loops and conditionals.

  While it would have been possible to implement most of Common Lisp `format` in this package (under the name `format*`, of course), it was not deemed worthwhile. It would have required a huge amount of code to implement even a decent subset of `format*`, yet the functionality it would provide over Emacs Lisp's `format` would rarely be useful.
- Vector constants use square brackets in Emacs Lisp, but `#(a b c)` notation in Common Lisp. To further complicate matters, Emacs 19 introduces its own `#(` notation for something entirely different--strings with properties.
- Characters are distinct from integers in Common Lisp. The notation for character constants is also different: `#\A` instead of `?A`. Also, `string=` and `string-equal` are synonyms in Emacs Lisp whereas the latter is case-insensitive in Common Lisp.
- Data types. Some Common Lisp data types do not exist in Emacs Lisp. Rational numbers and complex numbers are not present, nor are large integers (all integers are "fixnums"). All arrays are one-dimensional. There are no readtables or pathnames; streams are a set of existing data types rather than a new data type of their own. Hash tables, random-states, structures, and packages (obarrays) are built from Lisp vectors or lists rather than being distinct types.
- The Common Lisp Object System (CLOS) is not implemented, nor is the Common Lisp Condition System.
- Common Lisp features that are completely redundant with Emacs Lisp features of a different name generally have not been implemented. For example, Common Lisp writes `defconstant` where Emacs Lisp uses `defconst`. Similarly, `make-list` takes its arguments in different ways in the two Lisps but does exactly the same thing, so this package has not bothered to implement a Common Lisp-style `make-list`.
- A few more notable Common Lisp features not included in this package: `compiler-let`, `tagbody`, `prog`, `ldb/dpb`, `parse-integer`, `cerror`.
- Recursion. While recursion works in Emacs Lisp just like it does in Common Lisp, various details of the Emacs Lisp system and compiler make recursion much less efficient than it is in most Lisps. Some schools of thought prefer to use recursion in Lisp over other techniques; they would sum a list of numbers using something like


```
(defun sum-list (list)
  (if list
      (+ (car list) (sum-list (cdr list)))
    0))
```

where a more iteratively-minded programmer might write one of these forms:

```
(let ((total 0)) (dolist (x my-list) (incf total x)) total)
(loop for x in my-list sum x)
```

While this would be mainly a stylistic choice in most Common Lisps, in Emacs Lisp you should be aware that the iterative forms are much faster than recursion. Also, Lisp programmers will want to note that the current Emacs Lisp compiler does not optimize tail recursion.

Go to the previous, next section.

Go to the previous, next section.

Function Index

a

b

c

- copy-tree
- count
- count-if
- count-if-not

d

- decf
- declaim
- declare
- defalias
- define-compiler-macro
- define-modify-macro
- define-setf-method
- defmacro*
- defsetf
- defstruct
- defsubst*
- deftype
- defun*
- delete
- delete*
- delete-duplicates
- delete-if
- delete-if-not
- destructuring-bind
- do
- do*
- do-all-symbols
- do-symbols
- dolist
- dotimes

e

f

g

h

- hash-table-count
- hash-table-p

i

- ignore-errors
- incf
- intersection
- isqrt

l

- labels
- last
- lcm
- ldiff
- letf
- letf*
- lexical-let
- lexical-let*
- list*
- list-length
- load-time-value
- locally
- loop

m

- macrolet
- make-hash-table
- make-random-state
- map
- mapc

n

o

- oddp

p

- pairlis
- plusp
- pop
- position
- position-if
- position-if-not
- proclaim
- progv
- psetf
- psetq
- push
- pushnew

r

- random*
- random-state-p
- rassoc
- rassoc*
- rassoc-if
- rassoc-if-not
- reduce
- rem*
- remf
- remhash
- remove
- remove*
- remove-duplicates

S

t

u

w

Go to the previous, next section.

Go to the previous section.

Variable Index

*

f

l

m

Go to the previous section.

The C Preprocessor

Last revised July 1992

for GCC version 2

Richard M. Stallman

The C Preprocessor

Transformations Made Globally

Most C preprocessor features are inactive unless you give specific commands to request their use. (Preprocessor commands are lines starting with `#'; see section Preprocessor Commands). But there are three transformations that the preprocessor always makes on all the input it receives, even in the absence of commands.

- All C comments are replaced with single spaces.
- Backslash-Newline sequences are deleted, no matter where. This feature allows you to break long lines for cosmetic purposes without changing their meaning.
- Predefined macro names are replaced with their expansions (see section Predefined Macros).

The first two transformations are done *before* nearly all other parsing and before preprocessor commands are recognized. Thus, for example, you can split a line cosmetically with Backslash-Newline anywhere (except when trigraphs are in use; see below).

```
/*
*/ # /*
*/ defi\
ne FO\
O 10\
20
```

is equivalent into `#define FOO 1020'. You can split even an escape sequence with Backslash-Newline. For example, you can split `"foo\bar"` between the `\' and the `b' to get

```
"foo\\
bar"
```

This behavior is unclean: in all other contexts, a Backslash can be inserted in a string constant as an ordinary character by writing a double Backslash, and this creates an exception. But the ANSI C standard requires it. (Strict ANSI C does not allow Newlines in string constants, so they do not consider this a problem.)

But there are a few exceptions to all three transformations.

- C comments and predefined macro names are not recognized inside a `#include' command in which the file name is delimited with `<' and `>'.
- C comments and predefined macro names are never recognized within a character or string constant. (Strictly speaking, this is the rule, not an exception, but it is worth noting here anyway.)
- Backslash-Newline may not safely be used within an ANSI "trigraph". Trigraphs are converted before Backslash-Newline is deleted. If you write what looks like a trigraph with a Backslash-Newline inside, the Backslash-Newline is deleted as usual, but it is then too late to

recognize the trigraph.

This exception is relevant only if you use the `-trigraphs' option to enable trigraph processing. See section Invoking the C Preprocessor.

Preprocessor Commands

Most preprocessor features are active only if you use preprocessor commands to request their use.

Preprocessor commands are lines in your program that start with `#'. The `#' is followed by an identifier that is the command name. For example, `#define' is the command that defines a macro. Whitespace is also allowed before and after the `#'.

The set of valid command names is fixed. Programs cannot define new preprocessor commands.

Some command names require arguments; these make up the rest of the command line and must be separated from the command name by whitespace. For example, `#define' must be followed by a macro name and the intended expansion of the macro.

A preprocessor command cannot be more than one line in normal circumstances. It may be split cosmetically with Backslash-Newline, but that has no effect on its meaning. Comments containing Newlines can also divide the command into multiple lines, but the comments are changed to Spaces before the command is interpreted. The only way a significant Newline can occur in a preprocessor command is within a string constant or character constant. Note that most C compilers that might be applied to the output from the preprocessor do not accept string or character constants containing Newlines.

The `#' and the command name cannot come from a macro expansion. For example, if `foo' is defined as a macro expanding to `define', that does not make `#foo' a valid preprocessor command.

Header Files

A header file is a file containing C declarations and macro definitions (see section Macros) to be shared between several source files. You request the use of a header file in your program with the C preprocessor command `#include'.

Uses of Header Files

Header files serve two kinds of purposes.

- System header files declare the interfaces to parts of the operating system. You include them in your program to supply the definitions and declarations you need to invoke system calls and libraries.
- Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.

Including a header file produces the same results in C compilation as copying the header file into each source file that needs it. But such copying would be time-consuming and error-prone. With a header the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs that include the header file will automatically use the new version when next recompiled. The header file eliminates the labor of finding and changing all the copies as well as the risk that a failure to find one copy will result in inconsistencies within a program.

The usual convention is to give header files names that end with `.h'.

The `#include' Command

Both user and system header files are included using the preprocessor command `#include'. It has three variants:

`#include <file>`

This variant is used for system header files. It searches for a file named file in a list of directories specified by you, then in a standard list of system directories. You specify directories to search for header files with the command option `-I' (see section Invoking the C Preprocessor). The option `-nostdinc' inhibits searching the standard system directories; in this case only the directories you specify are searched.

The parsing of this form of `#include' is slightly special because comments are not recognized within the `<...>'. Thus, in `#include <x/*y>' the `/*' does not start a comment and the command specifies inclusion of a system header file named `x/*y'. Of course, a header file with such a name is unlikely to exist on Unix, where shell wildcard features would make it hard to manipulate.

The argument file may not contain a `>' character. It may, however, contain a `<' character.

`#include "file"`

This variant is used for header files of your own program. It searches for a file named file first in the current directory, then in the same directories used for system header files. The current directory is the directory of the current input file. It is tried first because it is presumed to be the location of the files that the current input file refers to. (If the `-I-' option is used, the special treatment of the current directory is inhibited.)

The argument file may not contain `"' characters. If backslashes occur within file, they are considered ordinary text characters, not escape characters. None of the character escape sequences appropriate to string constants in C are processed. Thus, `#include "x\n\\y"' specifies a filename containing three backslashes. It is not clear why this behavior is ever useful, but the ANSI standard specifies it.

`#include anything else`

This variant is called a computed #include. Any `#include' command whose argument does not fit the above two forms is a computed include. The text anything else is checked for macro calls, which are expanded (see section Macros). When this is done, the result must fit one of the above two variants--in particular, the expanded text must in the end be surrounded by either quotes or angle braces.

This feature allows you to define a macro which controls the file name to be used at a later pc
the program. One application of this is to allow a site-configuration file for your program to
specify the names of the system include files to be used. This can help in porting the program to
various operating systems in which the necessary system header files are found in different places.

How `#include' Works

The `#include' command works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the `#include' command. For example, given two files as follows:

```
/* File program.c */
int x;
#include "header.h"

main ()
{
  printf (test ());
}


/* File header.h */
char *test ();
```

the output generated by the C preprocessor for `program.c' as input would be

```
int x;
char *test ();

main ()
{
  printf (test ());
}
```

Included files are not limited to declarations and macro definitions; those are merely the typical uses. Any fragment of a C program can be included from another file. The include file could even contain the beginning of a statement that is concluded in the containing file, or the end of a statement that was started in the including file. However, a comment or a string or character constant may not start in the included file and finish in the including file. An unterminated comment, string constant or character constant in an included file is considered to end (with an error message) at the end of the file.

The line following the `#include' command is always treated as a separate line by the C preprocessor even if the included file lacks a final newline.

Once-Only Include Files

Very often, one header file includes another. It can easily result that a certain header file is included more than once. This may lead to errors, if the header file defines structure types or typedefs, and is certainly wasteful. Therefore, we often wish to prevent multiple inclusion of a header file.

The standard way to do this is to enclose the entire real contents of the file in a conditional, like this:

```
#ifndef __FILE_FOO_SEEN__
#define __FILE_FOO_SEEN__

the entire file

#endif /* __FILE_FOO_SEEN__ */
```

The macro `__FILE_FOO_SEEN__` indicates that the file has been included once already; its name should begin with `__' to avoid conflicts with user programs, and it should contain the name of the file and some additional text, to avoid conflicts with other header files.

The GNU C preprocessor is programmed to notice when a header file uses this particular construct and handle it efficiently. If a header file is contained entirely in a `#ifndef' conditional, then it records that fact. If a subsequent `#include' specifies the same file, and the macro in the `#ifndef' is already defined, then the file is entirely skipped, without even reading it.

There is also an explicit command to tell the preprocessor that it need not include a file more than once. This is called `#pragma once', and was used *in addition to* the `#ifndef' conditional around the contents of the header file. `#pragma once' is now obsolete and should not be used at all.

In the Objective C language, there is a variant of `#include' called `#import' which includes a file, but does so at most once. If you use `#import' *instead of* `#include', then you don't need the conditionals inside the header file to prevent multiple execution of the contents.

`#import' is obsolete because it is not a well-designed feature. It requires the users of a header file--the applications programmers--to know that a certain header file should only be included once. It is much better for the header file's implementor to write the file so that users don't need to know this. Using `#ifndef' accomplishes this goal.

Inheritance and Header Files

Inheritance is what happens when one object or file derives some of its contents by virtual copying from another object or file. In the case of C header files, inheritance means that one header file includes another header file and then replaces or adds something.

If the inheriting header file and the base header file have different names, then inheritance is straightforward: simply write `#include "base"' in the inheriting file.

Sometimes it is necessary to give the inheriting file the same name as the base file. This is less

straightforward.

For example, suppose an application program uses the system header file `sys/signal.h', but the version of `/usr/include/sys/signal.h' on a particular system doesn't do what the application program expects. It might be convenient to define a "local" version, perhaps under the name `/usr/local/include/sys/signal.h', to override or add to the one supplied by the system.

You can do this by using the option `-I.' for compilation, and writing a file `sys/signal.h' that does what the application program expects. But making this file include the standard `sys/signal.h' is not so easy--writing `#include <sys/signal.h>' in that file doesn't work, because it includes your own version of the file, not the standard system version. Used in that file itself, this leads to an infinite recursion and a fatal error in compilation.

`#include </usr/include/sys/signal.h>' would find the proper file, but that is not clean, since it makes an assumption about where the system header file is found. This is bad for maintenance, since it means that any change in where the system's header files are kept requires a change somewhere else.

The clean way to solve this problem is to use `#include_next', which means, "Include the *next* file with this name." This command works like `#include' except in searching for the specified file: it starts searching the list of header file directories *after* the directory in which the current file was found.

Suppose you specify `-I /usr/local/include', and the list of directories to search also includes `/usr/include'; and suppose that both directories contain a file named `sys/signal.h'. Ordinary `#include <sys/signal.h>' finds the file under `/usr/local/include'. If that file contains `#include_next <sys/signal.h>', it starts searching after that directory, and finds the file in `/usr/include'.

Macros

A macro is a sort of abbreviation which you can define once and then use later. There are many complicated features associated with macros in the C preprocessor.

Simple Macros

A simple macro is a kind of abbreviation. It is a name which stands for a fragment of code. Some people refer to these as manifest constants.

Before you can use a macro, you must define it explicitly with the `#define' command. `#define' is followed by the name of the macro and then the code it should be an abbreviation for. For example,

```
#define BUFFER_SIZE 1020
```

defines a macro named `BUFFER_SIZE' as an abbreviation for the text `1020'. Therefore, if somewhere after this `#define' command there comes a C statement of the form

```
foo = (char *) xmalloc (BUFFER_SIZE);
```

then the C preprocessor will recognize and expand the macro `BUFFER_SIZE', resulting in

```
foo = (char *) xmalloc (1020);
```

the definition must be a single line; however, it may not end in the middle of a multi-line string constant or character constant.

The use of all upper case for macro names is a standard convention. Programs are easier to read when it is possible to tell at a glance which names are macros.

Normally, a macro definition must be a single line, like all C preprocessor commands. (You can split a long macro definition cosmetically with Backslash-Newline.) There is one exception: Newlines can be included in the macro definition if within a string or character constant. By the same token, it is not possible for a macro definition to contain an unbalanced quote character; the definition automatically extends to include the matching quote character that ends the string or character constant. Comments within a macro definition may contain Newlines, which make no difference since the comments are entirely replaced with Spaces regardless of their contents.

Aside from the above, there is no restriction on what can go in a macro body. Parentheses need not balance. The body need not resemble valid C code. (Of course, you might get error messages from the C compiler when you use the macro.)

The C preprocessor scans your program sequentially, so macro definitions take effect at the place you write them. Therefore, the following input to the C preprocessor

```
foo = X;
#define X 4
bar = X;
```

produces as output

```
foo = X;

bar = 4;
```

After the preprocessor expands a macro name, the macro's definition body is appended to the front of the remaining input, and the check for macro calls continues. Therefore, the macro body can contain calls to other macros. For example, after

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
```

the name `TABLESIZE' when used in the program would go through two stages of expansion, resulting ultimately in `1020'.

This is not at all the same as defining `TABLESIZE' to be `1020'. The `#define' for `TABLESIZE' uses exactly the body you specify--in this case, `BUFSIZE'---and does not check to see whether it too is the name of a macro. It's only when you *use* `TABLESIZE' that the result of its expansion is checked for

more macro names. See section Cascaded Use of Macros.

Macros with Arguments

A simple macro always stands for exactly the same text, each time it is used. Macros can be more flexible when they accept arguments. Arguments are fragments of code that you supply each time the macro is used. These fragments are included in the expansion of the macro according to the directions in the macro definition.

To define a macro that uses arguments, you write a `#define' command with a list of argument names in parentheses after the name of the macro. The argument names may be any valid C identifiers, separated by commas and optionally whitespace. The open-parenthesis must follow the macro name immediately, with no space in between.

For example, here is a macro that computes the minimum of two numeric values, as it is defined in many C programs:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

(This is not the best way to define a "minimum" macro in GNU C. See section Duplication of Side Effects, for more information.)

To use a macro that expects arguments, you write the name of the macro followed by a list of actual arguments in parentheses, separated by commas. The number of actual arguments you give must match the number of arguments the macro expects. Examples of use of the macro `min' include `min (1, 2)' and `min (x + 28, *p)'.

The expansion text of the macro depends on the arguments you use. Each of the argument names of the macro is replaced, throughout the macro definition, with the corresponding actual argument. Using the same macro `min' defined above, `min (1, 2)' expands into

```
((1) < (2) ? (1) : (2))
```

where `1' has been substituted for `X' and `2' for `Y'.

Likewise, `min (x + 28, *p)' expands into

```
((x + 28) < (*p) ? (x + 28) : (*p))
```

Parentheses in the actual arguments must balance; a comma within parentheses does not end an argument. However, there is no requirement for brackets or braces to balance, and they do not prevent a comma from separating arguments. Thus,

```
macro (array[x = y, x + 1])
```

passes two arguments to `macro`: `array[x = y' and `x + 1]'. If you want to supply `array[x = y, x + 1]' as an argument, you must write it as `array[(x = y, x + 1)]', which is equivalent C code.

After the actual arguments are substituted into the macro body, the entire result is appended to the fr of the remaining input, and the check for macro calls continues. Therefore, the actual arguments can contain calls to other macros, either with or without arguments, or even to the same macro. The macro body can also contain calls to other macros. For example, `min (min (a, b), c)' expands into this text:

```
((((a) < (b) ? (a) : (b))) < (c)
 ? (((a) < (b) ? (a) : (b)))
 : (c))
```

(Line breaks shown here for clarity would not actually be generated.)

If a macro `foo` takes one argument, and you want to supply an empty argument, you must write at least some whitespace between the parentheses, like this: `foo ( )'. Just `foo ()' is providing no arguments, which is an error if `foo` expects an argument. But `foo0 ()' is the correct way to call a macro defined to take zero arguments, like this:

```
#define foo0() ...
```

If you use the macro name followed by something other than an open-parenthesis (after ignoring any spaces, tabs and comments that follow), it is not a call to the macro, and the preprocessor does not change what you have written. Therefore, it is possible for the same name to be a variable or function in your program as well as a macro, and you can choose in each instance whether to refer to the macro (if an actual argument list follows) or the variable or function (if an argument list does not follow).

Such dual use of one name could be confusing and should be avoided except when the two meanings are effectively synonymous: that is, when the name is both a macro and a function and the two have similar effects. You can think of the name simply as a function; use of the name for purposes other than calling it (such as, to take the address) will refer to the function, while calls will expand the macro and generate better but equivalent code. For example, you can use a function named `min' in the same source file that defines the macro. If you write `&min' with no argument list, you refer to the function. If you write `min (x, bb)', with an argument list, the macro is expanded. If you write `(min) (a, bb)', where the name `min' is not followed by an open-parenthesis, the macro is not expanded, so you wind up with a call to the function `min'.

You may not define the same name as both a simple macro and a macro with arguments.

In the definition of a macro with arguments, the list of argument names must follow the macro name immediately with no space in between. If there is a space after the macro name, the macro is defined as taking no arguments, and all the rest of the line is taken to be the expansion. The reason for this is that it is often useful to define a macro that takes no arguments and whose definition begins with an identifier in parentheses. This rule about spaces makes it possible for you to do either this:

```
#define FOO(x) - 1 / (x)
```

(which defines `FOO' to take an argument and expand into minus the reciprocal of that argument) or this:

```
#define BAR (x) - 1 / (x)
```

(which defines `BAR' to take no argument and always expand into `(x) - 1 / (x)').

Note that the *uses* of a macro with arguments can have spaces before the left parenthesis; it's the *definition* where it matters whether there is a space.

Predefined Macros

Several simple macros are predefined. You can use them without giving definitions for them. They fall into two classes: standard macros and system-specific macros.

Standard Predefined Macros

The standard predefined macros are available with the same meanings regardless of the machine or operating system on which you are using GNU C. Their names all start and end with double underscores. Those preceding `__GNUC__` in this table are standardized by ANSI C; the rest are GNU C extensions.

`__FILE__`

This macro expands to the name of the current input file, in the form of a C string constant. The precise name returned is the one that was specified in `#include' or as the input file name argument.

`__LINE__`

This macro expands to the current input line number, in the form of a decimal integer constant. While we call it a predefined macro, it's a pretty strange macro, since its "definition" changes with each new line of source code.

This and `__FILE__' are useful in generating an error message to report an inconsistency detected by the program; the message can state the source line at which the inconsistency was detected. For example,

```
fprintf (stderr, "Internal error: "
                 "negative string length "
                 "%d at %s, line %d.",
         length, __FILE__, __LINE__);
```

A `#include' command changes the expansions of `__FILE__' and `__LINE__' to correspond to the included file. At the end of that file, when processing resumes on the input file that contained the `#include' command, the expansions of `__FILE__' and `__LINE__' revert to the values they had before the `#include' (but `__LINE__' is then incremented by one as processing moves to the line after the `#include').

The expansions of both `__FILE__' and `__LINE__' are altered if a `#line' command is used. See section Combining Source Files.

`__INCLUDE_LEVEL__`

This macro expands to a decimal integer constant that represents the depth of nesting in include files. The value of this macro is incremented on every `#include' command and decremented at every end of file.

`__DATE__`

This macro expands to a string constant that describes the date on which the preprocessor is being run. The string constant contains eleven characters and looks like `"Jan 29 1987"' or `"Apr 1 1905"'.

`__TIME__`

This macro expands to a string constant that describes the time at which the preprocessor is being run. The string constant contains eight characters and looks like `"23:59:01"'.

`__STDC__`

This macro expands to the constant 1, to signify that this is ANSI Standard C. (Whether that is actually true depends on what C compiler will operate on the output from the preprocessor.)

`__GNUC__`

This macro is defined if and only if this is GNU C. This macro is defined only when the entire GNU C compiler is in use; if you invoke the preprocessor directly, `__GNUC__' is undefined.

`__GNUG__`

The GNU C compiler defines this when the compilation language is C++; use `__GNUG__' to distinguish between GNU C and GNU C++.

`__cplusplus`

The draft ANSI standard for C++ used to require predefining this variable. Though it is no longer required, GNU C++ continues to define it, as do other popular C++ compilers. You can use `__cplusplus' to test whether a header is compiled by a C compiler or a C++ compiler.

`__STRICT_ANSI__`

This macro is defined if and only if the `-ansi' switch was specified when GNU C was invoked. Its definition is the null string. This macro exists primarily to direct certain GNU header files not to define certain traditional Unix constructs which are incompatible with ANSI C.

`__BASE_FILE__`

This macro expands to the name of the main input file, in the form of a C string constant. This is the source file that was specified as an argument when the C compiler was invoked.

`__VERSION__`

This macro expands to a string which describes the version number of GNU C. The string is normally a sequence of decimal numbers separated by periods, such as `"1.18"'. The only reasonable use of this macro is to incorporate it into a string constant.

`__OPTIMIZE__`

This macro is defined in optimizing compilations. It causes certain GNU header files to define alternative macro definitions for some system library functions. It is unwise to refer to or test the definition of this macro unless you make very sure that programs will execute with the same effect regardless.

`__CHAR_UNSIGNED__`

This macro is defined if and only if the data type `char` is unsigned on the target machine. It exists to cause the standard header file `limit.h' to work correctly. It is bad practice to refer to this macro yourself; instead, refer to the standard macros defined in `limit.h'. The preprocessor

uses this macro to determine whether or not to sign-extend large character constants written in octal; see section The `#if' Command.

Nonstandard Predefined Macros

The C preprocessor normally has several predefined macros that vary between machines because their purpose is to indicate what type of system and machine is in use. This manual, being for all systems and machines, cannot tell you exactly what their names are; instead, we offer a list of some typical ones. You can use `cpp -dM' to see the values of predefined macros; see section Invoking the C Preprocessor.

Some nonstandard predefined macros describe the operating system in use, with more or less specificity. For example,

`unix`

`unix' is normally predefined on all Unix systems.

`BSD`

`BSD' is predefined on recent versions of Berkeley Unix (perhaps only in version 4.3).

Other nonstandard predefined macros describe the kind of CPU, with more or less specificity. For example,

`vax`

`vax' is predefined on Vax computers.

`mc68000`

`mc68000' is predefined on most computers whose CPU is a Motorola 68000, 68010 or 68020.

`m68k`

`m68k' is also predefined on most computers whose CPU is a 68000, 68010 or 68020; however, some makers use `mc68000' and some use `m68k'. Some predefine both names. What happens in GNU C depends on the system you are using it on.

`M68020`

`M68020' has been observed to be predefined on some systems that use 68020 CPUs--in addition to `mc68000' and `m68k', which are less specific.

`_AM29K`

`_AM29000`

Both `_AM29K' and `_AM29000' are predefined for the AMD 29000 CPU family.

`ns32000`

`ns32000' is predefined on computers which use the National Semiconductor 32000 series CPU.

Yet other nonstandard predefined macros describe the manufacturer of the system. For example,

`sun`

`sun' is predefined on all models of Sun computers.

`pyr`

`pyr' is predefined on all models of Pyramid computers.

`sequent`

`sequent' is predefined on all models of Sequent computers.

These predefined symbols are not only nonstandard, they are contrary to the ANSI standard because their names do not start with underscores. Therefore, the option `-ansi' inhibits the definition of these symbols.

This tends to make `-ansi' useless, since many programs depend on the customary nonstandard predefined symbols. Even system header files check them and will generate incorrect declarations if they do not find the names that are expected. You might think that the header files supplied for the Uglix computer would not need to test what machine they are running on, because they can simply assume it is the Uglix; but often they do, and they do so using the customary names. As a result, very few C programs will compile with `-ansi'. We intend to avoid such problems on the GNU system.

What, then, should you do in an ANSI C program to test the type of machine it will run on?

GNU C offers a parallel series of symbols for this purpose, whose names are made from the customary ones by adding `__' at the beginning and end. Thus, the symbol `__vax__` would be available on a Vax, and so on.

The set of nonstandard predefined names in the GNU C preprocessor is controlled (when `cpp` is itself compiled) by the macro `CPP_PREDEFINES', which should be a string containing `-D' options, separated by spaces. For example, on the Sun 3, we use the following definition:

```
#define CPP_PREDEFINES "-Dmc68000 -Dsun -Dunix -Dm68k"
```

This macro is usually specified in `tm.h'.

Stringification

Stringification means turning a code fragment into a string constant whose contents are the text for the code fragment. For example, stringifying `foo (z)' results in `"foo (z)"'.

In the C preprocessor, stringification is an option available when macro arguments are substituted into the macro definition. In the body of the definition, when an argument name appears, the character `#' before the name specifies stringification of the corresponding actual argument when it is substituted at that point in the definition. The same argument may be substituted in other places in the definition without stringification if the argument name appears in those places with no `#'.

Here is an example of a macro definition that uses stringification:

```
#define WARN_IF(EXP) \
do { if (EXP) \
        fprintf (stderr, "Warning: " #EXP "\n"); } \
while (0)
```

Here the actual argument for `EXP' is substituted once as given, into the `if' statement, and once as stringified, into the argument to `fprintf'. The `do' and `while (0)' are a kludge to make it possible to write `WARN_IF (arg);', which the resemblance of `WARN_IF' to a function would make C programmers

want to do; see section Swallowing the Semicolon).

The stringification feature is limited to transforming one macro argument into one string constant: there is no way to combine the argument with other text and then stringify it all together. But the example above shows how an equivalent result can be obtained in ANSI Standard C using the feature that adjacent string constants are concatenated as one string constant. The preprocessor stringifies the actual value of `EXP' into a separate string constant, resulting in text like

```
do { if (x == 0) \
        fprintf (stderr, "Warning: " "x == 0" "\n"); } \
while (0)
```

but the C compiler then sees three consecutive string constants and concatenates them into one, producing effectively

```
do { if (x == 0) \
        fprintf (stderr, "Warning: x == 0\n"); } \
while (0)
```

Stringification in C involves more than putting doublequote characters around the fragment; it is necessary to put backslashes in front of all doublequote characters, and all backslashes in string and character constants, in order to get a valid C string constant with the proper contents. Thus, stringifying `p = "foo\n";' results in `"p = \"foo\\n\";"'. However, backslashes that are not inside of string or character constants are not duplicated: `\n' by itself stringifies to `"\n"'.

Whitespace (including comments) in the text being stringified is handled according to precise rules. All leading and trailing whitespace is ignored. Any sequence of whitespace in the middle of the text is converted to a single space in the stringified result.

Concatenation

Concatenation means joining two strings into one. In the context of macro expansion, concatenation refers to joining two lexical units into one longer one. Specifically, an actual argument to the macro can be concatenated with another actual argument or with fixed text to produce a longer name. The longer name might be the name of a function, variable or type, or a C keyword; it might even be the name of another macro, in which case it will be expanded.

When you define a macro, you request concatenation with the special operator `##' in the macro body. When the macro is called, after actual arguments are substituted, all `##' operators are deleted, and so is any whitespace next to them (including whitespace that was part of an actual argument). The result is to concatenate the syntactic tokens on either side of the `##'.

Consider a C program that interprets named commands. There probably needs to be a table of commands, perhaps an array of structures declared as follows:

```
struct command
{
```


```
  char *name;
  void (*function) ();
};

struct command commands[] =
{
  { "quit", quit_command},
  { "help", help_command},
  ...
};
```

It would be cleaner not to have to give each command name twice, once in the string constant and once in the function name. A macro which takes the name of a command as an argument can make this unnecessary. The string constant can be created with stringification, and the function name by concatenating the argument with `_command'. Here is how it is done:

```
#define COMMAND(NAME)  { #NAME, NAME ## _command }

struct command commands[] =
{
  COMMAND (quit),
  COMMAND (help),
  ...
};
```

The usual case of concatenation is concatenating two names (or a name and a number) into a longer name. But this isn't the only valid case. It is also possible to concatenate two numbers (or a number and a name, such as `1.5' and `e3') into a number. Also, multi-character operators such as `+=' can be formed by concatenation. In some cases it is even possible to piece together a string constant. However, two pieces of text that don't together form a valid lexical unit cannot be concatenated. For example, concatenation with `x' on one side and `+' on the other is not meaningful because those two characters can't fit together in any lexical unit of C. The ANSI standard says that such attempts at concatenation are undefined, but in the GNU C preprocessor it is well defined: it puts the `x' and `+' side by side with no particular special results.

Keep in mind that the C preprocessor converts comments to whitespace before macros are even considered. Therefore, you cannot create a comment by concatenating `/' and `*': the `/*' sequence that starts a comment is not a lexical unit, but rather the beginning of a "long" space character. Also, you can freely use comments next to a `##' in a macro definition, or in actual arguments that will be concatenated, because the comments will be converted to spaces at first sight, and concatenation will later discard the spaces.

Undefining Macros

To undefine a macro means to cancel its definition. This is done with the `#undef' command. `#undef' is followed by the macro name to be undefined.

Like definition, undefinition occurs at a specific point in the source file, and it applies starting from that point. The name ceases to be a macro name, and from that point on it is treated by the preprocessor as if it had never been a macro name.

For example,

```
#define FOO 4
x = FOO;
#undef FOO
x = FOO;
```

expands into

```
x = 4;

x = FOO;
```

In this example, `FOO' had better be a variable or function as well as (temporarily) a macro, in order for the result of the expansion to be valid C code.

The same form of `#undef' command will cancel definitions with arguments or definitions that don't expect arguments. The `#undef' command has no effect when used on a name not currently defined as a macro.

Redefining Macros

Redefining a macro means defining (with `#define') a name that is already defined as a macro.

A redefinition is trivial if the new definition is transparently identical to the old one. You probably wouldn't deliberately write a trivial redefinition, but they can happen automatically when a header file is included more than once (see section Header Files), so they are accepted silently and without effect.

Nontrivial redefinition is considered likely to be an error, so it provokes a warning message from the preprocessor. However, sometimes it is useful to change the definition of a macro in mid-compilation. You can inhibit the warning by undefining the macro with `#undef' before the second definition.

In order for a redefinition to be trivial, the new definition must exactly match the one already in effect, with two possible exceptions:

- Whitespace may be added or deleted at the beginning or the end.
- Whitespace may be changed in the middle (but not inside strings). However, it may not be eliminated entirely, and it may not be added where there was no whitespace at all.

Recall that a comment counts as whitespace.

Pitfalls and Subtleties of Macros

In this section we describe some special rules that apply to macros and macro expansion, and point out certain cases in which the rules have counterintuitive consequences that you must watch out for.

Improperly Nested Constructs

Recall that when a macro is called with arguments, the arguments are substituted into the macro body and the result is checked, together with the rest of the input file, for more macro calls.

It is possible to piece together a macro call coming partially from the macro body and partially from the actual arguments. For example,

```
#define double(x) (2*(x))
#define call_with_1(x) x(1)
```

would expand `call_with_1 (double)' into `(2*(1))'.

Macro definitions do not have to have balanced parentheses. By writing an unbalanced open parenthesis in a macro body, it is possible to create a macro call that begins inside the macro body but ends outside of it. For example,

```
#define strange(file) fprintf (file, "%s %d",
...
strange(stderr) p, 35)
```

This bizarre example expands to `fprintf (stderr, "%s %d", p, 35)'!

Unintended Grouping of Arithmetic

You may have noticed that in most of the macro definition examples shown above, each occurrence of a macro argument name had parentheses around it. In addition, another pair of parentheses usually surround the entire macro definition. Here is why it is best to write macros that way.

Suppose you define a macro as follows,

```
#define ceil_div(x, y) (x + y - 1) / y
```

whose purpose is to divide, rounding up. (One use for this operation is to compute how many `int' objects are needed to hold a certain number of `char' objects.) Then suppose it is used as follows:

```
a = ceil_div (b & c, sizeof (int));
```

This expands into


```
a = (b & c + sizeof (int) - 1) / sizeof (int);
```

which does not do what is intended. The operator-precedence rules of C make it equivalent to this:

```
a = (b & (c + sizeof (int) - 1)) / sizeof (int);
```

But what we want is this:

```
a = ((b & c) + sizeof (int) - 1)) / sizeof (int);
```

Defining the macro as

```
#define ceil_div(x, y) ((x) + (y) - 1) / (y)
```

provides the desired result.

However, unintended grouping can result in another way. Consider `sizeof ceil_div(1, 2)'. That has the appearance of a C expression that would compute the size of the type of `ceil_div (1, 2)', but in fact it means something very different. Here is what it expands to:

```
sizeof ((1) + (2) - 1) / (2)
```

This would take the size of an integer and divide it by two. The precedence rules have put the division outside the `sizeof' when it was intended to be inside.

Parentheses around the entire macro definition can prevent such problems. Here, then, is the recommended way to define `ceil_div':

```
#define ceil_div(x, y) (((x) + (y) - 1) / (y))
```

Swallowing the Semicolon

Often it is desirable to define a macro that expands into a compound statement. Consider, for example, the following macro, that advances a pointer (the argument `p' says where to find it) across whitespace characters:

```
#define SKIP_SPACES (p, limit)  \
{ register char *lim = (limit); \
  while (p != lim) {            \
    if (*p++ != ' ') {          \
      p--; break; }}}
```

Here Backslash-Newline is used to split the macro definition, which must be a single line, so that it resembles the way such C code would be laid out if not part of a macro definition.

A call to this macro might be `SKIP_SPACES (p, lim)'. Strictly speaking, the call expands to a compound statement, which is a complete statement with no need for a semicolon to end it. But it looks like a function call. So it minimizes confusion if you can use it like a function call, writing a semicolon

afterward, as in `SKIP_SPACES (p, lim);'

But this can cause trouble before `else' statements, because the semicolon is actually a null statement. Suppose you write

```
if (*p != 0)
  SKIP_SPACES (p, lim);
else ...
```

The presence of two statements--the compound statement and a null statement--in between the `if' condition and the `else' makes invalid C code.

The definition of the macro `SKIP_SPACES' can be altered to solve this problem, using a `do ... while' statement. Here is how:

```
#define SKIP_SPACES (p, limit)     \
do { register char *lim = (limit); \
     while (p != lim) {            \
       if (*p++ != ' ') {          \
         p--; break; }}}           \
while (0)
```

Now `SKIP_SPACES (p, lim);' expands into

```
do {...} while (0);
```

which is one statement.

Duplication of Side Effects

Many C programs define a macro `min', for "minimum", like this:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

When you use this macro with an argument containing a side effect, as shown here,

```
next = min (x + y, foo (z));
```

it expands as follows:

```
next = ((x + y) < (foo (z)) ? (x + y) : (foo (z)));
```

where `x + y' has been substituted for `X' and `foo (z)' for `Y'.

The function `foo' is used only once in the statement as it appears in the program, but the expression `foo (z)' has been substituted twice into the macro expansion. As a result, `foo' might be called two times when the statement is executed. If it has side effects or if it takes a long time to compute, the results might not be what you intended. We say that `min' is an unsafe macro.

The best solution to this problem is to define `min' in a way that computes the value of `foo (z)' only once. The C language offers no standard way to do this, but it can be done with GNU C extensions as follows:

```
#define min(X, Y)                       \
({ typeof (X) __x = (X), __y = (Y);   \
   (__x < __y) ? __x : __y; })
```

If you do not wish to use GNU C extensions, the only solution is to be careful when *using* the macro `min'. For example, you can calculate the value of `foo (z)', save it in a variable, and use that variable in `min':

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
...
{
  int tem = foo (z);
  next = min (x + y, tem);
}
```

(where we assume that `foo' returns type `int').

Self-Referential Macros

A self-referential macro is one whose name appears in its definition. A special feature of ANSI Standard C is that the self-reference is not considered a macro call. It is passed into the preprocessor output unchanged.

Let's consider an example:

```
#define foo (4 + foo)
```

where `foo' is also a variable in your program.

Following the ordinary rules, each reference to `foo' will expand into `(4 + foo)'; then this will be rescanned and will expand into `(4 + (4 + foo))'; and so on until it causes a fatal error (memory full) in the preprocessor.

However, the special rule about self-reference cuts this process short after one step, at `(4 + foo)'. Therefore, this macro definition has the possibly useful effect of causing the program to add 4 to the value of `foo' wherever `foo' is referred to.

In most cases, it is a bad idea to take advantage of this feature. A person reading the program who sees that `foo' is a variable will not expect that it is a macro as well. The reader will come across the identifier `foo' in the program and think its value should be that of the variable `foo', whereas in fact the value is four greater.

The special rule for self-reference applies also to indirect self-reference. This is the case where a macro x expands to use a macro `y', and the expansion of `y' refers to the macro `x'. The resulting reference to `x'

comes indirectly from the expansion of `x', so it is a self-reference and is not further expanded. Thus, after

```
#define x (4 + y)
#define y (2 * x)
```

`x' would expand into `(4 + (2 * x))'. Clear?

But suppose `y' is used elsewhere, not from the definition of `x'. Then the use of `x' in the expansion of `y' is not a self-reference because `x' is not "in progress". So it does expand. However, the expansion of `x' contains a reference to `y', and that is an indirect self-reference now because `y' is "in progress". The result is that `y' expands to `(2 * (4 + y))'.

It is not clear that this behavior would ever be useful, but it is specified by the ANSI C standard, so you may need to understand it.

Separate Expansion of Macro Arguments

We have explained that the expansion of a macro, including the substituted actual arguments, is scanned over again for macro calls to be expanded.

What really happens is more subtle: first each actual argument text is scanned separately for macro calls. Then the results of this are substituted into the macro body to produce the macro expansion, and the macro expansion is scanned again for macros to expand.

The result is that the actual arguments are scanned *twice* to expand macro calls in them.

Most of the time, this has no effect. If the actual argument contained any macro calls, they are expanded during the first scan. The result therefore contains no macro calls, so the second scan does not change it. If the actual argument were substituted as given, with no prescan, the single remaining scan would find the same macro calls and produce the same results.

You might expect the double scan to change the results when a self-referential macro is used in an actual argument of another macro (see section Self-Referential Macros): the self-referential macro would be expanded once in the first scan, and a second time in the second scan. But this is not what happens. The self-references that do not expand in the first scan are marked so that they will not expand in the second scan either.

The prescan is not done when an argument is stringified or concatenated. Thus,

```
#define str(s) #s
#define foo 4
str (foo)
```

expands to `"foo"'. Once more, prescan has been prevented from having any noticeable effect.

More precisely, stringification and concatenation use the argument as written, in un-prescanned form. The same actual argument would be used in prescanned form if it is substituted elsewhere without stringification or concatenation.


```
#define str(s) #s lose(s)
#define foo 4
str (foo)
```

expands to `"foo" lose(4)'.

You might now ask, "Why mention the prescan, if it makes no difference? And why not skip it and make the preprocessor faster?" The answer is that the prescan does make a difference in three special cases:

- Nested calls to a macro.
- Macros that call other macros that stringify or concatenate.
- Macros whose expansions contain unshielded commas.

We say that nested calls to a macro occur when a macro's actual argument contains a call to that very macro. For example, if `f' is a macro that expects one argument, `f (f (1))' is a nested pair of calls to `f'. The desired expansion is made by expanding `f (1)' and substituting that into the definition of `f'. The prescan causes the expected result to happen. Without the prescan, `f (1)' itself would be substituted as an actual argument, and the inner use of `f' would appear during the main scan as an indirect self-reference and would not be expanded. Here, the prescan cancels an undesirable side effect (in the medical, not computational, sense of the term) of the special rule for self-referential macros.

But prescan causes trouble in certain other cases of nested macro calls. Here is an example:

```
#define foo  a,b
#define bar(x) lose(x)
#define lose(x) (1 + (x))

bar(foo)
```

We would like `bar(foo)' to turn into `(1 + (foo))', which would then turn into `(1 + (a,b))'. But instead, `bar(foo)' expands into `lose(a,b)', and you get an error because `lose` requires a single argument. In this case, the problem is easily solved by the same parentheses that ought to be used to prevent misnesting of arithmetic operations:

```
#define foo (a,b)
#define bar(x) lose((x))
```

The problem is more serious when the operands of the macro are not expressions; for example, when they are statements. Then parentheses are unacceptable because they would make for invalid C code:

```
#define foo { int a, b; ... }
```

In GNU C you can shield the commas using the `({...})' construct which turns a compound statement into an expression:

```
#define foo ({ int a, b; ... })
```

Or you can rewrite the macro definition to avoid such commas:

```
#define foo { int a; int b; ... }
```

There is also one case where prescan is useful. It is possible to use prescan to expand an argument and then stringify it--if you use two levels of macros. Let's add a new macro `xstr' to the example shown above:

```
#define xstr(s) str(s)
#define str(s) #s
#define foo 4
xstr (foo)
```

This expands into `"4"', not `"foo"'. The reason for the difference is that the argument of `xstr' is expanded at prescan (because `xstr' does not specify stringification or concatenation of the argument). The result of prescan then forms the actual argument for `str'. `str' uses its argument without prescan because it performs stringification; but it cannot prevent or undo the prescanning already done by `xstr'.

Cascaded Use of Macros

A cascade of macros is when one macro's body contains a reference to another macro. This is very common practice. For example,

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
```

This is not at all the same as defining `TABLESIZE' to be `1020'. The `#define' for `TABLESIZE' uses exactly the body you specify--in this case, `BUFSIZE'---and does not check to see whether it too is the name of a macro.

It's only when you *use* `TABLESIZE' that the result of its expansion is checked for more macro names.

This makes a difference if you change the definition of `BUFSIZE' at some point in the source file. `TABLESIZE', defined as shown, will always expand using the definition of `BUFSIZE' that is currently in effect:

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
#undef BUFSIZE
#define BUFSIZE 37
```

Now `TABLESIZE' expands (in two stages) to `37'.

Newlines in Macro Arguments

Traditional macro processing carries forward all newlines in macro arguments into the expansion of the macro. This means that, if some of the arguments are substituted more than once, or not at all, or out of order, newlines can be duplicated, lost, or moved around within the expansion. If the expansion consists of multiple statements, then the effect is to distort the line numbers of some of these statements. The result can be incorrect line numbers, in error messages or displayed in a debugger.

The GNU C preprocessor operating in ANSI C mode adjusts appropriately for multiple use of an argument--the first use expands all the newlines, and subsequent uses of the same argument produce no newlines. But even in this mode, it can produce incorrect line numbering if arguments are used out of order, or not used at all.

Here is an example illustrating this problem:

```
#define ignore_second_arg(a,b,c) a; c

ignore_second_arg (foo (),
                   ignored (),
                   syntax error);
```

The syntax error triggered by the tokens `syntax error' results in an error message citing line four, even though the statement text comes from line five.

Conditionals

In a macro processor, a conditional is a command that allows a part of the program to be ignored during compilation, on some conditions. In the C preprocessor, a conditional can test either an arithmetic expression or whether a name is defined as a macro.

A conditional in the C preprocessor resembles in some ways an `if' statement in C, but it is important to understand the difference between them. The condition in an `if' statement is tested during the execution of your program. Its purpose is to allow your program to behave differently from run to run, depending on the data it is operating on. The condition in a preprocessor conditional command is tested when your program is compiled. Its purpose is to allow different code to be included in the program depending on the situation at the time of compilation.

Why Conditionals are Used

Generally there are three kinds of reason to use a conditional.

- A program may need to use different code depending on the machine or operating system it is to run on. In some cases the code for one operating system may be erroneous on another operating system; for example, it might refer to library routines that do not exist on the other system. When this happens, it is not enough to avoid executing the invalid code: merely having it in the program makes it impossible to link the program and run it. With a preprocessor conditional, the offending

code can be effectively excised from the program when it is not valid.

- You may want to be able to compile the same source file into two different programs. Sometimes the difference between the programs is that one makes frequent time-consuming consistency checks on its intermediate data while the other does not.
- A conditional whose condition is always false is a good way to exclude code from the program but keep it as a sort of comment for future reference.

Most simple programs that are intended to run on only one machine will not need to use preprocessor conditionals.

Syntax of Conditionals

A conditional in the C preprocessor begins with a conditional command: `#if', `#ifdef' or `#ifndef'. See section Conditionals and Macros, for information on `#ifdef' and `#ifndef'; only `#if' is explained here.

The `#if' Command

The `#if' command in its simplest form consists of

```
#if expression
controlled text
#endif /* expression */
```

The comment following the `#endif' is not required, but it is a good practice because it helps people match the `#endif' to the corresponding `#if'. Such comments should always be used, except in short conditionals that are not nested. In fact, you can put anything at all after the `#endif' and it will be ignored by the GNU C preprocessor, but only comments are acceptable in ANSI Standard C.

expression is a C expression of integer type, subject to stringent restrictions. It may contain

- Integer constants, which are all regarded as `long` or `unsigned long`.
- Character constants, which are interpreted according to the character set and conventions of the machine and operating system on which the preprocessor is running. The GNU C preprocessor uses the C data type `char' for these character constants; therefore, whether some character codes are negative is determined by the C compiler used to compile the preprocessor. If it treats `char' as signed, then character codes large enough to set the sign bit will be considered negative; otherwise, no character code is considered negative.
- Arithmetic operators for addition, subtraction, multiplication, division, bitwise operations, shifts, comparisons, and `&&' and `||'.
- Identifiers that are not macros, which are all treated as zero(!).
- Macro calls. All macro calls in the expression are expanded before actual computation of the expression's value begins.

Note that `sizeof' operators and `enum`-type values are not allowed. `enum`-type values, like all other identifiers that are not taken as macro calls and expanded, are treated as zero.

The controlled text inside of a conditional can include preprocessor commands. Then the commands inside the conditional are obeyed only if that branch of the conditional succeeds. The text can also contain other conditional groups. However, the `#if' and `#endif' commands must balance.

The `#else' Command

The `#else' command can be added to a conditional to provide alternative text to be used if the condition is false. This is what it looks like:

```
#if expression
text-if-true
#else /* Not expression */
text-if-false
#endif /* Not expression */
```

If expression is nonzero, and thus the text-if-true is active, then `#else' acts like a failing conditional and the text-if-false is ignored. Contrariwise, if the `#if' conditional fails, the text-if-false is considered included.

The `#elif' Command

One common case of nested conditionals is used to check for more than two possible alternatives. For example, you might have

```
#if X == 1
...
#else /* X != 1 */
#if X == 2
...
#else /* X != 2 */
...
#endif /* X != 2 */
#endif /* X != 1 */
```

Another conditional command, `#elif', allows this to be abbreviated as follows:

```
#if X == 1
...
#elif X == 2
...
#else /* X != 2 and X != 1*/
...
#endif /* X != 2 and X != 1*/
```

`#elif' stands for "else if". Like `#else', it goes in the middle of a `#if'-`#endif' pair and subdivides it; it does not require a matching `#endif' of its own. Like `#if', the `#elif' command includes an expression to

be tested.

The text following the `#elif' is processed only if the original `#if'-condition failed and the `#elif' condition succeeds. More than one `#elif' can go in the same `#if'-`#endif' group. Then the text after each `#elif' is processed only if the `#elif' condition succeeds after the original `#if' and any previous `#elif' commands within it have failed. `#else' is equivalent to `#elif 1', and `#else' is allowed after any number of `#elif' commands, but `#elif' may not follow `#else'.

Keeping Deleted Code for Future Reference

If you replace or delete a part of the program but want to keep the old code around as a comment for future reference, the easy way to do this is to put `#if 0' before it and `#endif' after it.

This works even if the code being turned off contains conditionals, but they must be entire conditionals (balanced `#if' and `#endif').

Conditionals and Macros

Conditionals are useful in connection with macros or assertions, because those are the only ways that an expression's value can vary from one compilation to another. A `#if' command whose expression uses no macros or assertions is equivalent to `#if 1' or `#if 0'; you might as well determine which one, by computing the value of the expression yourself, and then simplify the program.

For example, here is a conditional that tests the expression `BUFSIZE == 1020', where `BUFSIZE' must be a macro.

```
#if BUFSIZE == 1020
  printf ("Large buffers!\n");
#endif /* BUFSIZE is large */
```

(Programmers often wish they could test the size of a variable or data type in `#if', but this does not work. The preprocessor does not understand `sizeof`, or typedef names, or even the type keywords such as `int`.)

The special operator `defined' is used in `#if' expressions to test whether a certain name is defined as a macro. Either `defined name' or `defined (name)' is an expression whose value is 1 if name is defined as macro at the current point in the program, and 0 otherwise. For the `defined' operator it makes no difference what the definition of the macro is; all that matters is whether there is a definition. Thus, for example,

```
#if defined (vax) || defined (ns16000)
```

would include the following code if either of the names `vax' and `ns16000' is defined as a macro. You can test the same condition using assertions (see section Assertions), like this:

```
#if #cpu (vax) || #cpu (ns16000)
```

If a macro is defined and later undefined with `#undef', subsequent use of the `defined' operator retur because the name is no longer defined. If the macro is defined again with another `#define', `defined' recommence returning 1.

Conditionals that test just the definedness of one name are very common, so there are two special short conditional commands for this case.

`#ifdef name`

is equivalent to `#if defined (name)'.

`#ifndef name`

is equivalent to `#if ! defined (name)'.

Macro definitions can vary between compilations for several reasons.

- Some macros are predefined on each kind of machine. For example, on a Vax, the name `vax' is a predefined macro. On other machines, it would not be defined.
- Many more macros are defined by system header files. Different systems and machines define different macros, or give them different values. It is useful to test these macros with conditionals to avoid using a system feature on a machine where it is not implemented.
- Macros are a common way of allowing users to customize a program for different machines or applications. For example, the macro `BUFSIZE' might be defined in a configuration file for your program that is included as a header file in each source file. You would use `BUFSIZE' in a preprocessor conditional in order to generate different code depending on the chosen configuration.
- Macros can be defined or undefined with `-D' and `-U' command options when you compile the program. You can arrange to compile the same source file into two different programs by choosing a macro name to specify which program you want, writing conditionals to test whether or how this macro is defined, and then controlling the state of the macro with compiler command options. See section Invoking the C Preprocessor.

Assertions

Assertions are a more systematic alternative to macros in writing conditionals to test what sort of computer or system the compiled program will run on. Assertions are usually predefined, but you can define them with preprocessor commands or command-line options.

The macros traditionally used to describe the type of target are not classified in any way according to which question they answer; they may indicate a hardware architecture, a particular hardware model, an operating system, a particular version of an operating system, or specific configuration options. These are jumbled together in a single namespace. In contrast, each assertion consists of a named question and an answer. The question is usually called the predicate. An assertion looks like this:

```
#predicate (answer)
```

You must use a properly formed identifier for predicate. The value of answer can be any sequence of words; all characters are significant except for leading and trailing whitespace, and differences in internal whitespace sequences are ignored. Thus, `x + y' is different from `x+y' but equivalent to `x + y'. `)' is not

allowed in an answer.

Here is a conditional to test whether the answer answer is asserted for the predicate predicate:

```
#if #predicate (answer)
```

There may be more than one answer asserted for a given predicate. If you omit the answer, you can test whether *any* answer is asserted for predicate:

```
#if #predicate
```

Most of the time, the assertions you test will be predefined assertions. GNU C provides three predefined predicates: `system`, `cpu`, and `machine`. `system` is for assertions about the type of software, `cpu` describes the type of computer architecture, and `machine` gives more information about the computer. For example, on a GNU system, the following assertions would be true:

```
#system (gnu)
#system (mach)
#system (mach 3)
#system (mach 3.subversion)
#system (hurd)
#system (hurd version)
```

and perhaps others. The alternatives with more or less version information let you ask more or less detailed questions about the type of system software.

On a Unix system, you would find `#system (unix)` and perhaps one of: `#system (aix)`, `#system (bsd)`, `#system (hpux)`, `#system (lynx)`, `#system (mach)`, `#system (posix)`, `#system (svr3)`, `#system (svr4)`, or `#system (xpg4)` with possible version numbers following.

Other values for `system` are `#system (mvs)` and `#system (vms)`.

Portability note: Many Unix C compilers provide only one answer for the `system` assertion: `#system (unix)`, if they support assertions at all. This is less than useful.

An assertion with a multi-word answer is completely different from several assertions with individual single-word answers. For example, the presence of `system (mach 3.0)` does not mean that `system (3.0)` is true. It also does not directly imply `system (mach)`, but in GNU C, that last will normally be asserted as well.

The current list of possible assertion values for `cpu` is: `#cpu (a29k)`, `#cpu (alpha)`, `#cpu (arm)`, `#cpu (clipper)`, `#cpu (convex)`, `#cpu (elxsi)`, `#cpu (tron)`, `#cpu (h8300)`, `#cpu (i370)`, `#cpu (i386)`, `#cpu (i860)`, `#cpu (i960)`, `#cpu (m68k)`, `#cpu (m88k)`, `#cpu (mips)`, `#cpu (ns32k)`, `#cpu (hppa)`, `#cpu (pyr)`, `#cpu (ibm032)`, `#cpu (rs6000)`, `#cpu (sh)`, `#cpu (sparc)`, `#cpu (spur)`, `#cpu (tahoe)`, `#cpu (vax)`, `#cpu (we32000)`.

You can create assertions within a C program using `#assert', like this:

```
#assert predicate (answer)
```

(Note the absence of a `#' before predicate.)

Each time you do this, you assert a new true answer for predicate. Asserting one answer does not invalidate previously asserted answers; they all remain true. The only way to remove an assertion is with `#unassert'. `#unassert' has the same syntax as `#assert'. You can also remove all assertions about predicate like this:

```
#unassert predicate
```

You can also add or cancel assertions using command options when you run `gcc` or `cpp`. See section Invoking the C Preprocessor.

The `#error' and `#warning' Commands

The command `#error' causes the preprocessor to report a fatal error. The rest of the line that follows `#error' is used as the error message.

You would use `#error' inside of a conditional that detects a combination of parameters which you know the program does not properly support. For example, if you know that the program will not run properly on a Vax, you might write

```
#ifdef vax
#error Won't work on Vaxen.  See comments at get_last_object.
#endif
```

See section Nonstandard Predefined Macros, for why this works.

If you have several configuration parameters that must be set up by the installation in a consistent way, you can use conditionals to detect an inconsistency and report it with `#error'. For example,

```
#if HASH_TABLE_SIZE % 2 == 0 || HASH_TABLE_SIZE % 3 == 0 \
    || HASH_TABLE_SIZE % 5 == 0
#error HASH_TABLE_SIZE should not be divisible by a small prime
#endif
```

The command `#warning' is like the command `#error', but causes the preprocessor to issue a warning and continue preprocessing. The rest of the line that follows `#warning' is used as the warning message.

You might use `#warning' in obsolete header files, with a message directing the user to the header file which should be used instead.

Combining Source Files

One of the jobs of the C preprocessor is to inform the C compiler of where each line of C code came from: which source file and which line number.

C code can come from multiple source files if you use `#include'; both `#include' and the use of conditionals and macros can cause the line number of a line in the preprocessor output to be different from the line's number in the original source file. You will appreciate the value of making both the C compiler (in error messages) and symbolic debuggers such as GDB use the line numbers in your source file.

The C preprocessor builds on this feature by offering a command by which you can control the feature explicitly. This is useful when a file for input to the C preprocessor is the output from another program such as the `bison` parser generator, which operates on another file that is the true source file. Parts of the output from `bison` are generated from scratch, other parts come from a standard parser file. The rest are copied nearly verbatim from the source file, but their line numbers in the `bison` output are not the same as their original line numbers. Naturally you would like compiler error messages and symbolic debuggers to know the original source file and line number of each line in the `bison` input.

`bison` arranges this by writing `#line' commands into the output file. `#line' is a command that specifies the original line number and source file name for subsequent input in the current preprocessor input file. `#line' has three variants:

`#line linenum`

> Here linenum is a decimal integer constant. This specifies that the line number of the following line of input, in its original source file, was linenum.

`#line linenum filename`

> Here linenum is a decimal integer constant and filename is a string constant. This specifies that the following line of input came originally from source file filename and its line number there was linenum. Keep in mind that filename is not just a file name; it is surrounded by doublequote characters so that it looks like a string constant.

`#line anything else`

> anything else is checked for macro calls, which are expanded. The result should be a decimal integer constant followed optionally by a string constant, as described above.

`#line' commands alter the results of the `__FILE__' and `__LINE__' predefined macros from that point on. See section Standard Predefined Macros.

The output of the preprocessor (which is the input for the rest of the compiler) contains commands that look much like `#line' commands. They start with just `#' instead of `#line', but this is followed by a line number and file name as in `#line'. See section C Preprocessor Output.

Miscellaneous Preprocessor Commands

This section describes three additional preprocessor commands. They are not very useful, but are mentioned for completeness.

The null command consists of a `#' followed by a Newline, with only whitespace (including comments) in between. A null command is understood as a preprocessor command but has no effect on the preprocessor output. The primary significance of the existence of the null command is that an input line consisting of just a `#' will produce no output, rather than a line of output containing just a `#'. Supposedly some old C programs contain such lines.

The ANSI standard specifies that the `#pragma' command has an arbitrary, implementation-defined effect. In the GNU C preprocessor, `#pragma' commands are not used, except for `#pragma once' (see section Once-Only Include Files). However, they are left in the preprocessor output, so they are available to the compilation pass.

The `#ident' command is supported for compatibility with certain other systems. It is followed by a line of text. On some systems, the text is copied into a special place in the object file; on most systems, the text is ignored and this command has no effect. Typically `#ident' is only used in header files supplied with those systems where it is meaningful.

C Preprocessor Output

The output from the C preprocessor looks much like the input, except that all preprocessor command lines have been replaced with blank lines and all comments with spaces. Whitespace within a line is not altered; however, a space is inserted after the expansions of most macro calls.

Source file name and line number information is conveyed by lines of the form

```
# linenum filename flags
```

which are inserted as needed into the middle of the input (but never within a string or character constant). Such a line means that the following line originated in file filename at line linenum.

After the file name comes zero or more flags, which are `1', `2' or `3'. If there are multiple flags, spaces separate them. Here is what the flags mean:

`1'
: This indicates the start of a new file.

`2'
: This indicates returning to a file (after having included another file).

`3'
: This indicates that the following text comes from a system header file, so certain warnings should be suppressed.

Invoking the C Preprocessor

Most often when you use the C preprocessor you will not have to invoke it explicitly: the C compiler will do so automatically. However, the preprocessor is sometimes useful individually.

The C preprocessor expects two file names as arguments, infile and outfile. The preprocessor reads infile together with any other files it specifies with `#include'. All the output generated by the combined input files is written in outfile.

Either infile or outfile may be `-', which as infile means to read from standard input and as outfile means to write to standard output. Also, if outfile or both file names are omitted, the standard output and standard input are used for the omitted file names.

Here is a table of command options accepted by the C preprocessor. These options can also be given when compiling a C program; they are passed along automatically to the preprocessor when it is invoked by the compiler.

`-P'

Inhibit generation of `#'-lines with line-number information in the output from the preprocessor (see section C Preprocessor Output). This might be useful when running the preprocessor on something that is not C code and will be sent to a program which might be confused by the `#'-lines.

`-C'

Do not discard comments: pass them through to the output file. Comments appearing in arguments of a macro call will be copied to the output before the expansion of the macro call.

`-traditional'

Try to imitate the behavior of old-fashioned C, as opposed to ANSI C.

`'

Traditional macro expansion pays no attention to singlequote or doublequote characters; macro argument symbols are replaced by the argument values even when they appear within apparent string or character constants.

`'

Traditionally, it is permissible for a macro expansion to end in the middle of a string or character constant. The constant continues into the text surrounding the macro call.

`'

However, traditionally the end of the line terminates a string or character constant, with no error.

`'

In traditional C, a comment is equivalent to no text at all. (In ANSI C, a comment counts as whitespace.)

`'

Traditional C does not have the concept of a "preprocessing number". It considers `1.0e+4' to be three tokens: `1.0e', `+', and `4'.

`'

A macro is not suppressed within its own definition, in traditional C. Thus, any macro that is used recursively inevitably causes an error.

`'

The character `#' has no special meaning within a macro definition in traditional C.

`'

In traditional C, the text at the end of a macro expansion can run together with the text after the macro call, to produce a single token. (This is impossible in ANSI C.)

`'

Traditionally, `\' inside a macro argument suppresses the syntactic significance of the following character.

`-trigraphs'

Process ANSI standard trigraph sequences. These are three-character sequences, all starting with `??', that are defined by ANSI C to stand for single characters. For example, `??/' stands for `\', so `'??/n" is a character constant for a newline. Strictly speaking, the GNU C preprocessor does not support all programs in ANSI Standard C unless `-trigraphs' is used, but if you ever notice the difference it will be with relief.

You don't want to know any more about trigraphs.

`-pedantic'

Issue warnings required by the ANSI C standard in certain cases such as when text other than a comment follows `#else' or `#endif'.

`-pedantic-errors'

Like `-pedantic', except that errors are produced rather than warnings.

`-Wtrigraphs'

Warn if any trigraphs are encountered (assuming they are enabled).

`-Wcomment'

Warn whenever a comment-start sequence `/*' appears in a comment.

`-Wall'

Requests both `-Wtrigraphs' and `-Wcomment' (but not `-Wtraditional').

`-Wtraditional'

Warn about certain constructs that behave differently in traditional and ANSI C.

`-I directory'

Add the directory directory to the end of the list of directories to be searched for header files (see section The `#include' Command). This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one `-I' option, the directories are scanned in left-to-right order; the standard system directories come after.

`-I-'

Any directories specified with `-I' options before the `-I-' option are searched only for the case of `#include "file"'; they are not searched for `#include <file>'.

If additional directories are specified with `-I' options after the `-I-', these directories are searched for all `#include' commands.

In addition, the `-I-' option inhibits the use of the current directory as the first search directory for `#include "file"'. Therefore, the current directory is searched only if it is requested explicitly with `-I.'. Specifying both `-I-' and `-I.' allows you to control precisely which directories are searched before the current one and which are searched after.

`-nostdinc'

Do not search the standard system directories for header files. Only the directories you have specified with `-I' options (and the current directory, if appropriate) are searched.

`-nostdinc++'

Do not search for header files in the C++-specific standard directories, but do still search the other standard directories. (This option is used when building libg++.)

`-D name'

Predefine name as a macro, with definition `1'.

`-D name=definition'

Predefine name as a macro, with definition definition. There are no restrictions on the contents of definition, but if you are invoking the preprocessor from a shell or shell-like program you may need to use the shell's quoting syntax to protect characters such as spaces that have a meaning in the shell syntax. If you use more than one `-D' for the same name, the rightmost definition takes effect.

`-U name'

Do not predefine name. If both `-U' and `-D' are specified for one name, the `-U' beats the `-D' and the name is not predefined.

`-undef'

Do not predefine any nonstandard macros.

`-A predicate(answer)'

Make an assertion with the predicate predicate and answer answer. See section Assertions.

You can use `-A-' to disable all predefined assertions; it also undefines all predefined macros that identify the type of target system.

`-dM'

Instead of outputting the result of preprocessing, output a list of `#define' commands for all the macros defined during the execution of the preprocessor, including predefined macros. This gives you a way of finding out what is predefined in your version of the preprocessor; assuming you have no file `foo.h', the command

```
touch foo.h; cpp -dM foo.h
```

will show the values of any predefined macros.

`-dD'

Like `-dM' except in two respects: it does *not* include the predefined macros, and it outputs *both* the `#define' commands and the result of preprocessing. Both kinds of output go to the standard output file.

`-M'

Instead of outputting the result of preprocessing, output a rule suitable for `make` describing the dependencies of the main source file. The preprocessor outputs one `make` rule containing the object file name for that source file, a colon, and the names of all the included files. If there are many included files then the rule is split into several lines using `\'-newline.

This feature is used in automatic updating of makefiles.

`-MM'

Like `-M' but mention only the files included with `#include "file"'. System header files included with `#include <file>' are omitted.

`-MD'

Like `-M' but the dependency information is written to files with names made by replacing `.c' with `.d' at the end of the input file names. This is in addition to compiling the file as specified---`-MD' does not inhibit ordinary compilation the way `-M' does.

In Mach, you can use the utility `md` to merge the `.d' files into a single dependency file suitable for using with the `make' command.

`-MMD'

Like `-MD' except mention only user header files, not system header files.

`-H'

Print the name of each header file used, in addition to other normal activities.

`-imacros file'

Process file as input, discarding the resulting output, before processing the regular input file. Because the output generated from file is discarded, the only effect of `-imacros file' is to make the macros defined in file available for use in the main input.

`-include file'

Process file as input, and include all the resulting output, before processing the regular input file.

`-idirafter dir'

Add the directory dir to the second include path. The directories on the second include path are searched when a header file is not found in any of the directories in the main include path (the one that `-I' adds to).

`-iprefix prefix'

Specify prefix as the prefix for subsequent `-iwithprefix' options.

`-iwithprefix dir'

Add a directory to the second include path. The directory's name is made by concatenating prefix and dir, where prefix was specified previously with `-iprefix'.

`-lang-c'

`-lang-c++'

`-lang-objc'

`-lang-objc++'

Specify the source language. `-lang-c++' makes the preprocessor handle C++ comment syntax (comments may begin with `//', in which case they end at end of line), and includes extra default include directories for C++; and `-lang-objc' enables the Objective C `#import' command. `-lang-c' explicitly turns off both of these extensions, and `-lang-objc++' enables both.

These options are generated by the compiler driver `gcc`, but not passed from the `gcc' command line.

`-lint'

Look for commands to the program checker `lint` embedded in comments, and emit them preceded by `#pragma lint'. For example, the comment `/* NOTREACHED */' becomes `#pragma lint NOTREACHED'.

This option is available only when you call `cpp` directly; `gcc` will not pass it from its command line.

`-$'

Forbid the use of `$' in identifiers. This is required for ANSI conformance. `gcc` automatically supplies this option to the preprocessor if you specify `-ansi', but `gcc` doesn't recognize the `-$' option itself--to use it without the other effects of `-ansi', you must call the preprocessor directly.

Concept Index

a

c

h

i

l

m

n

o

p

r

s

t

u

Index of Commands, Macros and Options

#

- #pragma
- #pragma once
- #unassert
- #warning

-

- -$
- -A
- -C
- -D
- -dD
- -dM
- -H
- -I
- -idirafter
- -imacros
- -include
- -iprefix
- -iwithprefix
- -lang-c
- -lang-c++
- -lang-objc
- -lang-objc++
- -M
- -MD
- -MM
- -MMD
- -nostdinc
- -nostdinc++
- -P
- -pedantic
- -pedantic-errors
- -traditional

_

b

d

m

n

- ns32000

p

- pyr

s

- sequent
- sun
- system header files

u

- unix

v

- vax

Go to the next section.

Transformations Made Globally

Most C preprocessor features are inactive unless you give specific directives to request their use. (Preprocessing directives are lines starting with `#'; see section Preprocessing Directives). But there are three transformations that the preprocessor always makes on all the input it receives, even in the absence of directives.

- All C comments are replaced with single spaces.
- Backslash-Newline sequences are deleted, no matter where. This feature allows you to break long lines for cosmetic purposes without changing their meaning.
- Predefined macro names are replaced with their expansions (see section Predefined Macros).

The first two transformations are done *before* nearly all other parsing and before preprocessing directives are recognized. Thus, for example, you can split a line cosmetically with Backslash-Newline anywhere (except when trigraphs are in use; see below).

```
/*
*/ # /*
*/ defi\
ne FO\
O 10\
20
```

is equivalent into `#define FOO 1020'. You can split even an escape sequence with Backslash-Newline. For example, you can split `"foo\bar"` between the `\' and the `b' to get

```
"foo\\
bar"
```

This behavior is unclean: in all other contexts, a Backslash can be inserted in a string constant as an ordinary character by writing a double Backslash, and this creates an exception. But the ANSI C standard requires it. (Strict ANSI C does not allow Newlines in string constants, so they do not consider this a problem.)

But there are a few exceptions to all three transformations.

- C comments and predefined macro names are not recognized inside a `#include' directive in which the file name is delimited with `<' and `>'.
- C comments and predefined macro names are never recognized within a character or string constant. (Strictly speaking, this is the rule, not an exception, but it is worth noting here anyway.)
- Backslash-Newline may not safely be used within an ANSI "trigraph". Trigraphs are converted before Backslash-Newline is deleted. If you write what looks like a trigraph with a Backslash-Newline inside, the Backslash-Newline is deleted as usual, but it is then too late to recognize the trigraph.

This exception is relevant only if you use the `-trigraphs' option to enable trigraph processing section Invoking the C Preprocessor.

Preprocessing Directives

Most preprocessor features are active only if you use preprocessing directives to request their use.

Preprocessing directives are lines in your program that start with `#'. The `#' is followed by an identifier that is the directive name. For example, `#define' is the directive that defines a macro. Whitespace is also allowed before and after the `#'.

The set of valid directive names is fixed. Programs cannot define new preprocessing directives.

Some directive names require arguments; these make up the rest of the directive line and must be separated from the directive name by whitespace. For example, `#define' must be followed by a macro name and the intended expansion of the macro. See section Simple Macros.

A preprocessing directive cannot be more than one line in normal circumstances. It may be split cosmetically with Backslash-Newline, but that has no effect on its meaning. Comments containing Newlines can also divide the directive into multiple lines, but the comments are changed to Spaces before the directive is interpreted. The only way a significant Newline can occur in a preprocessing directive is within a string constant or character constant. Note that most C compilers that might be applied to the output from the preprocessor do not accept string or character constants containing Newlines.

The `#' and the directive name cannot come from a macro expansion. For example, if `foo' is defined as a macro expanding to `define', that does not make `#foo' a valid preprocessing directive.

Header Files

A header file is a file containing C declarations and macro definitions (see section Macros) to be shared between several source files. You request the use of a header file in your program with the C preprocessing directive `#include'.

Uses of Header Files

Header files serve two kinds of purposes.

- System header files declare the interfaces to parts of the operating system. You include them in your program to supply the definitions and declarations you need to invoke system calls and libraries.
- Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.

Including a header file produces the same results in C compilation as copying the header file into each

source file that needs it. But such copying would be time-consuming and error-prone. With a header the related declarations appear in only one place. If they need to be changed, they can be changed in place, and programs that include the header file will automatically use the new version when next recompiled. The header file eliminates the labor of finding and changing all the copies as well as the risk that a failure to find one copy will result in inconsistencies within a program.

The usual convention is to give header files names that end with `.h'. Avoid unusual characters in header file names, as they reduce portability.

The `#include' Directive

Both user and system header files are included using the preprocessing directive `#include'. It has three variants:

`#include <file>`

This variant is used for system header files. It searches for a file named file in a list of directories specified by you, then in a standard list of system directories. You specify directories to search for header files with the command option `-I' (see section Invoking the C Preprocessor). The option `-nostdinc' inhibits searching the standard system directories; in this case only the directories you specify are searched.

The parsing of this form of `#include' is slightly special because comments are not recognized within the `<...>'. Thus, in `#include <x/*y>' the `/*' does not start a comment and the directive specifies inclusion of a system header file named `x/*y'. Of course, a header file with such a name is unlikely to exist on Unix, where shell wildcard features would make it hard to manipulate.

The argument file may not contain a `>' character. It may, however, contain a `<' character.

`#include "file"`

This variant is used for header files of your own program. It searches for a file named file first in the current directory, then in the same directories used for system header files. The current directory is the directory of the current input file. It is tried first because it is presumed to be the location of the files that the current input file refers to. (If the `-I-' option is used, the special treatment of the current directory is inhibited.)

The argument file may not contain `"' characters. If backslashes occur within file, they are considered ordinary text characters, not escape characters. None of the character escape sequences appropriate to string constants in C are processed. Thus, `#include "x\n\\y"' specifies a filename containing three backslashes. It is not clear why this behavior is ever useful, but the ANSI standard specifies it.

`#include anything else`

This variant is called a computed #include. Any `#include' directive whose argument does not fit the above two forms is a computed include. The text anything else is checked for macro calls, which are expanded (see section Macros). When this is done, the result must fit one of the above two variants--in particular, the expanded text must in the end be surrounded by either quotes or angle braces.

This feature allows you to define a macro which controls the file name to be used at a later poi the program. One application of this is to allow a site-specific configuration file for your progr to specify the names of the system include files to be used. This can help in porting the program to various operating systems in which the necessary system header files are found in different places.

How `#include' Works

The `#include' directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the `#include' directive. For example, given a header file `header.h' as follows,

```
char *test ();
```

and a main program called `program.c' that uses the header file, like this,

```
int x;
#include "header.h"

main ()
{
  printf (test ());
}
```

the output generated by the C preprocessor for `program.c' as input would be

```
int x;
char *test ();

main ()
{
  printf (test ());
}
```

Included files are not limited to declarations and macro definitions; those are merely the typical uses. Any fragment of a C program can be included from another file. The include file could even contain the beginning of a statement that is concluded in the containing file, or the end of a statement that was started in the including file. However, a comment or a string or character constant may not start in the included file and finish in the including file. An unterminated comment, string constant or character constant in an included file is considered to end (with an error message) at the end of the file.

It is possible for a header file to begin or end a syntactic unit such as a function definition, but that would be very confusing, so don't do it.

The line following the `#include' directive is always treated as a separate line by the C preprocessor even if the included file lacks a final newline.

Once-Only Include Files

Very often, one header file includes another. It can easily result that a certain header file is included more than once. This may lead to errors, if the header file defines structure types or typedefs, and is certainly wasteful. Therefore, we often wish to prevent multiple inclusion of a header file.

The standard way to do this is to enclose the entire real contents of the file in a conditional, like this:

```
#ifndef FILE_FOO_SEEN
#define FILE_FOO_SEEN

the entire file

#endif /* FILE_FOO_SEEN */
```

The macro `FILE_FOO_SEEN` indicates that the file has been included once already. In a user header file, the macro name should not begin with `_'. In a system header file, this name should begin with `__' to avoid conflicts with user programs. In any kind of header file, the macro name should contain the name of the file and some additional text, to avoid conflicts with other header files.

The GNU C preprocessor is programmed to notice when a header file uses this particular construct and handle it efficiently. If a header file is contained entirely in a `#ifndef' conditional, then it records that fact. If a subsequent `#include' specifies the same file, and the macro in the `#ifndef' is already defined, then the file is entirely skipped, without even reading it.

There is also an explicit directive to tell the preprocessor that it need not include a file more than once. This is called `#pragma once', and was used *in addition to* the `#ifndef' conditional around the contents of the header file. `#pragma once' is now obsolete and should not be used at all.

In the Objective C language, there is a variant of `#include' called `#import' which includes a file, but does so at most once. If you use `#import' *instead of* `#include', then you don't need the conditionals inside the header file to prevent multiple execution of the contents.

`#import' is obsolete because it is not a well designed feature. It requires the users of a header file--the applications programmers--to know that a certain header file should only be included once. It is much better for the header file's implementor to write the file so that users don't need to know this. Using `#ifndef' accomplishes this goal.

Inheritance and Header Files

Inheritance is what happens when one object or file derives some of its contents by virtual copying from another object or file. In the case of C header files, inheritance means that one header file includes another header file and then replaces or adds something.

If the inheriting header file and the base header file have different names, then inheritance is straightforward: simply write `#include "base"' in the inheriting file.

Sometimes it is necessary to give the inheriting file the same name as the base file. This is less

straightforward.

For example, suppose an application program uses the system header file `sys/signal.h', but the version of `/usr/include/sys/signal.h' on a particular system doesn't do what the application program expects. It might be convenient to define a "local" version, perhaps under the name `/usr/local/include/sys/signal.h', to override or add to the one supplied by the system.

You can do this by using the option `-I.' for compilation, and writing a file `sys/signal.h' that does what the application program expects. But making this file include the standard `sys/signal.h' is not so easy--writing `#include <sys/signal.h>' in that file doesn't work, because it includes your own version of the file, not the standard system version. Used in that file itself, this leads to an infinite recursion and a fatal error in compilation.

`#include </usr/include/sys/signal.h>' would find the proper file, but that is not clean, since it makes an assumption about where the system header file is found. This is bad for maintenance, since it means that any change in where the system's header files are kept requires a change somewhere else.

The clean way to solve this problem is to use `#include_next', which means, "Include the *next* file with this name." This directive works like `#include' except in searching for the specified file: it starts searching the list of header file directories *after* the directory in which the current file was found.

Suppose you specify `-I /usr/local/include', and the list of directories to search also includes `/usr/include'; and suppose that both directories contain a file named `sys/signal.h'. Ordinary `#include <sys/signal.h>' finds the file under `/usr/local/include'. If that file contains `#include_next <sys/signal.h>', it starts searching after that directory, and finds the file in `/usr/include'.

Macros

A macro is a sort of abbreviation which you can define once and then use later. There are many complicated features associated with macros in the C preprocessor.

Simple Macros

A simple macro is a kind of abbreviation. It is a name which stands for a fragment of code. Some people refer to these as manifest constants.

Before you can use a macro, you must define it explicitly with the `#define' directive. `#define' is followed by the name of the macro and then the code it should be an abbreviation for. For example,

```
#define BUFFER_SIZE 1020
```

defines a macro named `BUFFER_SIZE' as an abbreviation for the text `1020'. If somewhere after this `#define' directive there comes a C statement of the form

```
foo = (char *) xmalloc (BUFFER_SIZE);
```

then the C preprocessor will recognize and expand the macro `BUFFER_SIZE', resulting in

```
foo = (char *) xmalloc (1020);
```

The use of all upper case for macro names is a standard convention. Programs are easier to read when it is possible to tell at a glance which names are macros.

Normally, a macro definition must be a single line, like all C preprocessing directives. (You can split a long macro definition cosmetically with Backslash-Newline.) There is one exception: Newlines can be included in the macro definition if within a string or character constant. This is because it is not possible for a macro definition to contain an unbalanced quote character; the definition automatically extends to include the matching quote character that ends the string or character constant. Comments within a macro definition may contain Newlines, which make no difference since the comments are entirely replaced with Spaces regardless of their contents.

Aside from the above, there is no restriction on what can go in a macro body. Parentheses need not balance. The body need not resemble valid C code. (But if it does not, you may get error messages from the C compiler when you use the macro.)

The C preprocessor scans your program sequentially, so macro definitions take effect at the place you write them. Therefore, the following input to the C preprocessor

```
foo = X;
#define X 4
bar = X;
```

produces as output

```
foo = X;

bar = 4;
```

After the preprocessor expands a macro name, the macro's definition body is appended to the front of the remaining input, and the check for macro calls continues. Therefore, the macro body can contain calls to other macros. For example, after

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
```

the name `TABLESIZE' when used in the program would go through two stages of expansion, resulting ultimately in `1020'.

This is not at all the same as defining `TABLESIZE' to be `1020'. The `#define' for `TABLESIZE' uses exactly the body you specify--in this case, `BUFSIZE'---and does not check to see whether it too is the name of a macro. It's only when you *use* `TABLESIZE' that the result of its expansion is checked for more macro names. See section Cascaded Use of Macros.

Macros with Arguments

A simple macro always stands for exactly the same text, each time it is used. Macros can be more flexible when they accept arguments. Arguments are fragments of code that you supply each time the macro is used. These fragments are included in the expansion of the macro according to the directions in the macro definition. A macro that accepts arguments is called a function-like macro because the syntax for using it looks like a function call.

To define a macro that uses arguments, you write a `#define' directive with a list of argument names in parentheses after the name of the macro. The argument names may be any valid C identifiers, separated by commas and optionally whitespace. The open-parenthesis must follow the macro name immediately, with no space in between.

For example, here is a macro that computes the minimum of two numeric values, as it is defined in many C programs:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

(This is not the best way to define a "minimum" macro in GNU C. See section Duplication of Side Effects, for more information.)

To use a macro that expects arguments, you write the name of the macro followed by a list of actual arguments in parentheses, separated by commas. The number of actual arguments you give must match the number of arguments the macro expects. Examples of use of the macro `min' include `min (1, 2)' and `min (x + 28, *p)'.

The expansion text of the macro depends on the arguments you use. Each of the argument names of the macro is replaced, throughout the macro definition, with the corresponding actual argument. Using the same macro `min' defined above, `min (1, 2)' expands into

```
((1) < (2) ? (1) : (2))
```

where `1' has been substituted for `X' and `2' for `Y'.

Likewise, `min (x + 28, *p)' expands into

```
((x + 28) < (*p) ? (x + 28) : (*p))
```

Parentheses in the actual arguments must balance; a comma within parentheses does not end an argument. However, there is no requirement for brackets or braces to balance, and they do not prevent a comma from separating arguments. Thus,

```
macro (array[x = y, x + 1])
```

passes two arguments to `macro`: `array[x = y' and `x + 1]'. If you want to supply `array[x = y, x + 1]' as an argument, you must write it as `array[(x = y, x + 1)]', which is equivalent C code.

After the actual arguments are substituted into the macro body, the entire result is appended to the front

of the remaining input, and the check for macro calls continues. Therefore, the actual arguments can contain calls to other macros, either with or without arguments, or even to the same macro. The macro body can also contain calls to other macros. For example, `min (min (a, b), c)' expands into this text:

```
((((a) < (b) ? (a) : (b))) < (c)
 ? (((a) < (b) ? (a) : (b)))
 : (c))
```

(Line breaks shown here for clarity would not actually be generated.)

If a macro `foo` takes one argument, and you want to supply an empty argument, you must write at least some whitespace between the parentheses, like this: `foo ( )'. Just `foo ()' is providing no arguments, which is an error if `foo` expects an argument. But `foo0 ()' is the correct way to call a macro defined to take zero arguments, like this:

```
#define foo0() ...
```

If you use the macro name followed by something other than an open-parenthesis (after ignoring any spaces, tabs and comments that follow), it is not a call to the macro, and the preprocessor does not change what you have written. Therefore, it is possible for the same name to be a variable or function in your program as well as a macro, and you can choose in each instance whether to refer to the macro (if an actual argument list follows) or the variable or function (if an argument list does not follow).

Such dual use of one name could be confusing and should be avoided except when the two meanings are effectively synonymous: that is, when the name is both a macro and a function and the two have similar effects. You can think of the name simply as a function; use of the name for purposes other than calling it (such as, to take the address) will refer to the function, while calls will expand the macro and generate better but equivalent code. For example, you can use a function named `min' in the same source file that defines the macro. If you write `&min' with no argument list, you refer to the function. If you write `min (x, bb)', with an argument list, the macro is expanded. If you write `(min) (a, bb)', where the name `min' is not followed by an open-parenthesis, the macro is not expanded, so you wind up with a call to the function `min'.

You may not define the same name as both a simple macro and a macro with arguments.

In the definition of a macro with arguments, the list of argument names must follow the macro name immediately with no space in between. If there is a space after the macro name, the macro is defined as taking no arguments, and all the rest of the line is taken to be the expansion. The reason for this is that it is often useful to define a macro that takes no arguments and whose definition begins with an identifier in parentheses. This rule about spaces makes it possible for you to do either this:

```
#define FOO(x) - 1 / (x)
```

(which defines `FOO' to take an argument and expand into minus the reciprocal of that argument) or this:

```
#define BAR (x) - 1 / (x)
```

(which defines `BAR' to take no argument and always expand into `(x) - 1 / (x)').

Note that the *uses* of a macro with arguments can have spaces before the left parenthesis; it's the *definition* where it matters whether there is a space.

Predefined Macros

Several simple macros are predefined. You can use them without giving definitions for them. They fall into two classes: standard macros and system-specific macros.

Standard Predefined Macros

The standard predefined macros are available with the same meanings regardless of the machine or operating system on which you are using GNU C. Their names all start and end with double underscores. Those preceding `__GNUC__` in this table are standardized by ANSI C; the rest are GNU C extensions.

`__FILE__`

This macro expands to the name of the current input file, in the form of a C string constant. The precise name returned is the one that was specified in `#include' or as the input file name argument.

`__LINE__`

This macro expands to the current input line number, in the form of a decimal integer constant. While we call it a predefined macro, it's a pretty strange macro, since its "definition" changes with each new line of source code.

This and `__FILE__' are useful in generating an error message to report an inconsistency detected by the program; the message can state the source line at which the inconsistency was detected. For example,

```
fprintf (stderr, "Internal error: "
                 "negative string length "
                 "%d at %s, line %d.",
         length, __FILE__, __LINE__);
```

A `#include' directive changes the expansions of `__FILE__' and `__LINE__' to correspond to the included file. At the end of that file, when processing resumes on the input file that contained the `#include' directive, the expansions of `__FILE__' and `__LINE__' revert to the values they had before the `#include' (but `__LINE__' is then incremented by one as processing moves to the line after the `#include').

The expansions of both `__FILE__' and `__LINE__' are altered if a `#line' directive is used. See section Combining Source Files.

`__DATE__`

This macro expands to a string constant that describes the date on which the preprocessor is being run. The string constant contains eleven characters and looks like `"Jan 29 1987"' or `"Apr 1 1905"'.

`__TIME__`

This macro expands to a string constant that describes the time at which the preprocessor is being run. The string constant contains eight characters and looks like `'"23:59:01"'.

`__STDC__`

This macro expands to the constant 1, to signify that this is ANSI Standard C. (Whether that is actually true depends on what C compiler will operate on the output from the preprocessor.)

`__STDC_VERSION__`

This macro expands to the C Standard's version number, a long integer constant of the form `yyyymmL' where yyyy and mm are the year and month of the Standard version. This signifies which version of the C Standard the preprocessor conforms to. Like `__STDC__', whether this version number is accurate for the entire implementation depends on what C compiler will operate on the output from the preprocessor.

`__GNUC__`

This macro is defined if and only if this is GNU C. This macro is defined only when the entire GNU C compiler is in use; if you invoke the preprocessor directly, `__GNUC__' is undefined. The value identifies the major version number of GNU CC (`1' for GNU CC version 1, which is now obsolete, and `2' for version 2).

`__GNUC_MINOR__`

The macro contains the minor version number of the compiler. This can be used to work around differences between different releases of the compiler (for example, if gcc 2.6.3 is known to support a feature, you can test for `__GNUC__ > 2 || (__GNUC__ == 2 && __GNUC_MINOR__ >= 6)`). The last number, `3' in the example above, denotes the bugfix level of the compiler; no macro contains this value.

`__GNUG__`

The GNU C compiler defines this when the compilation language is C++; use `__GNUG__' to distinguish between GNU C and GNU C++.

`__cplusplus`

The draft ANSI standard for C++ used to require predefining this variable. Though it is no longer required, GNU C++ continues to define it, as do other popular C++ compilers. You can use `__cplusplus' to test whether a header is compiled by a C compiler or a C++ compiler.

`__STRICT_ANSI__`

This macro is defined if and only if the `-ansi' switch was specified when GNU C was invoked. Its definition is the null string. This macro exists primarily to direct certain GNU header files not to define certain traditional Unix constructs which are incompatible with ANSI C.

`__BASE_FILE__`

This macro expands to the name of the main input file, in the form of a C string constant. This is the source file that was specified as an argument when the C compiler was invoked.

`__INCLUDE_LEVEL__`

This macro expands to a decimal integer constant that represents the depth of nesting in include files. The value of this macro is incremented on every `#include' directive and decremented at every end of file. For input files specified by command line arguments, the nesting level is zero.

`__VERSION__`

This macro expands to a string which describes the version number of GNU C. The string is normally a sequence of decimal numbers separated by periods, such as `"2.6.0"'. The only reasonable use of this macro is to incorporate it into a string constant.

`__OPTIMIZE__`

This macro is defined in optimizing compilations. It causes certain GNU header files to define alternative macro definitions for some system library functions. It is unwise to refer to or test the definition of this macro unless you make very sure that programs will execute with the same effect regardless.

`__CHAR_UNSIGNED__`

This macro is defined if and only if the data type `char` is unsigned on the target machine. It exists to cause the standard header file `` `limit.h' `` to work correctly. It is bad practice to refer to this macro yourself; instead, refer to the standard macros defined in `` `limit.h' ``. The preprocessor uses this macro to determine whether or not to sign-extend large character constants written in octal; see section The `#if' Directive.

`__REGISTER_PREFIX__`

This macro expands to a string describing the prefix applied to cpu registers in assembler code. It can be used to write assembler code that is usable in multiple environments. For example, in the `m68k-aout' environment it expands to the string `"", but in the `m68k-coff' environment it expands to the string `"%"'.

`__USER_LABEL_PREFIX__`

This macro expands to a string describing the prefix applied to user generated labels in assembler code. It can be used to write assembler code that is usable in multiple environments. For example, in the `m68k-aout' environment it expands to the string `"_"', but in the `m68k-coff' environment it expands to the string `""'.

Nonstandard Predefined Macros

The C preprocessor normally has several predefined macros that vary between machines because their purpose is to indicate what type of system and machine is in use. This manual, being for all systems and machines, cannot tell you exactly what their names are; instead, we offer a list of some typical ones. You can use `cpp -dM' to see the values of predefined macros; see section Invoking the C Preprocessor.

Some nonstandard predefined macros describe the operating system in use, with more or less specificity. For example,

`unix`

`unix' is normally predefined on all Unix systems.

`BSD`

`BSD' is predefined on recent versions of Berkeley Unix (perhaps only in version 4.3).

Other nonstandard predefined macros describe the kind of CPU, with more or less specificity. For example,

`vax`

`vax' is predefined on Vax computers.

`mc68000`

`mc68000' is predefined on most computers whose CPU is a Motorola 68000, 68010 or 68020.

`m68k`

`m68k' is also predefined on most computers whose CPU is a 68000, 68010 or 68020; however, some makers use `mc68000' and some use `m68k'. Some predefine both names. What happens in GNU C depends on the system you are using it on.

`M68020`

`M68020' has been observed to be predefined on some systems that use 68020 CPUs--in addition to `mc68000' and `m68k', which are less specific.

`_AM29K`

`_AM29000`

Both `_AM29K' and `_AM29000' are predefined for the AMD 29000 CPU family.

`ns32000`

`ns32000' is predefined on computers which use the National Semiconductor 32000 series CPU.

Yet other nonstandard predefined macros describe the manufacturer of the system. For example,

`sun`

`sun' is predefined on all models of Sun computers.

`pyr`

`pyr' is predefined on all models of Pyramid computers.

`sequent`

`sequent' is predefined on all models of Sequent computers.

These predefined symbols are not only nonstandard, they are contrary to the ANSI standard because their names do not start with underscores. Therefore, the option `-ansi' inhibits the definition of these symbols.

This tends to make `-ansi' useless, since many programs depend on the customary nonstandard predefined symbols. Even system header files check them and will generate incorrect declarations if they do not find the names that are expected. You might think that the header files supplied for the Uglix computer would not need to test what machine they are running on, because they can simply assume it is the Uglix; but often they do, and they do so using the customary names. As a result, very few C programs will compile with `-ansi'. We intend to avoid such problems on the GNU system.

What, then, should you do in an ANSI C program to test the type of machine it will run on?

GNU C offers a parallel series of symbols for this purpose, whose names are made from the customary ones by adding `__' at the beginning and end. Thus, the symbol `__vax__` would be available on a Vax, and so on.

The set of nonstandard predefined names in the GNU C preprocessor is controlled (when `cpp` is itself compiled) by the macro `CPP_PREDEFINES', which should be a string containing `-D' options,

separated by spaces. For example, on the Sun 3, we use the following definition:

```
#define CPP_PREDEFINES "-Dmc68000 -Dsun -Dunix -Dm68k"
```

This macro is usually specified in `tm.h'.

Stringification

Stringification means turning a code fragment into a string constant whose contents are the text for the code fragment. For example, stringifying `foo (z)' results in `"foo (z)"'.

In the C preprocessor, stringification is an option available when macro arguments are substituted into the macro definition. In the body of the definition, when an argument name appears, the character `#' before the name specifies stringification of the corresponding actual argument when it is substituted at that point in the definition. The same argument may be substituted in other places in the definition without stringification if the argument name appears in those places with no `#'.

Here is an example of a macro definition that uses stringification:

```
#define WARN_IF(EXP) \
do { if (EXP) \
        fprintf (stderr, "Warning: " #EXP "\n"); } \
while (0)
```

Here the actual argument for `EXP' is substituted once as given, into the `if' statement, and once as stringified, into the argument to `fprintf'. The `do' and `while (0)' are a kludge to make it possible to write `WARN_IF (arg);', which the resemblance of `WARN_IF' to a function would make C programmers want to do; see section Swallowing the Semicolon.

The stringification feature is limited to transforming one macro argument into one string constant: there is no way to combine the argument with other text and then stringify it all together. But the example above shows how an equivalent result can be obtained in ANSI Standard C using the feature that adjacent string constants are concatenated as one string constant. The preprocessor stringifies the actual value of `EXP' into a separate string constant, resulting in text like

```
do { if (x == 0) \
        fprintf (stderr, "Warning: " "x == 0" "\n"); } \
while (0)
```

but the C compiler then sees three consecutive string constants and concatenates them into one, producing effectively

```
do { if (x == 0) \
        fprintf (stderr, "Warning: x == 0\n"); } \
while (0)
```

Stringification in C involves more than putting doublequote characters around the fragment; it is

necessary to put backslashes in front of all doublequote characters, and all backslashes in string and character constants, in order to get a valid C string constant with the proper contents. Thus, stringifying `p = "foo\n";' results in `"p = \"foo\\n\";"'. However, backslashes that are not inside of string or character constants are not duplicated: `\n' by itself stringifies to `"\n"'.

Whitespace (including comments) in the text being stringified is handled according to precise rules. All leading and trailing whitespace is ignored. Any sequence of whitespace in the middle of the text is converted to a single space in the stringified result.

Concatenation

Concatenation means joining two strings into one. In the context of macro expansion, concatenation refers to joining two lexical units into one longer one. Specifically, an actual argument to the macro can be concatenated with another actual argument or with fixed text to produce a longer name. The longer name might be the name of a function, variable or type, or a C keyword; it might even be the name of another macro, in which case it will be expanded.

When you define a macro, you request concatenation with the special operator `##' in the macro body. When the macro is called, after actual arguments are substituted, all `##' operators are deleted, and so is any whitespace next to them (including whitespace that was part of an actual argument). The result is to concatenate the syntactic tokens on either side of the `##'.

Consider a C program that interprets named commands. There probably needs to be a table of commands, perhaps an array of structures declared as follows:

```
struct command
{
  char *name;
  void (*function) ();
};

struct command commands[] =
{
  { "quit", quit_command},
  { "help", help_command},
  ...
};
```

It would be cleaner not to have to give each command name twice, once in the string constant and once in the function name. A macro which takes the name of a command as an argument can make this unnecessary. The string constant can be created with stringification, and the function name by concatenating the argument with `_command'. Here is how it is done:

```
#define COMMAND(NAME)  { #NAME, NAME ## _command }

struct command commands[] =
```


```
{
  COMMAND (quit),
  COMMAND (help),
  ...
};
```

The usual case of concatenation is concatenating two names (or a name and a number) into a longer name. But this isn't the only valid case. It is also possible to concatenate two numbers (or a number and a name, such as `1.5' and `e3') into a number. Also, multi-character operators such as `+=' can be formed by concatenation. In some cases it is even possible to piece together a string constant. However, two pieces of text that don't together form a valid lexical unit cannot be concatenated. For example, concatenation with `x' on one side and `+' on the other is not meaningful because those two characters can't fit together in any lexical unit of C. The ANSI standard says that such attempts at concatenation are undefined, but in the GNU C preprocessor it is well defined: it puts the `x' and `+' side by side with no particular special results.

Keep in mind that the C preprocessor converts comments to whitespace before macros are even considered. Therefore, you cannot create a comment by concatenating `/' and `*': the `/*' sequence that starts a comment is not a lexical unit, but rather the beginning of a "long" space character. Also, you can freely use comments next to a `##' in a macro definition, or in actual arguments that will be concatenated, because the comments will be converted to spaces at first sight, and concatenation will later discard the spaces.

Undefining Macros

To undefine a macro means to cancel its definition. This is done with the `#undef' directive. `#undef' is followed by the macro name to be undefined.

Like definition, undefinition occurs at a specific point in the source file, and it applies starting from that point. The name ceases to be a macro name, and from that point on it is treated by the preprocessor as if it had never been a macro name.

For example,

```
#define FOO 4
x = FOO;
#undef FOO
x = FOO;
```

expands into

```
x = 4;

x = FOO;
```

In this example, `FOO' had better be a variable or function as well as (temporarily) a macro, in order for the result of the expansion to be valid C code.

The same form of `#undef' directive will cancel definitions with arguments or definitions that don't expect arguments. The `#undef' directive has no effect when used on a name not currently defined as a macro.

Redefining Macros

Redefining a macro means defining (with `#define') a name that is already defined as a macro.

A redefinition is trivial if the new definition is transparently identical to the old one. You probably wouldn't deliberately write a trivial redefinition, but they can happen automatically when a header file is included more than once (see section Header Files), so they are accepted silently and without effect.

Nontrivial redefinition is considered likely to be an error, so it provokes a warning message from the preprocessor. However, sometimes it is useful to change the definition of a macro in mid-compilation. You can inhibit the warning by undefining the macro with `#undef' before the second definition.

In order for a redefinition to be trivial, the new definition must exactly match the one already in effect, with two possible exceptions:

- Whitespace may be added or deleted at the beginning or the end.
- Whitespace may be changed in the middle (but not inside strings). However, it may not be eliminated entirely, and it may not be added where there was no whitespace at all.

Recall that a comment counts as whitespace.

Pitfalls and Subtleties of Macros

In this section we describe some special rules that apply to macros and macro expansion, and point out certain cases in which the rules have counterintuitive consequences that you must watch out for.

Improperly Nested Constructs

Recall that when a macro is called with arguments, the arguments are substituted into the macro body and the result is checked, together with the rest of the input file, for more macro calls.

It is possible to piece together a macro call coming partially from the macro body and partially from the actual arguments. For example,

```
#define double(x) (2*(x))
#define call_with_1(x) x(1)
```

would expand `call_with_1 (double)' into `(2*(1))'.

Macro definitions do not have to have balanced parentheses. By writing an unbalanced open parenthesis in a macro body, it is possible to create a macro call that begins inside the macro body but ends outside of it. For example,

```
#define strange(file) fprintf (file, "%s %d",
```


```
...
strange(stderr) p, 35)
```

This bizarre example expands to `fprintf (stderr, "%s %d", p, 35)'!

Unintended Grouping of Arithmetic

You may have noticed that in most of the macro definition examples shown above, each occurrence of a macro argument name had parentheses around it. In addition, another pair of parentheses usually surround the entire macro definition. Here is why it is best to write macros that way.

Suppose you define a macro as follows,

```
#define ceil_div(x, y) (x + y - 1) / y
```

whose purpose is to divide, rounding up. (One use for this operation is to compute how many `int' objects are needed to hold a certain number of `char' objects.) Then suppose it is used as follows:

```
a = ceil_div (b & c, sizeof (int));
```

This expands into

```
a = (b & c + sizeof (int) - 1) / sizeof (int);
```

which does not do what is intended. The operator-precedence rules of C make it equivalent to this:

```
a = (b & (c + sizeof (int) - 1)) / sizeof (int);
```

But what we want is this:

```
a = ((b & c) + sizeof (int) - 1)) / sizeof (int);
```

Defining the macro as

```
#define ceil_div(x, y) ((x) + (y) - 1) / (y)
```

provides the desired result.

However, unintended grouping can result in another way. Consider `sizeof ceil_div(1, 2)'. That has the appearance of a C expression that would compute the size of the type of `ceil_div (1, 2)', but in fact it means something very different. Here is what it expands to:

```
sizeof ((1) + (2) - 1) / (2)
```

This would take the size of an integer and divide it by two. The precedence rules have put the division outside the `sizeof' when it was intended to be inside.

Parentheses around the entire macro definition can prevent such problems. Here, then, is the

recommended way to define `ceil_div':

```
#define ceil_div(x, y) (((x) + (y) - 1) / (y))
```

Swallowing the Semicolon

Often it is desirable to define a macro that expands into a compound statement. Consider, for example, the following macro, that advances a pointer (the argument `p' says where to find it) across whitespace characters:

```
#define SKIP_SPACES (p, limit)  \
{ register char *lim = (limit); \
  while (p != lim) {            \
    if (*p++ != ' ') {          \
      p--; break; }}}
```

Here Backslash-Newline is used to split the macro definition, which must be a single line, so that it resembles the way such C code would be laid out if not part of a macro definition.

A call to this macro might be `SKIP_SPACES (p, lim)'. Strictly speaking, the call expands to a compound statement, which is a complete statement with no need for a semicolon to end it. But it looks like a function call. So it minimizes confusion if you can use it like a function call, writing a semicolon afterward, as in `SKIP_SPACES (p, lim);'

But this can cause trouble before `else' statements, because the semicolon is actually a null statement. Suppose you write

```
if (*p != 0)
  SKIP_SPACES (p, lim);
else ...
```

The presence of two statements--the compound statement and a null statement--in between the `if' condition and the `else' makes invalid C code.

The definition of the macro `SKIP_SPACES' can be altered to solve this problem, using a `do ... while' statement. Here is how:

```
#define SKIP_SPACES (p, limit)     \
do { register char *lim = (limit); \
     while (p != lim) {            \
       if (*p++ != ' ') {          \
         p--; break; }}}           \
while (0)
```

Now `SKIP_SPACES (p, lim);' expands into

```
do {...} while (0);
```

which is one statement.

Duplication of Side Effects

Many C programs define a macro `min', for "minimum", like this:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

When you use this macro with an argument containing a side effect, as shown here,

```
next = min (x + y, foo (z));
```

it expands as follows:

```
next = ((x + y) < (foo (z)) ? (x + y) : (foo (z)));
```

where `x + y' has been substituted for `X' and `foo (z)' for `Y'.

The function `foo' is used only once in the statement as it appears in the program, but the expression `foo (z)' has been substituted twice into the macro expansion. As a result, `foo' might be called two times when the statement is executed. If it has side effects or if it takes a long time to compute, the results might not be what you intended. We say that `min' is an unsafe macro.

The best solution to this problem is to define `min' in a way that computes the value of `foo (z)' only once. The C language offers no standard way to do this, but it can be done with GNU C extensions as follows:

```
#define min(X, Y)                       \
({ typeof (X) __x = (X), __y = (Y);   \
   (__x < __y) ? __x : __y; })
```

If you do not wish to use GNU C extensions, the only solution is to be careful when *using* the macro `min'. For example, you can calculate the value of `foo (z)', save it in a variable, and use that variable in `min':

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
...
{
  int tem = foo (z);
  next = min (x + y, tem);
}
```

(where we assume that `foo' returns type `int').

Self-Referential Macros

A self-referential macro is one whose name appears in its definition. A special feature of ANSI Standard

C is that the self-reference is not considered a macro call. It is passed into the preprocessor output unchanged.

Let's consider an example:

```
#define foo (4 + foo)
```

where `foo' is also a variable in your program.

Following the ordinary rules, each reference to `foo' will expand into `(4 + foo)'; then this will be rescanned and will expand into `(4 + (4 + foo))'; and so on until it causes a fatal error (memory full) in the preprocessor.

However, the special rule about self-reference cuts this process short after one step, at `(4 + foo)'. Therefore, this macro definition has the possibly useful effect of causing the program to add 4 to the value of `foo' wherever `foo' is referred to.

In most cases, it is a bad idea to take advantage of this feature. A person reading the program who sees that `foo' is a variable will not expect that it is a macro as well. The reader will come across the identifier `foo' in the program and think its value should be that of the variable `foo', whereas in fact the value is four greater.

The special rule for self-reference applies also to indirect self-reference. This is the case where a macro x expands to use a macro `y', and the expansion of `y' refers to the macro `x'. The resulting reference to `x' comes indirectly from the expansion of `x', so it is a self-reference and is not further expanded. Thus, after

```
#define x (4 + y)
#define y (2 * x)
```

`x' would expand into `(4 + (2 * x))'. Clear?

But suppose `y' is used elsewhere, not from the definition of `x'. Then the use of `x' in the expansion of `y' is not a self-reference because `x' is not "in progress". So it does expand. However, the expansion of `x' contains a reference to `y', and that is an indirect self-reference now because `y' is "in progress". The result is that `y' expands to `(2 * (4 + y))'.

It is not clear that this behavior would ever be useful, but it is specified by the ANSI C standard, so you may need to understand it.

Separate Expansion of Macro Arguments

We have explained that the expansion of a macro, including the substituted actual arguments, is scanned over again for macro calls to be expanded.

What really happens is more subtle: first each actual argument text is scanned separately for macro calls. Then the results of this are substituted into the macro body to produce the macro expansion, and the macro expansion is scanned again for macros to expand.

The result is that the actual arguments are scanned *twice* to expand macro calls in them.

Most of the time, this has no effect. If the actual argument contained any macro calls, they are expanded during the first scan. The result therefore contains no macro calls, so the second scan does not change it. If the actual argument were substituted as given, with no prescan, the single remaining scan would find the same macro calls and produce the same results.

You might expect the double scan to change the results when a self-referential macro is used in an actual argument of another macro (see section Self-Referential Macros): the self-referential macro would be expanded once in the first scan, and a second time in the second scan. But this is not what happens. The self-references that do not expand in the first scan are marked so that they will not expand in the second scan either.

The prescan is not done when an argument is stringified or concatenated. Thus,

```
#define str(s) #s
#define foo 4
str (foo)
```

expands to `"foo"'. Once more, prescan has been prevented from having any noticeable effect.

More precisely, stringification and concatenation use the argument as written, in un-prescanned form. The same actual argument would be used in prescanned form if it is substituted elsewhere without stringification or concatenation.

```
#define str(s) #s lose(s)
#define foo 4
str (foo)
```

expands to `"foo" lose(4)'.

You might now ask, "Why mention the prescan, if it makes no difference? And why not skip it and make the preprocessor faster?" The answer is that the prescan does make a difference in three special cases:

- Nested calls to a macro.
- Macros that call other macros that stringify or concatenate.
- Macros whose expansions contain unshielded commas.

We say that nested calls to a macro occur when a macro's actual argument contains a call to that very macro. For example, if `f' is a macro that expects one argument, `f (f (1))' is a nested pair of calls to `f'. The desired expansion is made by expanding `f (1)' and substituting that into the definition of `f'. The prescan causes the expected result to happen. Without the prescan, `f (1)' itself would be substituted as an actual argument, and the inner use of `f' would appear during the main scan as an indirect self-reference and would not be expanded. Here, the prescan cancels an undesirable side effect (in the medical, not computational, sense of the term) of the special rule for self-referential macros.

But prescan causes trouble in certain other cases of nested macro calls. Here is an example:


```
#define foo  a,b
#define bar(x) lose(x)
#define lose(x) (1 + (x))

bar(foo)
```

We would like `bar(foo)' to turn into `(1 + (foo))', which would then turn into `(1 + (a,b))'. But instead, `bar(foo)' expands into `lose(a,b)', and you get an error because `lose` requires a single argument. In this case, the problem is easily solved by the same parentheses that ought to be used to prevent misnesting of arithmetic operations:

```
#define foo (a,b)
#define bar(x) lose((x))
```

The problem is more serious when the operands of the macro are not expressions; for example, when they are statements. Then parentheses are unacceptable because they would make for invalid C code:

```
#define foo { int a, b; ... }
```

In GNU C you can shield the commas using the `({...})' construct which turns a compound statement into an expression:

```
#define foo ({ int a, b; ... })
```

Or you can rewrite the macro definition to avoid such commas:

```
#define foo { int a; int b; ... }
```

There is also one case where prescan is useful. It is possible to use prescan to expand an argument and then stringify it--if you use two levels of macros. Let's add a new macro `xstr' to the example shown above:

```
#define xstr(s) str(s)
#define str(s) #s
#define foo 4
xstr (foo)
```

This expands into `"4"', not `"foo"'. The reason for the difference is that the argument of `xstr' is expanded at prescan (because `xstr' does not specify stringification or concatenation of the argument). The result of prescan then forms the actual argument for `str'. `str' uses its argument without prescan because it performs stringification; but it cannot prevent or undo the prescanning already done by `xstr'.

Cascaded Use of Macros

A cascade of macros is when one macro's body contains a reference to another macro. This is very common practice. For example,


```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
```

This is not at all the same as defining `TABLESIZE' to be `1020'. The `#define' for `TABLESIZE' uses exactly the body you specify--in this case, `BUFSIZE'---and does not check to see whether it too is the name of a macro.

It's only when you *use* `TABLESIZE' that the result of its expansion is checked for more macro names.

This makes a difference if you change the definition of `BUFSIZE' at some point in the source file. `TABLESIZE', defined as shown, will always expand using the definition of `BUFSIZE' that is currently in effect:

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
#undef BUFSIZE
#define BUFSIZE 37
```

Now `TABLESIZE' expands (in two stages) to `37'. (The `#undef' is to prevent any warning about the nontrivial redefinition of `BUFSIZE'.)

Newlines in Macro Arguments

Traditional macro processing carries forward all newlines in macro arguments into the expansion of the macro. This means that, if some of the arguments are substituted more than once, or not at all, or out of order, newlines can be duplicated, lost, or moved around within the expansion. If the expansion consists of multiple statements, then the effect is to distort the line numbers of some of these statements. The result can be incorrect line numbers, in error messages or displayed in a debugger.

The GNU C preprocessor operating in ANSI C mode adjusts appropriately for multiple use of an argument--the first use expands all the newlines, and subsequent uses of the same argument produce no newlines. But even in this mode, it can produce incorrect line numbering if arguments are used out of order, or not used at all.

Here is an example illustrating this problem:

```
#define ignore_second_arg(a,b,c) a; c

ignore_second_arg (foo (),
                   ignored (),
                   syntax error);
```

The syntax error triggered by the tokens `syntax error' results in an error message citing line four, even though the statement text comes from line five.

Conditionals

In a macro processor, a conditional is a directive that allows a part of the program to be ignored during compilation, on some conditions. In the C preprocessor, a conditional can test either an arithmetic expression or whether a name is defined as a macro.

A conditional in the C preprocessor resembles in some ways an `if' statement in C, but it is important to understand the difference between them. The condition in an `if' statement is tested during the execution of your program. Its purpose is to allow your program to behave differently from run to run, depending on the data it is operating on. The condition in a preprocessing conditional directive is tested when your program is compiled. Its purpose is to allow different code to be included in the program depending on the situation at the time of compilation.

Why Conditionals are Used

Generally there are three kinds of reason to use a conditional.

- A program may need to use different code depending on the machine or operating system it is to run on. In some cases the code for one operating system may be erroneous on another operating system; for example, it might refer to library routines that do not exist on the other system. When this happens, it is not enough to avoid executing the invalid code: merely having it in the program makes it impossible to link the program and run it. With a preprocessing conditional, the offending code can be effectively excised from the program when it is not valid.
- You may want to be able to compile the same source file into two different programs. Sometimes the difference between the programs is that one makes frequent time-consuming consistency checks on its intermediate data, or prints the values of those data for debugging, while the other does not.
- A conditional whose condition is always false is a good way to exclude code from the program but keep it as a sort of comment for future reference.

Most simple programs that are intended to run on only one machine will not need to use preprocessing conditionals.

Syntax of Conditionals

A conditional in the C preprocessor begins with a conditional directive: `#if', `#ifdef' or `#ifndef'. See section Conditionals and Macros, for information on `#ifdef' and `#ifndef'; only `#if' is explained here.

The `#if' Directive

The `#if' directive in its simplest form consists of

```
#if expression
controlled text
#endif /* expression */
```

The comment following the `#endif' is not required, but it is a good practice because it helps people match the `#endif' to the corresponding `#if'. Such comments should always be used, except in short conditionals that are not nested. In fact, you can put anything at all after the `#endif' and it will be ignored by the GNU C preprocessor, but only comments are acceptable in ANSI Standard C.

expression is a C expression of integer type, subject to stringent restrictions. It may contain

- Integer constants, which are all regarded as `long` or `unsigned long`.
- Character constants, which are interpreted according to the character set and conventions of the machine and operating system on which the preprocessor is running. The GNU C preprocessor uses the C data type `char' for these character constants; therefore, whether some character codes are negative is determined by the C compiler used to compile the preprocessor. If it treats `char' as signed, then character codes large enough to set the sign bit will be considered negative; otherwise, no character code is considered negative.
- Arithmetic operators for addition, subtraction, multiplication, division, bitwise operations, shifts, comparisons, and logical operations (`&&' and `||').
- Identifiers that are not macros, which are all treated as zero(!).
- Macro calls. All macro calls in the expression are expanded before actual computation of the expression's value begins.

Note that `sizeof' operators and `enum`-type values are not allowed. `enum`-type values, like all other identifiers that are not taken as macro calls and expanded, are treated as zero.

The controlled text inside of a conditional can include preprocessing directives. Then the directives inside the conditional are obeyed only if that branch of the conditional succeeds. The text can also contain other conditional groups. However, the `#if' and `#endif' directives must balance.

The `#else' Directive

The `#else' directive can be added to a conditional to provide alternative text to be used if the condition is false. This is what it looks like:

```
#if expression
text-if-true
#else /* Not expression */
text-if-false
#endif /* Not expression */
```

If expression is nonzero, and thus the text-if-true is active, then `#else' acts like a failing conditional and the text-if-false is ignored. Contrariwise, if the `#if' conditional fails, the text-if-false is considered included.

The `#elif' Directive

One common case of nested conditionals is used to check for more than two possible alternatives. For example, you might have


```
#if X == 1
...
#else /* X != 1 */
#if X == 2
...
#else /* X != 2 */
...
#endif /* X != 2 */
#endif /* X != 1 */
```

Another conditional directive, `#elif', allows this to be abbreviated as follows:

```
#if X == 1
...
#elif X == 2
...
#else /* X != 2 and X != 1*/
...
#endif /* X != 2 and X != 1*/
```

`#elif' stands for "else if". Like `#else', it goes in the middle of a `#if'-`#endif' pair and subdivides it; it does not require a matching `#endif' of its own. Like `#if', the `#elif' directive includes an expression to be tested.

The text following the `#elif' is processed only if the original `#if'-condition failed and the `#elif' condition succeeds. More than one `#elif' can go in the same `#if'-`#endif' group. Then the text after each `#elif' is processed only if the `#elif' condition succeeds after the original `#if' and any previous `#elif' directives within it have failed. `#else' is equivalent to `#elif 1', and `#else' is allowed after any number of `#elif' directives, but `#elif' may not follow `#else'.

Keeping Deleted Code for Future Reference

If you replace or delete a part of the program but want to keep the old code around as a comment for future reference, the easy way to do this is to put `#if 0' before it and `#endif' after it. This is better than using comment delimiters `/*' and `*/' since those won't work if the code already contains comments (C comments do not nest).

This works even if the code being turned off contains conditionals, but they must be entire conditionals (balanced `#if' and `#endif').

Conversely, do not use `#if 0' for comments which are not C code. Use the comment delimiters `/*' and `*/' instead. The interior of `#if 0' must consist of complete tokens; in particular, singlequote characters must balance. But comments often contain unbalanced singlequote characters (known in English as apostrophes). These confuse `#if 0'. They do not confuse `/*'.

Conditionals and Macros

Conditionals are useful in connection with macros or assertions, because those are the only ways that an expression's value can vary from one compilation to another. A `#if' directive whose expression uses no macros or assertions is equivalent to `#if 1' or `#if 0'; you might as well determine which one, by computing the value of the expression yourself, and then simplify the program.

For example, here is a conditional that tests the expression `BUFSIZE == 1020', where `BUFSIZE' must be a macro.

```
#if BUFSIZE == 1020
  printf ("Large buffers!\n");
#endif /* BUFSIZE is large */
```

(Programmers often wish they could test the size of a variable or data type in `#if', but this does not work. The preprocessor does not understand `sizeof`, or typedef names, or even the type keywords such as `int`.)

The special operator `defined' is used in `#if' expressions to test whether a certain name is defined as a macro. Either `defined name' or `defined (name)' is an expression whose value is 1 if name is defined as macro at the current point in the program, and 0 otherwise. For the `defined' operator it makes no difference what the definition of the macro is; all that matters is whether there is a definition. Thus, for example,

```
#if defined (vax) || defined (ns16000)
```

would succeed if either of the names `vax' and `ns16000' is defined as a macro. You can test the same condition using assertions (see section Assertions), like this:

```
#if #cpu (vax) || #cpu (ns16000)
```

If a macro is defined and later undefined with `#undef', subsequent use of the `defined' operator returns 0, because the name is no longer defined. If the macro is defined again with another `#define', `defined' will recommence returning 1.

Conditionals that test whether just one name is defined are very common, so there are two special short conditional directives for this case.

`#ifdef name`

is equivalent to `#if defined (name)'.

`#ifndef name`

is equivalent to `#if ! defined (name)'.

Macro definitions can vary between compilations for several reasons.

- Some macros are predefined on each kind of machine. For example, on a Vax, the name `vax' is a predefined macro. On other machines, it would not be defined.

- Many more macros are defined by system header files. Different systems and machines define different macros, or give them different values. It is useful to test these macros with conditionals to avoid using a system feature on a machine where it is not implemented.
- Macros are a common way of allowing users to customize a program for different machines or applications. For example, the macro `BUFSIZE' might be defined in a configuration file for your program that is included as a header file in each source file. You would use `BUFSIZE' in a preprocessing conditional in order to generate different code depending on the chosen configuration.
- Macros can be defined or undefined with `-D' and `-U' command options when you compile the program. You can arrange to compile the same source file into two different programs by choosing a macro name to specify which program you want, writing conditionals to test whether or how this macro is defined, and then controlling the state of the macro with compiler command options. See section Invoking the C Preprocessor.

Assertions

Assertions are a more systematic alternative to macros in writing conditionals to test what sort of computer or system the compiled program will run on. Assertions are usually predefined, but you can define them with preprocessing directives or command-line options.

The macros traditionally used to describe the type of target are not classified in any way according to which question they answer; they may indicate a hardware architecture, a particular hardware model, an operating system, a particular version of an operating system, or specific configuration options. These are jumbled together in a single namespace. In contrast, each assertion consists of a named question and an answer. The question is usually called the predicate. An assertion looks like this:

```
#predicate (answer)
```

You must use a properly formed identifier for predicate. The value of answer can be any sequence of words; all characters are significant except for leading and trailing whitespace, and differences in internal whitespace sequences are ignored. Thus, `x + y' is different from `x+y' but equivalent to `x + y'. `)' is not allowed in an answer.

Here is a conditional to test whether the answer answer is asserted for the predicate predicate:

```
#if #predicate (answer)
```

There may be more than one answer asserted for a given predicate. If you omit the answer, you can test whether *any* answer is asserted for predicate:

```
#if #predicate
```

Most of the time, the assertions you test will be predefined assertions. GNU C provides three predefined predicates: `system`, `cpu`, and `machine`. `system` is for assertions about the type of software, `cpu` describes the type of computer architecture, and `machine` gives more information about the computer. For example, on a GNU system, the following assertions would be true:


```
#system (gnu)
#system (mach)
#system (mach 3)
#system (mach 3.subversion)
#system (hurd)
#system (hurd version)
```

and perhaps others. The alternatives with more or less version information let you ask more or less detailed questions about the type of system software.

On a Unix system, you would find `#system (unix)` and perhaps one of: `#system (aix)`, `#system (bsd)`, `#system (hpux)`, `#system (lynx)`, `#system (mach)`, `#system (posix)`, `#system (svr3)`, `#system (svr4)`, or `#system (xpg4)` with possible version numbers following.

Other values for `system` are `#system (mvs)` and `#system (vms)`.

Portability note: Many Unix C compilers provide only one answer for the `system` assertion: `#system (unix)`, if they support assertions at all. This is less than useful.

An assertion with a multi-word answer is completely different from several assertions with individual single-word answers. For example, the presence of `system (mach 3.0)` does not mean that `system (3.0)` is true. It also does not directly imply `system (mach)`, but in GNU C, that last will normally be asserted as well.

The current list of possible assertion values for `cpu` is: `#cpu (a29k)`, `#cpu (alpha)`, `#cpu (arm)`, `#cpu (clipper)`, `#cpu (convex)`, `#cpu (elxsi)`, `#cpu (tron)`, `#cpu (h8300)`, `#cpu (i370)`, `#cpu (i386)`, `#cpu (i860)`, `#cpu (i960)`, `#cpu (m68k)`, `#cpu (m88k)`, `#cpu (mips)`, `#cpu (ns32k)`, `#cpu (hppa)`, `#cpu (pyr)`, `#cpu (ibm032)`, `#cpu (rs6000)`, `#cpu (sh)`, `#cpu (sparc)`, `#cpu (spur)`, `#cpu (tahoe)`, `#cpu (vax)`, `#cpu (we32000)`.

You can create assertions within a C program using `#assert', like this:

```
#assert predicate (answer)
```

(Note the absence of a `#' before predicate.)

Each time you do this, you assert a new true answer for predicate. Asserting one answer does not invalidate previously asserted answers; they all remain true. The only way to remove an assertion is with `#unassert'. `#unassert' has the same syntax as `#assert'. You can also remove all assertions about predicate like this:

```
#unassert predicate
```

You can also add or cancel assertions using command options when you run `gcc` or `cpp`. See section Invoking the C Preprocessor.

The `#error' and `#warning' Directives

The directive `#error' causes the preprocessor to report a fatal error. The rest of the line that follows `#error' is used as the error message.

You would use `#error' inside of a conditional that detects a combination of parameters which you know the program does not properly support. For example, if you know that the program will not run properly on a Vax, you might write

```
#ifdef __vax__
#error Won't work on Vaxen.  See comments at get_last_object.
#endif
```

See section Nonstandard Predefined Macros, for why this works.

If you have several configuration parameters that must be set up by the installation in a consistent way, you can use conditionals to detect an inconsistency and report it with `#error'. For example,

```
#if HASH_TABLE_SIZE % 2 == 0 || HASH_TABLE_SIZE % 3 == 0 \
    || HASH_TABLE_SIZE % 5 == 0
#error HASH_TABLE_SIZE should not be divisible by a small prime
#endif
```

The directive `#warning' is like the directive `#error', but causes the preprocessor to issue a warning and continue preprocessing. The rest of the line that follows `#warning' is used as the warning message.

You might use `#warning' in obsolete header files, with a message directing the user to the header file which should be used instead.

Combining Source Files

One of the jobs of the C preprocessor is to inform the C compiler of where each line of C code came from: which source file and which line number.

C code can come from multiple source files if you use `#include'; both `#include' and the use of conditionals and macros can cause the line number of a line in the preprocessor output to be different from the line's number in the original source file. You will appreciate the value of making both the C compiler (in error messages) and symbolic debuggers such as GDB use the line numbers in your source file.

The C preprocessor builds on this feature by offering a directive by which you can control the feature explicitly. This is useful when a file for input to the C preprocessor is the output from another program such as the `bison` parser generator, which operates on another file that is the true source file. Parts of the output from `bison` are generated from scratch, other parts come from a standard parser file. The rest are copied nearly verbatim from the source file, but their line numbers in the `bison` output are not the same as their original line numbers. Naturally you would like compiler error messages and symbolic

debuggers to know the original source file and line number of each line in the `bison` input.

`bison` arranges this by writing `#line' directives into the output file. `#line' is a directive that specifies the original line number and source file name for subsequent input in the current preprocessor input file. `#line' has three variants:

`#line linenum`

Here linenum is a decimal integer constant. This specifies that the line number of the following line of input, in its original source file, was linenum.

`#line linenum filename`

Here linenum is a decimal integer constant and filename is a string constant. This specifies that the following line of input came originally from source file filename and its line number there was linenum. Keep in mind that filename is not just a file name; it is surrounded by doublequote characters so that it looks like a string constant.

`#line anything else`

anything else is checked for macro calls, which are expanded. The result should be a decimal integer constant followed optionally by a string constant, as described above.

`#line' directives alter the results of the `__FILE__' and `__LINE__' predefined macros from that point on. See section Standard Predefined Macros.

The output of the preprocessor (which is the input for the rest of the compiler) contains directives that look much like `#line' directives. They start with just `#' instead of `#line', but this is followed by a line number and file name as in `#line'. See section C Preprocessor Output.

Miscellaneous Preprocessing Directives

This section describes three additional preprocessing directives. They are not very useful, but are mentioned for completeness.

The null directive consists of a `#' followed by a Newline, with only whitespace (including comments) in between. A null directive is understood as a preprocessing directive but has no effect on the preprocessor output. The primary significance of the existence of the null directive is that an input line consisting of just a `#' will produce no output, rather than a line of output containing just a `#'. Supposedly some old C programs contain such lines.

The ANSI standard specifies that the `#pragma' directive has an arbitrary, implementation-defined effect. In the GNU C preprocessor, `#pragma' directives are not used, except for `#pragma once' (see section Once-Only Include Files). However, they are left in the preprocessor output, so they are available to the compilation pass.

The `#ident' directive is supported for compatibility with certain other systems. It is followed by a line of text. On some systems, the text is copied into a special place in the object file; on most systems, the text is ignored and this directive has no effect. Typically `#ident' is only used in header files supplied with those systems where it is meaningful.

C Preprocessor Output

The output from the C preprocessor looks much like the input, except that all preprocessing directive lines have been replaced with blank lines and all comments with spaces. Whitespace within a line is not altered; however, a space is inserted after the expansions of most macro calls.

Source file name and line number information is conveyed by lines of the form

```
# linenum filename flags
```

which are inserted as needed into the middle of the input (but never within a string or character constant). Such a line means that the following line originated in file filename at line linenum.

After the file name comes zero or more flags, which are `1', `2' or `3'. If there are multiple flags, spaces separate them. Here is what the flags mean:

`1'

This indicates the start of a new file.

`2'

This indicates returning to a file (after having included another file).

`3'

This indicates that the following text comes from a system header file, so certain warnings should be suppressed.

Invoking the C Preprocessor

Most often when you use the C preprocessor you will not have to invoke it explicitly: the C compiler will do so automatically. However, the preprocessor is sometimes useful on its own.

The C preprocessor expects two file names as arguments, infile and outfile. The preprocessor reads infile together with any other files it specifies with `#include'. All the output generated by the combined input files is written in outfile.

Either infile or outfile may be `-', which as infile means to read from standard input and as outfile means to write to standard output. Also, if outfile or both file names are omitted, the standard output and standard input are used for the omitted file names.

Here is a table of command options accepted by the C preprocessor. These options can also be given when compiling a C program; they are passed along automatically to the preprocessor when it is invoked by the compiler.

`-P'

Inhibit generation of `#'-lines with line-number information in the output from the preprocessor (see section C Preprocessor Output). This might be useful when running the preprocessor on something that is not C code and will be sent to a program which might be confused by the `#'-lines.

`-C'

Do not discard comments: pass them through to the output file. Comments appearing in arguments of a macro call will be copied to the output before the expansion of the macro call.

`-traditional'

Try to imitate the behavior of old-fashioned C, as opposed to ANSI C.

`'

Traditional macro expansion pays no attention to singlequote or doublequote characters; macro argument symbols are replaced by the argument values even when they appear within apparent string or character constants.

`'

Traditionally, it is permissible for a macro expansion to end in the middle of a string or character constant. The constant continues into the text surrounding the macro call.

`'

However, traditionally the end of the line terminates a string or character constant, with no error.

`'

In traditional C, a comment is equivalent to no text at all. (In ANSI C, a comment counts as whitespace.)

`'

Traditional C does not have the concept of a "preprocessing number". It considers `1.0e+4' to be three tokens: `1.0e', `+', and `4'.

`'

A macro is not suppressed within its own definition, in traditional C. Thus, any macro that is used recursively inevitably causes an error.

`'

The character `#' has no special meaning within a macro definition in traditional C.

`'

In traditional C, the text at the end of a macro expansion can run together with the text after the macro call, to produce a single token. (This is impossible in ANSI C.)

`'

Traditionally, `\' inside a macro argument suppresses the syntactic significance of the following character.

`-trigraphs'

Process ANSI standard trigraph sequences. These are three-character sequences, all starting with `??', that are defined by ANSI C to stand for single characters. For example, `??/' stands for `\', so `'??/n" is a character constant for a newline. Strictly speaking, the GNU C preprocessor does not support all programs in ANSI Standard C unless `-trigraphs' is used, but if you ever notice the difference it will be with relief.

You don't want to know any more about trigraphs.

`-pedantic'

Issue warnings required by the ANSI C standard in certain cases such as when text other than a comment follows `#else' or `#endif'.

`-pedantic-errors'

Like `-pedantic', except that errors are produced rather than warnings.

`-Wtrigraphs'

Warn if any trigraphs are encountered (assuming they are enabled).

`-Wcomment'

Warn whenever a comment-start sequence `/*' appears in a comment.

`-Wall'

Requests both `-Wtrigraphs' and `-Wcomment' (but not `-Wtraditional').

`-Wtraditional'

Warn about certain constructs that behave differently in traditional and ANSI C.

`-I directory'

Add the directory directory to the head of the list of directories to be searched for header files (see section The `#include' Directive). This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one `-I' option, the directories are scanned in left-to-right order; the standard system directories come after.

`-I-'

Any directories specified with `-I' options before the `-I-' option are searched only for the case of `#include "file"'; they are not searched for `#include <file>'.

If additional directories are specified with `-I' options after the `-I-', these directories are searched for all `#include' directives.

In addition, the `-I-' option inhibits the use of the current directory as the first search directory for `#include "file"'. Therefore, the current directory is searched only if it is requested explicitly with `-I.'. Specifying both `-I-' and `-I.' allows you to control precisely which directories are searched before the current one and which are searched after.

`-nostdinc'

Do not search the standard system directories for header files. Only the directories you have specified with `-I' options (and the current directory, if appropriate) are searched.

`-nostdinc++'

Do not search for header files in the C++-specific standard directories, but do still search the other standard directories. (This option is used when building libg++.)

`-D name'

Predefine name as a macro, with definition `1'.

`-D name=definition'

Predefine name as a macro, with definition definition. There are no restrictions on the contents of

definition, but if you are invoking the preprocessor from a shell or shell-like program you may need to use the shell's quoting syntax to protect characters such as spaces that have a meaning in the shell syntax. If you use more than one `-D' for the same name, the rightmost definition takes effect.

`-U name'

Do not predefine name. If both `-U' and `-D' are specified for one name, the `-U' beats the `-D' and the name is not predefined.

`-undef'

Do not predefine any nonstandard macros.

`-A predicate(answer)'

Make an assertion with the predicate predicate and answer answer. See section Assertions.

You can use `-A-' to disable all predefined assertions; it also undefines all predefined macros that identify the type of target system.

`-dM'

Instead of outputting the result of preprocessing, output a list of `#define' directives for all the macros defined during the execution of the preprocessor, including predefined macros. This gives you a way of finding out what is predefined in your version of the preprocessor; assuming you have no file `foo.h', the command

```
touch foo.h; cpp -dM foo.h
```

will show the values of any predefined macros.

`-dD'

Like `-dM' except in two respects: it does *not* include the predefined macros, and it outputs *both* the `#define' directives and the result of preprocessing. Both kinds of output go to the standard output file.

`-M [-MG]'

Instead of outputting the result of preprocessing, output a rule suitable for `make` describing the dependencies of the main source file. The preprocessor outputs one `make` rule containing the object file name for that source file, a colon, and the names of all the included files. If there are many included files then the rule is split into several lines using `\'-newline.

`-MG' says to treat missing header files as generated files and assume they live in the same directory as the source file. It must be specified in addition to `-M'.

This feature is used in automatic updating of makefiles.

`-MM [-MG]'

Like `-M' but mention only the files included with `#include "file"'. System header files included with `#include <file>' are omitted.

`-MD file'

Like `-M' but the dependency information is written to file. This is in addition to compiling the file as specified---`-MD' does not inhibit ordinary compilation the way `-M' does.

When invoking gcc, do not specify the file argument. Gcc will create file names made by replacing ".c" with ".d" at the end of the input file names.

In Mach, you can use the utility `md` to merge multiple dependency files into a single dependency file suitable for using with the `make' command.

`-MMD file'

Like `-MD' except mention only user header files, not system header files.

`-H'

Print the name of each header file used, in addition to other normal activities.

`-imacros file'

Process file as input, discarding the resulting output, before processing the regular input file. Because the output generated from file is discarded, the only effect of `-imacros file' is to make the macros defined in file available for use in the main input.

`-include file'

Process file as input, and include all the resulting output, before processing the regular input file.

`-idirafter dir'

Add the directory dir to the second include path. The directories on the second include path are searched when a header file is not found in any of the directories in the main include path (the one that `-I' adds to).

`-iprefix prefix'

Specify prefix as the prefix for subsequent `-iwithprefix' options.

`-iwithprefix dir'

Add a directory to the second include path. The directory's name is made by concatenating prefix and dir, where prefix was specified previously with `-iprefix'.

`-isystem dir'

Add a directory to the beginning of the second include path, marking it as a system directory, so that it gets the same special treatment as is applied to the standard system directories.

`-lang-c'

`-lang-c++'

`-lang-objc'

`-lang-objc++'

Specify the source language. `-lang-c++' makes the preprocessor handle C++ comment syntax (comments may begin with `//', in which case they end at end of line), and includes extra default include directories for C++; and `-lang-objc' enables the Objective C `#import' directive. `-lang-c' explicitly turns off both of these extensions, and `-lang-objc++' enables both.

These options are generated by the compiler driver `gcc`, but not passed from the `gcc' command line.

`-lint'

Look for commands to the program checker `lint` embedded in comments, and emit them

preceded by `#pragma lint'. For example, the comment `/* NOTREACHED */' becomes `#pragma lint NOTREACHED'.

This option is available only when you call `cpp` directly; `gcc` will not pass it from its command line.

`-$'

Forbid the use of `$' in identifiers. This is required for ANSI conformance. `gcc` automatically supplies this option to the preprocessor if you specify `-ansi', but `gcc` doesn't recognize the `-$' option itself--to use it without the other effects of `-ansi', you must call the preprocessor directly.

Concept Index

#

a

b

c

TOC

d

e

f

h

i

l

m

n

o

p

r

s

t

u

Go to the next section.

Go to the previous section.

Index of Directives, Macros and Options

#

-

Go to the previous section.

CVS--Concurrent Versions System

Go to the next section.

Version Management

with

CVS

for CVS 1.8.1

Per Cederqvist et al

About this manual

Up to this point, one of the weakest parts of CVS has been the documentation. CVS is a complex program. Previous versions of the manual were written in the manual page format, which is not really well suited for such a complex program.

When writing this manual, I had several goals in mind:

- No knowledge of RCS should be necessary.
- No previous knowledge of revision control software should be necessary. All terms, such as revision numbers, revision trees and merging are explained as they are introduced.
- The manual should concentrate on the things CVS users want to do, instead of what the CVS commands can do. The first part of this manual leads you through things you might want to do while doing development, and introduces the relevant CVS commands as they are needed.
- Information should be easy to find. In the reference manual in the appendices almost all information about every CVS command is gathered together. There is also an extensive index, and a lot of cross references.

This manual was contributed by Signum Support AB in Sweden. Signum is yet another in the growing list of companies that support free software. You are free to copy both this manual and the CVS program. See section GNU GENERAL PUBLIC LICENSE, for the details. Signum Support offers support

contracts and binary distribution for many programs, such as CVS, GNU Emacs, the GNU C compi and others. Write to us for more information.

```
Signum Support AB
Box 2044
S-580 02  Linkoping
Sweden

Email: info@signum.se
Phone: +46 (0)13 - 21 46 00
Fax:   +46 (0)13 - 21 47 00
```

Another company selling support for CVS is Cyclic Software, web: `http://www.cyclic.com/`, email: `info@cyclic.com`.

Checklist for the impatient reader

CVS is a complex system. You will need to read the manual to be able to use all of its capabilities. There are dangers that can easily be avoided if you know about them, and this manual tries to warn you about them. This checklist is intended to help you avoid the dangers without reading the entire manual. If you intend to read the entire manual you can skip this table.

Binary files

> CVS can handle binary files, but you must have RCS release 5.5 or later and a release of GNU diff that supports the `-a' flag (release 1.15 and later are OK). You must also configure both RCS and CVS to handle binary files when you install them.
>
> Keword substitution can be a source of trouble with binary files. See section Keyword substitution, for solutions.

The `admin` command

> Uncareful use of the `admin` command can cause CVS to cease working. See section admin--Administration front end for rcs, before trying to use it.

Credits

Roland Pesch, Cygnus Support <`pesch@cygnus.com`> wrote the manual pages which were distributed with CVS 1.3. Appendix A and B contain much text that was extracted from them. He also read an early draft of this manual and contributed many ideas and corrections.

The mailing-list `info-cvs` is sometimes informative. I have included information from postings made by the following persons: David G. Grubbs <`dgg@think.com`>.

Some text has been extracted from the man pages for RCS.

The CVS FAQ (see section What is CVS?) by David G. Grubbs has been used as a check-list to make

sure that this manual is as complete as possible. (This manual does however not include all of the material in the FAQ). The FAQ contains a lot of useful information.

In addition, the following persons have helped by telling me about mistakes I've made: Roxanne Brunskill <rbrunski@datap.ca>, Kathy Dyer <dyer@phoenix.ocf.llnl.gov>, Karl Pingle <pingle@acuson.com>, Thomas A Peterson <tap@src.honeywell.com>, Inge Wallin <ingwa@signum.se>, Dirk Koschuetzki <koschuet@fmi.uni-passau.de> and Michael Brown <brown@wi.extrel.com>.

BUGS

This manual is known to have room for improvement. Here is a list of known deficiencies:

- In the examples, the output from CVS is sometimes displayed, sometimes not.
- The input that you are supposed to type in the examples should have a different font than the output from the computer.
- This manual should be clearer about what file permissions you should set up in the repository, and about setuid/setgid.
- Some of the chapters are not yet complete. They are noted by comments in the `cvs.texinfo' file.
- This list is not complete. If you notice any error, omission, or something that is unclear, please send mail to bug-cvs@prep.ai.mit.edu.

I hope that you will find this manual useful, despite the above-mentioned shortcomings.

```
Linkoping, October 1993
Per Cederqvist
```

Go to the next section.

Go to the previous, next section.

What is CVS?

CVS is a version control system. Using it, you can record the history of your source files.

For example, bugs sometimes creep in when software is modified, and you might not detect the bug until a long time after you make the modification. With CVS, you can easily retrieve old versions to see exactly which change caused the bug. This can sometimes be a big help.

You could of course save every version of every file you have ever created. This would however waste an enormous amount of disk space. CVS stores all the versions of a file in a single file in a clever way that only stores the differences between versions.

CVS also helps you if you are part of a group of people working on the same project. It is all too easy to overwrite each others' changes unless you are extremely careful. Some editors, like GNU Emacs, try to make sure that the same file is never modified by two people at the same time. Unfortunately, if someone is using another editor, that safeguard will not work. CVS solves this problem by insulating the different developers from each other. Every developer works in his own directory, and CVS merges the work when each developer is done.

CVS started out as a bunch of shell scripts written by Dick Grune, posted to `comp.sources.unix` in the volume 6 release of December, 1986. While no actual code from these shell scripts is present in the current version of CVS much of the CVS conflict resolution algorithms come from them.

In April, 1989, Brian Berliner designed and coded CVS. Jeff Polk later helped Brian with the design of the CVS module and vendor branch support.

You can get CVS via anonymous ftp from a number of sites, for instance `prep.ai.mit.edu` in `` `pub/gnu' ``.

There is a mailing list for CVS where bug reports can be sent, questions can be asked, an FAQ is posted, and discussion about future enhancements to CVS take place. To submit a message to the list, write to <`info-cvs@prep.ai.mit.edu`>. To subscribe or unsubscribe, write to <`info-cvs-request@prep.ai.mit.edu`>. Please be specific about your email address.

CVS is not...

CVS can do a lot of things for you, but it does not try to be everything for everyone.

CVS is not a build system.

> Though the structure of your repository and modules file interact with your build system (e.g. `` `Makefile' ``s), they are essentially independent.
>
> CVS does not dictate how you build anything. It merely stores files for retrieval in a tree structure you devise.

CVS does not dictate how to use disk space in the checked out working directories. If you wr:t your `Makefile's or scripts in every directory so they have to know the relative positions o everything else, you wind up requiring the entire repository to be checked out. That's simply bad planning.

If you modularize your work, and construct a build system that will share files (via links, mounts, VPATH in `Makefile's, etc.), you can arrange your disk usage however you like.

But you have to remember that *any* such system is a lot of work to construct and maintain. CVS does not address the issues involved. You must use your brain and a collection of other tools to provide a build scheme to match your plans.

Of course, you should place the tools created to support such a build system (scripts, `Makefile's, etc) under CVS.

CVS is not a substitute for management.

Your managers and project leaders are expected to talk to you frequently enough to make certain you are aware of schedules, merge points, branch names and release dates. If they don't, CVS can't help.

CVS is an instrument for making sources dance to your tune. But you are the piper and the composer. No instrument plays itself or writes its own music.

CVS is not a substitute for developer communication.

When faced with conflicts within a single file, most developers manage to resolve them without too much effort. But a more general definition of "conflict" includes problems too difficult to solve without communication between developers.

CVS cannot determine when simultaneous changes within a single file, or across a whole collection of files, will logically conflict with one another. Its concept of a conflict is purely textual, arising when two changes to the same base file are near enough to spook the merge (i.e. diff3) command.

CVS does not claim to help at all in figuring out non-textual or distributed conflicts in program logic.

For example: Say you change the arguments to function X defined in file `A'. At the same time, someone edits file `B', adding new calls to function X using the old arguments. You are outside the realm of CVS's competence.

Acquire the habit of reading specs and talking to your peers.

CVS is not a configuration management system.

CVS is a source control system. The phrase "configuration management" is a marketing term, not an industry-recognized set of functions.

A true "configuration management system" would contain elements of the following:

Source control.

Dependency tracking.

Build systems (i.e. What to build and how to find things during a build. What is shared? What is local?)

Bug tracking.

Automated Testing procedures.

Release Engineering documentation and procedures.

Tape Construction.

Customer Installation.

A way for users to run different versions of the same software on the same host at the same time.

CVS provides only the first.

This section is taken from release 2.3 of the CVS FAQ.

Go to the previous, next section.

Go to the previous, next section.

Basic concepts

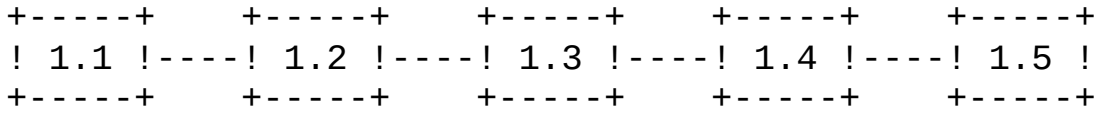
CVS stores all files in a centralized repository: a directory (such as `/usr/local/cvsroot' or `user@remotehost:/usr/local/cvsroot') which is populated with a hierarchy of files and directories. (see section Remote repositories for information about keeping the repository on a remote machine.)

Normally, you never access any of the files in the repository directly. Instead, you use CVS commands to get your own copy of the files, and then work on that copy. When you've finished a set of changes, you check (or commit) them back into the repository.

The files in the repository are organized in modules. Each module is made up of one or more files, and can include files from several directories. A typical usage is to define one module per project.

Revision numbers

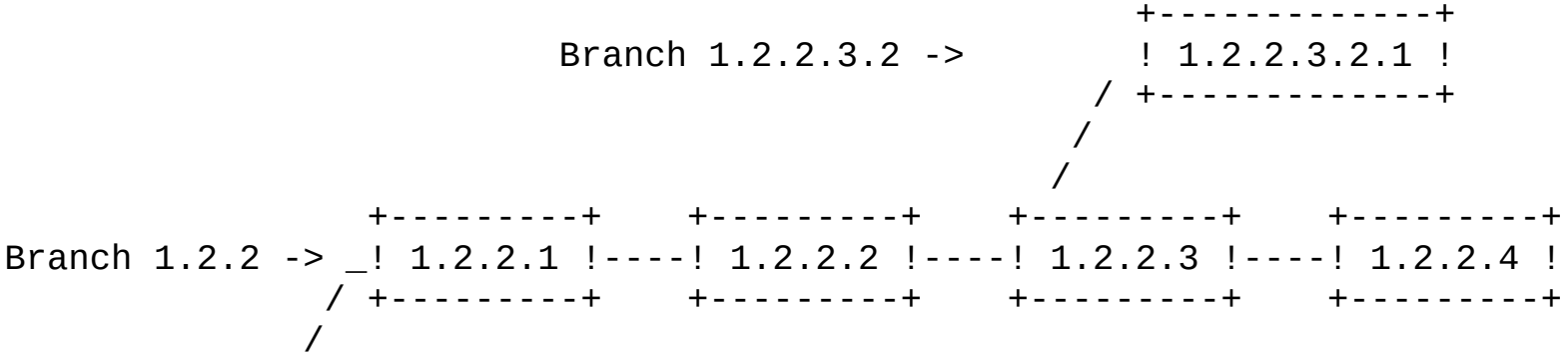
Each version of a file has a unique revision number. Revision numbers look like `1.1', `1.2', `1.3.2.2' or even `1.3.2.2.4.5'. A revision number always has an even number of period-separated decimal integers. By default revision 1.1 is the first revision of a file. Each successive revision is given a new number by increasing the rightmost number by one. The following figure displays a few revisions, with newer revisions to the right.

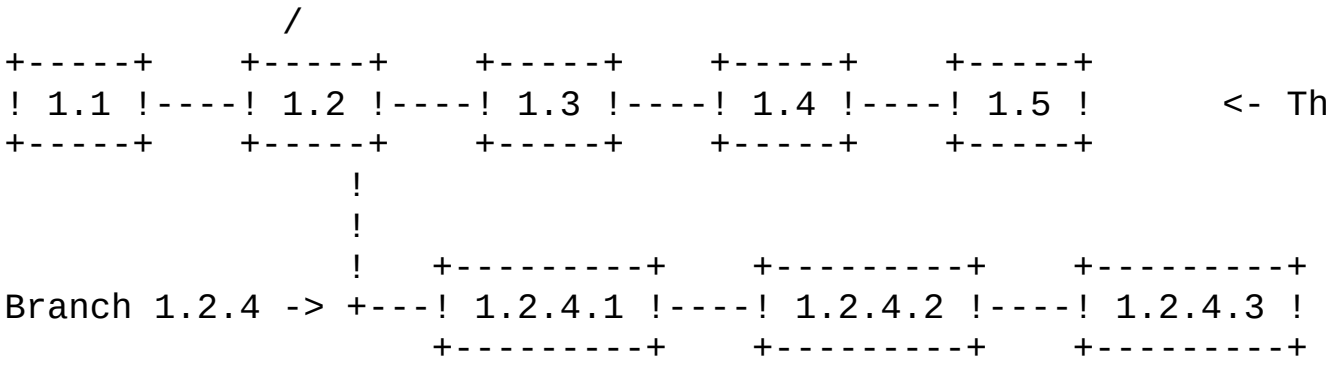


CVS is not limited to linear development. The revision tree can be split into branches, where each branch is a self-maintained line of development. Changes made on one branch can easily be moved back to the main trunk.

Each branch has a branch number, consisting of an odd number of period-separated decimal integers. The branch number is created by appending an integer to the revision number where the corresponding branch forked off. Having branch numbers allows more than one branch to be forked off from a certain revision.

All revisions on a branch have revision numbers formed by appending an ordinal number to the branch number. The following figure illustrates branching with an example.





The exact details of how the branch number is constructed is not something you normally need to be concerned about, but here is how it works: When CVS creates a branch number it picks the first unused even integer, starting with 2. So when you want to create a branch from revision 6.4 it will be numbered 6.4.2. All branch numbers ending in a zero (such as 6.4.0) are used internally by CVS (see section Magic branch numbers). The branch 1.1.1 has a special meaning. See section Tracking third-party sources.

Versions, revisions and releases

A file can have several versions, as described above. Likewise, a software product can have several versions. A software product is often given a version number such as `4.1.1'.

Versions in the first sense are called revisions in this document, and versions in the second sense are called releases. To avoid confusion, the word version is almost never used in this document.

Go to the previous, next section.

Go to the previous, next section.

A sample session

This section describes a typical work-session using CVS. It assumes that a repository is set up (see section The Repository).

Suppose you are working on a simple compiler. The source consists of a handful of C files and a `Makefile'. The compiler is called `tc' (Trivial Compiler), and the repository is set up so that there is a module called `tc'.

Getting the source

The first thing you must do is to get your own working copy of the source for `tc'. For this, you use the `checkout` command:

```
$ cvs checkout tc
```

This will create a new directory called `tc' and populate it with the source files.

```
$ cd tc
$ ls tc
CVS         Makefile    backend.c   driver.c    frontend.c  parser.c
```

The `CVS' directory is used internally by CVS. Normally, you should not modify or remove any of the files in it.

You start your favorite editor, hack away at `backend.c', and a couple of hours later you have added an optimization pass to the compiler. A note to RCS and SCCS users: There is no need to lock the files that you want to edit. See section Multiple developers for an explanation.

Committing your changes

When you have checked that the compiler is still compilable you decide to make a new version of `backend.c'.

```
$ cvs commit backend.c
```

CVS starts an editor, to allow you to enter a log message. You type in "Added an optimization pass.", save the temporary file, and exit the editor.

The environment variable `$CVSEDITOR` determines which editor is started. If `$CVSEDITOR` is not set, then if the environment variable `$EDITOR` is set, it will be used. If both `$CVSEDITOR` and `$EDITOR` are not set then the editor defaults to `vi`. If you want to avoid the overhead of starting an editor you can

specify the log message on the command line using the `-m' flag instead, like this:

```
$ cvs commit -m "Added an optimization pass" backend.c
```

Cleaning up

Before you turn to other tasks you decide to remove your working copy of tc. One acceptable way to do that is of course

```
$ cd ..
$ rm -r tc
```

but a better way is to use the `release` command (see section release--Indicate that a Module is no longer in use):

```
$ cd ..
$ cvs release -d tc
M driver.c
? tc
You have [1] altered files in this repository.
Are you sure you want to release (and delete) module `tc': n
** `release' aborted by user choice.
```

The `release` command checks that all your modifications have been committed. If history logging is enabled it also makes a note in the history file. See section The history file.

When you use the `-d' flag with `release`, it also removes your working copy.

In the example above, the `release` command wrote a couple of lines of output. `? tc' means that the file `` `tc' `` is unknown to CVS. That is nothing to worry about: `` `tc' `` is the executable compiler, and it should not be stored in the repository. See section Ignoring files via cvsignore, for information about how to make that warning go away. See section release output, for a complete explanation of all possible output from `release`.

`M driver.c' is more serious. It means that the file `` `driver.c' `` has been modified since it was checked out.

The `release` command always finishes by telling you how many modified files you have in your working copy of the sources, and then asks you for confirmation before deleting any files or making any note in the history file.

You decide to play it safe and answer n RET when `release` asks for confirmation.

Viewing differences

You do not remember modifying `driver.c', so you want to see what has happened to that file.

```
$ cd tc
$ cvs diff driver.c
```

This command runs `diff` to compare the version of `driver.c' that you checked out with your working copy. When you see the output you remember that you added a command line option that enabled the optimization pass. You check it in, and release the module.

```
$ cvs commit -m "Added an optimization pass" driver.c
Checking in driver.c;
/usr/local/cvsroot/tc/driver.c,v  <--  driver.c
new revision: 1.2; previous revision: 1.1
done
$ cd ..
$ cvs release -d tc
? tc
You have [0] altered files in this repository.
Are you sure you want to release (and delete) module `tc': y
```

Go to the previous, next section.

Go to the previous, next section.

The Repository

Figure 3 below shows a typical setup of a repository. Only directories are shown below.

```
/usr
 |
 +--local
 |   |
 |   +--cvsroot
 |   |    |
 |   |    +--CVSROOT
 |   |    |      (administrative files)
 |   |    |
 |   |    +--gnu
 |   |    |   |
 |   |    |   +--diff
 |   |    |   |   (source code to GNU diff)
 |   |    |   |
 |   |    |   +--rcs
 |   |    |   |   (source code to RCS)
 |   |    |   |
 |   |    |   +--cvs
 |   |    |       (source code to CVS)
 |   |    |
 |   |    +--yoyodyne
 |   |        |
 |   |        +--tc
 |   |        |    |
 |   |        |    +--man
 |   |        |    |
 |   |        |    +--testing
 |   |        |
 |   |        +--(other Yoyodyne software)
```

There are a couple of different ways to tell CVS where to find the repository. You can name the repository on the command line explicitly, with the `-d` (for "directory") option:

```
cvs -d /usr/local/cvsroot checkout yoyodyne/tc
```

Or you can set the `$CVSROOT` environment variable to an absolute path to the root of the repository, `` `/usr/local/cvsroot' `` in this example. To set `$CVSROOT`, all `csh` and `tcsh` users should have this line in their `` `.cshrc' `` or `` `.tcshrc' `` files:

```
setenv CVSROOT /usr/local/cvsroot
```

`sh` and `bash` users should instead have these lines in their `.profile' or `.bashrc':

```
CVSROOT=/usr/local/cvsroot
export CVSROOT
```

A repository specified with `-d` will override the `$CVSROOT` environment variable. Once you've checked a working copy out from the repository, it will remember where its repository is (the information is recorded in the `CVS/Root' file in the working copy).

The `-d` option and the `CVS/Root' file both override the `$CVSROOT` environment variable; however, CVS will complain if the `-d' argument and the `CVS/Root' file disagree.

There is nothing magical about the name `/usr/local/cvsroot'. You can choose to place the repository anywhere you like. See section Remote repositories to learn how the repository can be on a different machine than your working copy of the sources.

The repository is split in two parts. `$CVSROOT/CVSROOT' contains administrative files for CVS. The other directories contain the actual user-defined modules.

User modules

```
$CVSROOT
  |
  +--yoyodyne
  |   |
  |   +--tc
  |   |   |
          +--Makefile,v
          +--backend.c,v
          +--driver.c,v
          +--frontend.c,v
          +--parser.c,v
          +--man
          |    |
          |    +--tc.1,v
          |
          +--testing
               |
               +--testpgm.t,v
               +--test2.t,v
```

The figure above shows the contents of the `tc' module inside the repository. As you can see all file names end in `,v'. The files are history files. They contain, among other things, enough information to recreate any revision of the file, a log of all commit messages and the user-name of the person who committed the revision. CVS uses the facilities of RCS, a simpler version control system, to maintain these files. For a full description of the file format, see the `man` page rcsfile(5).

File permissions

All `,v' files are created read-only, and you should not change the permission of those files. The directories inside the repository should be writable by the persons that have permission to modify the files in each directory. This normally means that you must create a UNIX group (see group(5)) consisting of the persons that are to edit the files in a project, and set up the repository so that it is that group that owns the directory.

This means that you can only control access to files on a per-directory basis.

CVS tries to set up reasonable file permissions for new directories that are added inside the tree, but you must fix the permissions manually when a new directory should have different permissions than its parent directory.

Since CVS was not written to be run setuid, it is unsafe to try to run it setuid. You cannot use the setuid features of RCS together with CVS.

The administrative files

The directory `$CVSROOT/CVSROOT' contains some administrative files. See section Reference manual for the Administrative files, for a complete description. You can use CVS without any of these files, but some commands work better when at least the `modules' file is properly set up.

The most important of these files is the `modules' file. It defines all modules in the repository. This is a sample `modules' file.

```
CVSROOT         CVSROOT
modules         CVSROOT modules
cvs             gnu/cvs
rcs             gnu/rcs
diff            gnu/diff
tc              yoyodyne/tc
```

The `modules' file is line oriented. In its simplest form each line contains the name of the module, whitespace, and the directory where the module resides. The directory is a path relative to $CVSROOT. The last for lines in the example above are examples of such lines.

The line that defines the module called `modules' uses features that are not explained here. See section The modules file, for a full explanation of all the available features.

Editing administrative files

You edit the administrative files in the same way that you would edit any other module. Use `cvs checkout CVSROOT' to get a working copy, edit it, and commit your changes in the normal way.

It is possible to commit an erroneous administrative file. You can often fix the error and check in a new revision, but sometimes a particularly bad error in the administrative file makes it impossible to commit new revisions.

Multiple repositories

In some situations it is a good idea to have more than one repository, for instance if you have two development groups that work on separate projects without sharing any code. All you have to do to have several repositories is to specify the appropriate repository, using the `CVSROOT` environment variable, the `-d' option to CVS, or (once you have checked out a working directories) by simply allowing CVS to use the repository that was used to check out the working directory (see section The Repository).

Notwithstanding, it can be confusing to have two or more repositories.

None of the examples in this manual show multiple repositories.

Creating a repository

See the instructions in the `INSTALL' file in the CVS distribution.

Remote repositories

Your working copy of the sources can be on a different machine than the repository. Generally, using a remote repository is just like using a local one, except that the format of the repository name is:

```
user@hostname:/path/to/repository
```

The details of exactly what needs to be set up depend on how you are connecting to the server.

Connecting with rsh

CVS uses the `rsh' protocol to perform these operations, so the remote user host needs to have a `.rhosts' file which grants access to the local user.

For example, suppose you are the user `mozart' on the local machine `anklet.grunge.com', and the server machine is `chainsaw.brickyard.com'. On chainsaw, put the following line into the file `.rhosts' in `bach''s home directory:

```
anklet.grunge.com  mozart
```

Then test that `rsh` is working with

```
rsh -l bach chainsaw.brickyard.com echo $PATH
```

Next you have to make sure that `rsh` will be able to find the server. Make sure that the path which `rsh` printed in the above example includes the directory containing a program named `cvs` which is the server. You need to set the path in `.bashrc', `.cshrc', etc., not `.login' or `.profile'. Alternately, you can set the environment variable `CVS_SERVER` on the client machine to the filename of the server you want to use, for example `/usr/local/bin/cvs-1.6'.

There is no need to edit `inetd.conf` or start a CVS server daemon.

Continuing our example, supposing you want to access the module `foo' in the repository `/usr/local/cvsroot/', on machine `chainsaw.brickyard.com', you are ready to go:

```
cvs -d bach@chainsaw.brickyard.com:/user/local/cvsroot checkout foo
```

(The `bach@' can be omitted if the username is the same on both the local and remote hosts.)

Direct connection with password authentication

The CVS client can also connect to the server using a password protocol. This is particularly useful if using `rsh` is not feasible (for example, the server is behind a firewall), and Kerberos also is not available.

To use this method, it is necessary to make some adjustments on both the server and client sides.

Setting up the server for password authentication

On the server side, the file `/etc/inetd.conf' needs to be edited so `inetd` knows to run the command `cvs pserver` when it receives a connection on the right port. By default, the port number is 2401; it would be different if your client were compiled with `CVS_AUTH_PORT` defined to something else, though.

If your `inetd` allows raw port numbers in `/etc/inetd.conf', then the following (all on a single line in `inetd.conf') should be sufficient:

```
2401  stream  tcp  nowait  root  /usr/local/bin/cvs
cvs -b /usr/local/bin pserver
```

The `-b' option specifies the directory which contains the RCS binaries on the server.

If your `inetd` wants a symbolic service name instead of a raw port number, then put this in `/etc/services':

```
cvspserver      2401/tcp
```

and put `cvspserver` instead of `2401` in `inetd.conf'.

Once the above is taken care of, restart your `inetd`, or do whatever is necessary to force it to reread its initialization files.

Because the client stores and transmits passwords in cleartext (almost--see section Security considerations with password authentication for details), a separate CVS password file may be used, so people don't compromise their regular passwords when they access the repository. This file is `$CVSROOT/CVSROOT/passwd' (see section The administrative files). Its format is similar to `/etc/passwd', except that it only has two fields, username and password. For example:

```
bach:ULtgRLXo7NRxs
cwang:1sOp854gDF3DY
```

The password is encrypted according to the standard Unix `crypt()` function, so it is possible to paste in passwords directly from regular Unix `passwd' files.

When authenticating a password, the server first checks for the user in the CVS `passwd' file. If it finds the

user, it compares against that password. If it does not find the user, or if the CVS `passwd' file does not, then the server tries to match the password using the system's user-lookup routine. When using the CVS `passwd' file, the server runs under as the username specified in the the third argument in the entry, or as the first argument if there is no third argument (in this way CVS allows imaginary usernames provided the CVS `passwd' file indicates corresponding valid system usernames). In any case, CVS will have no privileges which the (valid) user would not have.

Right now, the only way to put a password in the CVS `passwd' file is to paste it there from somewhere else. Someday, there may be a `cvs passwd` command.

Using the client with password authentication

Before connecting to the server, the client must log in with the command `cvs login`. Logging in verifies a password with the server, and also records the password for later transactions with the server. The `cvs login` command needs to know the username, server hostname, and full repository path, and it gets this information from the repository argument or the `CVSROOT` environment variable.

`cvs login` is interactive -- it prompts for a password:

```
cvs -d bach@chainsaw.brickyard.com:/usr/local/cvsroot login
CVS password:
```

The password is checked with the server; if it is correct, the `login` succeeds, else it fails, complaining that the password was incorrect.

Once you have logged in, you can force CVS to connect directly to the server and authenticate with the stored password by prefixing the repository with `:pserver:':

```
cvs -d :pserver:bach@chainsaw.brickyard.com:/usr/local/cvsroot checkout foo
```

The `:pserver:' is necessary because without it, CVS will assume it should use `rsh` to connect with the server (see section Connecting with rsh). (Once you have a working copy checked out and are running CVS commands from within it, there is no longer any need to specify the repository explicitly, because CVS records it in the working copy's `CVS' subdirectory.)

Passwords are stored by default in the file `$HOME/.cvspass'. Its format is human-readable, but don't edit it unless you know what you are doing. The passwords are not stored in cleartext, but are trivially encoded to protect them from "innocent" compromise (i.e., inadvertently being seen by a system administrator who happens to look at that file).

The `CVS_PASSFILE` environment variable overrides this default. If you use this variable, make sure you set it *before* `cvs login` is run. If you were to set it after running `cvs login`, then later CVS commands would be unable to look up the password for transmission to the server.

The `CVS_PASSWORD` environment variable overrides *all* stored passwords. If it is set, CVS will use it for all password-authenticated connections.

Security considerations with password authentication

The passwords are stored on the client side in a trivial encoding of the cleartext, and transmitted in the same encoding. The encoding is done only to prevent inadvertent password compromises (i.e., a system administrator

accidentally looking at the file), and will not prevent even a naive attacker from gaining the password.

The separate CVS password file (see section Setting up the server for password authentication) allows people to use a different password for repository access than for login access. On the other hand, once a user has access to the repository, she can execute programs on the server system through a variety of means. Thus, repository access implies fairly broad system access as well. It might be possible to modify CVS to prevent that, but no one has done so as of this writing. Furthermore, there may be other ways in which having access to CVS allows people to gain more general access to the system; noone has done a careful audit.

In summary, anyone who gets the password gets repository access, and some measure of general system access as well. The password is available to anyone who can sniff network packets or read a protected (i.e., user read-only) file. If you want real security, get Kerberos.

Direct connection with kerberos

The main disadvantage of using rsh is that all the data needs to pass through additional programs, so it may be slower. So if you have kerberos installed you can connect via a direct TCP connection, authenticating with kerberos (note that the data transmitted is *not* encrypted).

To do this, CVS needs to be compiled with kerberos support; when configuring CVS it tries to detect whether kerberos is present or you can use the `--with-krb4' flag to configure.

You need to edit `inetd.conf` on the server machine to run `cvs kserver`. The client uses port 1999 by default; if you want to use another port specify it in the `CVS_CLIENT_PORT` environment variable on the client. Set `CVS_CLIENT_PORT` to `-1' to force an rsh connection.

When you want to use CVS, get a ticket in the usual way (generally `kinit`); it must be a ticket which allows you to log into the server machine. Then you are ready to go:

```
cvs -d chainsaw.brickyard.com:/user/local/cvsroot checkout foo
```

If CVS fails to connect, it will fall back to trying rsh.

Go to the previous, next section.

Go to the previous, next section.

Starting a project with CVS

Since CVS 1.x is bad at renaming files and moving them between directories, the first thing you do when you start a new project should be to think through your file organization. It is not impossible--just awkward--to rename or move files. See section Moving and renaming files.

What to do next depends on the situation at hand.

Setting up the files

The first step is to create the files inside the repository. This can be done in a couple of different ways.

Creating a module from a number of files

When you begin using CVS, you will probably already have several projects that can be put under CVS control. In these cases the easiest way is to use the `import` command. An example is probably the easiest way to explain how to use it. If the files you want to install in CVS reside in `dir', and you want them to appear in the repository as `$CVSROOT/yoyodyne/dir', you can do this:

```
$ cd dir
$ cvs import -m "Imported sources" yoyodyne/dir yoyo start
```

Unless you supply a log message with the `-m' flag, CVS starts an editor and prompts for a message. The string `yoyo' is a vendor tag, and `start' is a release tag. They may fill no purpose in this context, but since CVS requires them they must be present. See section Tracking third-party sources, for more information about them.

You can now verify that it worked, and remove your original source directory.

```
$ cd ..
$ mv dir dir.orig
$ cvs checkout yoyodyne/dir       # Explanation below
$ ls -R yoyodyne
$ rm -r dir.orig
```

Erasing the original sources is a good idea, to make sure that you do not accidentally edit them in dir, bypassing CVS. Of course, it would be wise to make sure that you have a backup of the sources before you remove them.

The `checkout` command can either take a module name as argument (as it has done in all previous examples) or a path name relative to `$CVSROOT`, as it did in the example above.

It is a good idea to check that the permissions CVS sets on the directories inside `$CVSROOT' are

reasonable, and that they belong to the proper groups. See section File permissions.

Creating Files From Other Version Control Systems

If you have a project which you are maintaining with another version control system, such as RCS, you may wish to put the files from that project into CVS, and preserve the revision history of the files.

From RCS

> If you have been using RCS, find the RCS files--usually a file named `foo.c' will have its RCS file in `RCS/foo.c,v' (but it could be other places; consult the RCS documentation for details). Then create the appropriate directories in CVS if they do not already exist. Then copy the files into the appropriate directories in the CVS repository (the name in the repository must be the name of the source file with `,v' added; the files go directly in the appopriate directory of the repository, not in an `RCS' subdirectory). This is one of the few times when it is a good idea to access the CVS repository directly, rather than using CVS commands. Then you are ready to check out a new working directory.

From another version control system

> Many version control systems have the ability to export RCS files in the standard format. If yours does, export the RCS files and then follow the above instructions.

From SCCS

> There is a script in the `contrib' directory of the CVS source distribution called `sccs2rcs' which converts SCCS files to RCS files. Note: you must run it on a machine which has both SCCS and RCS installed, and like everything else in contrib it is unsupported (your mileage may vary).

Creating a module from scratch

For a new project, the easiest thing to do is probably to create an empty directory structure, like this:

```
$ mkdir tc
$ mkdir tc/man
$ mkdir tc/testing
```

After that, you use the `import` command to create the corresponding (empty) directory structure inside the repository:

```
$ cd tc
$ cvs import -m "Created directory structure" yoyodyne/dir yoyo start
```

Then, use `add` to add files (and new directories) as they appear.

Check that the permissions CVS sets on the directories inside `$CVSROOT' are reasonable.

Defining the module

The next step is to define the module in the `modules' file. This is not strictly necessary, but modules can be convenient in grouping together related files and directories.

In simple cases these steps are sufficient to define a module.

1. Get a working copy of the modules file.

   ```
   $ cvs checkout modules
   $ cd modules
   ```

2. Edit the file and insert a line that defines the module. See section The administrative files, for an introduction. See section The modules file, for a full description of the modules file. You can use the following line to define the module `tc':

   ```
   tc   yoyodyne/tc
   ```

3. Commit your changes to the modules file.

   ```
   $ cvs commit -m "Added the tc module." modules
   ```

4. Release the modules module.

   ```
   $ cd ..
   $ cvs release -d modules
   ```

Go to the previous, next section.

Go to the previous, next section.

Multiple developers

When more than one person works on a software project things often get complicated. Often, two people try to edit the same file simultaneously. Some other version control systems (including RCS and SCCS) try to solve that particular problem by introducing file locking, so that only one person can edit each file at a time. Unfortunately, file locking can be very counter-productive. If two persons want to edit different parts of a file, there may be no reason to prevent either of them from doing so.

CVS does not use file locking. Instead, it allows many people to edit their own working copy of a file simultaneously. The first person that commits his changes has no automatic way of knowing that another has started to edit it. Others will get an error message when they try to commit the file. They must then use CVS commands to bring their working copy up to date with the repository revision. This process is almost automatic, and explained in this chapter.

There are many ways to organize a team of developers. CVS does not try to enforce a certain organization. It is a tool that can be used in several ways. It is often useful to inform the group of commits you have done. CVS has several ways of automating that process. See section Informing others about commits. See section Revision management, for more tips on how to use CVS.

File status

After you have checked out a file out from CVS, it is in one of these four states:

Up-to-date
: The file is identical with the latest revision in the repository.

Locally modified
: You have edited the file, and not yet committed your changes.

Needing update
: Someone else has committed a newer revision to the repository.

Needing merge
: Someone else have committed a newer revision to the repository, and you have also made modifications to the file.

You can use the `status` command to find out the status of a given file. See section status--Status info on the revisions.

Bringing a file up to date

When you want to update or merge a file, use the `update` command. For files that are not up to date this is roughly equivalent to a `checkout` command: the newest revision of the file is extracted from the repository and put in your working copy of the module.

Your modifications to a file are never lost when you use `update`. If no newer revision exists, running

`update` has no effect. If you have edited the file, and a newer revision is available, CVS will merge all changes into your working copy.

For instance, imagine that you checked out revision 1.4 and started editing it. In the meantime someone else committed revision 1.5, and shortly after that revision 1.6. If you run `update` on the file now, CVS will incorporate all changes between revision 1.4 and 1.6 into your file.

If any of the changes between 1.4 and 1.6 were made too close to any of the changes you have made, an overlap occurs. In such cases a warning is printed, and the resulting file includes both versions of the lines that overlap, delimited by special markers. See section update--Bring work tree in sync with repository, for a complete description of the `update` command.

Conflicts example

Suppose revision 1.4 of `` `driver.c' `` contains this:

```
#include <stdio.h>

void main()
{
    parse();
    if (nerr == 0)
        gencode();
    else
        fprintf(stderr, "No code generated.\n");
    exit(nerr == 0 ? 0 : 1);
}
```

Revision 1.6 of `` `driver.c' `` contains this:

```
#include <stdio.h>

int main(int argc,
         char **argv)
{
    parse();
    if (argc != 1)
    {
        fprintf(stderr, "tc: No args expected.\n");
        exit(1);
    }
    if (nerr == 0)
        gencode();
    else
        fprintf(stderr, "No code generated.\n");
    exit(!!nerr);
}
```

Your working copy of `driver.c', based on revision 1.4, contains this before you run `cvs update':

```
#include <stdlib.h>
#include <stdio.h>

void main()
{
    init_scanner();
    parse();
    if (nerr == 0)
        gencode();
    else
        fprintf(stderr, "No code generated.\n");
    exit(nerr == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

You run `cvs update':

```
$ cvs update driver.c
RCS file: /usr/local/cvsroot/yoyodyne/tc/driver.c,v
retrieving revision 1.4
retrieving revision 1.6
Merging differences between 1.4 and 1.6 into driver.c
rcsmerge warning: overlaps during merge
cvs update: conflicts found in driver.c
C driver.c
```

CVS tells you that there were some conflicts. Your original working file is saved unmodified in `.#driver.c.1.4'. The new version of `driver.c' contains this:

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc,
         char **argv)
{
    init_scanner();
    parse();
    if (argc != 1)
    {
        fprintf(stderr, "tc: No args expected.\n");
        exit(1);
    }
    if (nerr == 0)
        gencode();
    else
        fprintf(stderr, "No code generated.\n");
<<<<<<< driver.c
```


```
    exit(nerr == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
=======
    exit(!!nerr);
>>>>>>> 1.6
}
```

Note how all non-overlapping modifications are incorporated in your working copy, and that the overlapping section is clearly marked with `<<<<<<<', `=======' and `>>>>>>>'.

You resolve the conflict by editing the file, removing the markers and the erroneous line. Suppose you end up with this file:

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc,
         char **argv)
{
    init_scanner();
    parse();
    if (argc != 1)
    {
        fprintf(stderr, "tc: No args expected.\n");
        exit(1);
    }
    if (nerr == 0)
        gencode();
    else
        fprintf(stderr, "No code generated.\n");
    exit(nerr == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

You can now go ahead and commit this as revision 1.7.

```
$ cvs commit -m "Initialize scanner. Use symbolic exit values." driver.c
Checking in driver.c;
/usr/local/cvsroot/yoyodyne/tc/driver.c,v  <--  driver.c
new revision: 1.7; previous revision: 1.6
done
```

If you use release 1.04 or later of pcl-cvs (a GNU Emacs front-end for CVS) you can use an Emacs package called emerge to help you resolve conflicts. See the documentation for pcl-cvs.

Informing others about commits

It is often useful to inform others when you commit a new revision of a file. The `-i' option of the `modules' file, or the `loginfo' file, can be used to automate this process. See section The modules file. See section Loginfo. You can use these features of CVS to, for instance, instruct CVS to mail a message to all

developers, or post a message to a local newsgroup.

Several developers simultaneously attempting to run CVS

If several developers try to run CVS at the same time, one may get the following message:

```
[11:43:23] waiting for bach's lock in /usr/local/cvsroot/foo
```

CVS will try again every 30 seconds, and either continue with the operation or print the message again, if it still needs to wait. If a lock seems to stick around for an undue amount of time, find the person holding the lock and ask them about the cvs command they are running. If they aren't running a cvs command, look for and remove files starting with `` `#cvs.tfl' ``, `` `#cvs.rfl' ``, or `` `#cvs.wfl' `` from the repository.

Note that these locks are to protect CVS's internal data structures and have no relationship to the word lock in the sense used by RCS--a way to prevent other developers from working on a particular file.

Any number of people can be reading from a given repository at a time; only when someone is writing do the locks prevent other people from reading or writing.

One might hope for the following property

```
If someone commits some changes in one cvs command,
then an update by someone else will either get all the
changes, or none of them.
```

but CVS does *not* have this property. For example, given the files

```
a/one.c
a/two.c
b/three.c
b/four.c
```

if someone runs

```
cvs ci a/two.c b/three.c
```

and someone else runs `cvs update` at the same time, the person running `update` might get only the change to `` `b/three.c' `` and not the change to `` `a/two.c' ``.

Mechanisms to track who is editing files

For many groups, use of CVS in its default mode is perfectly satisfactory. Users may sometimes go to check in a modification only to find that another modification has intervened, but they deal with it and proceed with their check in. Other groups prefer to be able to know who is editing what files, so that if two people try to edit the same file they can choose to talk about who is doing what when rather than be surprised at check in time. The features in this section allow such coordination, while retaining the ability of two developers to edit the

same file at the same time.

For maximum benefit developers should use `cvs edit` (not `chmod`) to make files read-write to edit them, and `cvs release` (not `rm`) to discard a working directory which is no longer in use, but CVS is not able to enforce this behavior.

Telling CVS to watch certain files

To enable the watch features, you first specify that certain files are to be watched.

Command: **cvs watch on** *[-l] files ...*

Specify that developers should run `cvs edit` before editing files. CVS will create working copies of files read-only, to remind developers to run the `cvs edit` command before working on them.

If files includes the name of a directory, CVS arranges to watch all files added to the corresponding repository directory, and sets a default for files added in the future; this allows the user to set notification policies on a per-directory basis. The contents of the directory are processed recursively, unless the `-l` option is given.

If files is omitted, it defaults to the current directory.

Command: **cvs watch off** *[-l] files ...*

Do not provide notification about work on files. CVS will create working copies of files read-write.

The files and `-l` arguments are processed as for `cvs watch on`.

Telling CVS to notify you

You can tell CVS that you want to receive notifications about various actions taken on a file. You can do this without using `cvs watch on` for the file, but generally you will want to use `cvs watch on`, so that developers use the `cvs edit` command.

Command: **cvs watch add** *[-a action] [-l] files ...*

Add the current user to the list of people to receive notification of work done on files.

The `-a` option specifies what kinds of events CVS should notify the user about. action is one of the following:

`edit`
: Another user has applied the `cvs edit` command (described below) to a file.

`unedit`
: Another user has applied the `cvs unedit` command (described below) or the `cvs release` command to a file, or has deleted the file and allowed `cvs update` to recreate it.

`commit`
: Another user has committed changes to a file.

`all`
: All of the above.

`none`
: None of the above. (This is useful with `cvs edit`, described below.)

The `-a` option may appear more than once, or not at all. If omitted, the action defaults to `all`.

The files and `-l` option are processed as for the `cvs watch` commands.

Command: **cvs watch remove** *[-a action] [-l] files ...*

Remove a notification request established using `cvs watch add`; the arguments are the same. If the `-a` option is present, only watches for the specified actions are removed.

When the conditions exist for notification, CVS calls the `` `notify' `` administrative file, passing it the user to receive the notification and the user who is taking the action which results in notification. Normally `` `notify' `` will just send an email message.

Note that if you set this up in the straightforward way, users receive notifications on the server machine. One could of course write a `` `notify' `` script which directed notifications elsewhere, but to make this easy, CVS allows you to associate a notification address for each user. To do so create a file `` `users' `` in `` `CVSROOT' `` with a line for each user in the format user:value. Then instead of passing the name of the user to be notified to `` `notify' ``, CVS will pass the value (normally an email address on some other machine).

How to edit a file which is being watched

Since a file which is being watched is checked out read-only, you cannot simply edit it. To make it read-write, and inform others that you are planning to edit it, use the `cvs edit` command.

Command: **cvs edit** *[options] files ...*

Prepare to edit the working files files. CVS makes the files read-write, and notifies users who have requested `edit` notification for any of files.

The `cvs edit` command accepts the same options as the `cvs watch add` command, and establishes a temporary watch for the user on files; CVS will remove the watch when files are `unedit`ed or `commit`ted. If the user does not wish to receive notifications, she should specify `-a none`.

The files and `-l` option are processed as for the `cvs watch` commands.

Normally when you are done with a set of changes, you use the `cvs commit` command, which checks in your changes and returns the watched files to their usual read-only state. But if you instead decide to abandon your changes, or not to make any changes, you can use the `cvs unedit` command.

Command: **cvs unedit** *[-l] files ...*

Abandon work on the working files files, and revert them to the repository versions on which they are based. CVS makes those files read-only for which users have requested notification using `cvs watch on`. CVS notifies users who have requested `unedit` notification for any of files.

The files and `-l` option are processed as for the `cvs watch` commands.

When using client/server CVS, you can use the `cvs edit` and `cvs unedit` commands even if CVS is unable to succesfully communicate with the server; the notifications will be sent upon the next successful CVS command.

Information about who is watching and editing

Command: **cvs watchers** *[-l] files ...*

List the users currently watching changes to files. The report includes the files being watched, and the mail address of each watcher.

The files and `-l` arguments are processed as for the `cvs watch` commands.

Command: **cvs editors** *[-l] files ...*

List the users currently working on files. The report includes the mail address of each user, the time when the user began working with the file, and the host and path of the working directory containing the file.

The files and `-l` arguments are processed as for the `cvs watch` commands.

Using watches with old versions of CVS

If you use the watch features on a repository, it creates `` `CVS' `` directories in the repository and stores the information about watches in that directory. If you attempt to use CVS 1.6 or earlier with the repository, you get an error message such as

```
cvs update: cannot open CVS/Entries for reading: No such file or directory
```

and your operation will likely be aborted. To use the watch features, you must upgrade all copies of CVS which use that repository in local or server mode. If you cannot upgrade, use the `watch off` and `watch remove` commands to remove all watches, and that will restore the repository to a state which CVS 1.6 can cope with.

Go to the previous, next section.

Go to the previous, next section.

Branches

So far, all revisions shown in this manual have been on the main trunk of the revision tree, i.e., all revision numbers have been of the form x.y. One useful feature, especially when maintaining several releases of a software product at once, is the ability to make branches on the revision tree. Tags, symbolic names for revisions, will also be introduced in this chapter.

Tags--Symbolic revisions

The revision numbers live a life of their own. They need not have anything at all to do with the release numbers of your software product. Depending on how you use CVS the revision numbers might change several times between two releases. As an example, some of the source files that make up RCS 5.6 have the following revision numbers:

```
ci.c            5.21
co.c            5.9
ident.c         5.3
rcs.c           5.12
rcsbase.h       5.11
rcsdiff.c       5.10
rcsedit.c       5.11
rcsfcmp.c       5.9
rcsgen.c        5.10
rcslex.c        5.11
rcsmap.c        5.2
rcsutil.c       5.10
```

You can use the `tag` command to give a symbolic name to a certain revision of a file. You can use the `-v' flag to the `status` command to see all tags that a file has, and which revision numbers they represent. Tag names can contain uppercase and lowercase letters, digits, `-', and `_'. The two tag names BASE and HEAD are reserved for use by CVS. It is expected that future names which are special to CVS will contain characters such as `%' or `=', rather than being named analogously to BASE and HEAD, to avoid conflicts with actual tag names.

The following example shows how you can add a tag to a file. The commands must be issued inside your working copy of the module. That is, you should issue the command in the directory where `backend.c' resides.

```
$ cvs tag release-0-4 backend.c
T backend.c
$ cvs status -v backend.c
===================================================================
File: backend.c         Status: Up-to-date
```


```
    Version:            1.4     Tue Dec  1 14:39:01 1992
    RCS Version:        1.4     /usr/local/cvsroot/yoyodyne/tc/backend.
    Sticky Tag:         (none)
    Sticky Date:        (none)
    Sticky Options:     (none)

    Existing Tags:
        release-0-4                     (revision: 1.4)
```

There is seldom reason to tag a file in isolation. A more common use is to tag all the files that constitute a module with the same tag at strategic points in the development life-cycle, such as when a release is made.

```
$ cvs tag release-1-0 .
cvs tag: Tagging .
T Makefile
T backend.c
T driver.c
T frontend.c
T parser.c
```

(When you give CVS a directory as argument, it generally applies the operation to all the files in that directory, and (recursively), to any subdirectories that it may contain. See section Recursive behavior.)

The `checkout` command has a flag, `-r', that lets you check out a certain revision of a module. This flag makes it easy to retrieve the sources that make up release 1.0 of the module `tc' at any time in the future:

```
$ cvs checkout -r release-1-0 tc
```

This is useful, for instance, if someone claims that there is a bug in that release, but you cannot find the bug in the current working copy.

You can also check out a module as it was at any given date. See section checkout options.

When you tag more than one file with the same tag you can think about the tag as "a curve drawn through a matrix of filename vs. revision number." Say we have 5 files with the following revisions:

```
        file1   file2   file3   file4   file5

        1.1     1.1     1.1     1.1  /--1.1*      <-*-  TAG
        1.2*-   1.2     1.2    -1.2*-
        1.3  \- 1.3*-   1.3   / 1.3
        1.4         \   1.4  /  1.4
                     \-1.5*-   1.5
                       1.6
```

At some time in the past, the * versions were tagged. You can think of the tag as a handle attached to the curve drawn through the tagged revisions. When you pull on the handle, you get all the tagged revisions. Another way to look at it is that you "sight" through a set of revisions that is "flat" along the tagged revisions,

like this:

```
        file1   file2   file3   file4   file5

                        1.1
                        1.2
                1.1     1.3                      _
        1.1     1.2     1.4     1.1              /
        1.2*----1.3*----1.5*----1.2*----1.1     (--- <--- Look here
        1.3             1.6     1.3              \_
        1.4                     1.4
                                1.5
```

What branches are good for

Suppose that release 1.0 of tc has been made. You are continuing to develop tc, planning to create release 1.1 in a couple of months. After a while your customers start to complain about a fatal bug. You check out release 1.0 (see section Tags--Symbolic revisions) and find the bug (which turns out to have a trivial fix). However, the current revision of the sources are in a state of flux and are not expected to be stable for at least another month. There is no way to make a bugfix release based on the newest sources.

The thing to do in a situation like this is to create a branch on the revision trees for all the files that make up release 1.0 of tc. You can then make modifications to the branch without disturbing the main trunk. When the modifications are finished you can select to either incorporate them on the main trunk, or leave them on the branch.

Creating a branch

The `rtag` command can be used to create a branch. The `rtag` command is much like `tag`, but it does not require that you have a working copy of the module. See section rtag--Add a tag to the RCS file. (You can also use the `tag` command; see section tag--Add a symbolic tag to checked out version of RCS file).

```
$ cvs rtag -b -r release-1-0 release-1-0-patches tc
```

The `-b' flag makes `rtag` create a branch (rather than just a symbolic revision name). `-r release-1-0' says that this branch should be rooted at the node (in the revision tree) that corresponds to the tag `release-1-0'. Note that the numeric revision number that matches `release-1-0' will probably be different from file to file. The name of the new branch is `release-1-0-patches', and the module affected is `tc'.

To fix the problem in release 1.0, you need a working copy of the branch you just created.

```
$ cvs checkout -r release-1-0-patches tc
$ cvs status -v driver.c backend.c
===================================================================
File: driver.c          Status: Up-to-date
```


```
    Version:            1.7     Sat Dec  5 18:25:54 1992
    RCS Version:        1.7     /usr/local/cvsroot/yoyodyne/tc/driver.c
    Sticky Tag:         release-1-0-patches (branch: 1.7.2)
    Sticky Date:        (none)
    Sticky Options:     (none)

    Existing Tags:
        release-1-0-patches             (branch: 1.7.2)
        release-1-0                     (revision: 1.7)

===================================================================
File: backend.c         Status: Up-to-date

    Version:            1.4     Tue Dec  1 14:39:01 1992
    RCS Version:        1.4     /usr/local/cvsroot/yoyodyne/tc/backend.c,v
    Sticky Tag:         release-1-0-patches (branch: 1.4.2)
    Sticky Date:        (none)
    Sticky Options:     (none)

    Existing Tags:
        release-1-0-patches             (branch: 1.4.2)
        release-1-0                     (revision: 1.4)
        release-0-4                     (revision: 1.4)
```

As the output from the `status` command shows the branch number is created by adding a digit at the tail of the revision number it is based on. (If `release-1-0' corresponds to revision 1.4, the branch's revision number will be 1.4.2. For obscure reasons CVS always gives branches even numbers, starting at 2. See section Revision numbers).

Sticky tags

The `-r release-1-0-patches' flag that was given to `checkout` in the previous example is sticky, that is, it will apply to subsequent commands in this directory. If you commit any modifications, they are committed on the branch. You can later merge the modifications into the main trunk. See section Merging.

You can use the `status` command to see what sticky tags or dates are set:

```
$ vi driver.c   # Fix the bugs
$ cvs commit -m "Fixed initialization bug" driver.c
Checking in driver.c;
/usr/local/cvsroot/yoyodyne/tc/driver.c,v  <--  driver.c
new revision: 1.7.2.1; previous revision: 1.7
done
$ cvs status -v driver.c
===================================================================
File: driver.c          Status: Up-to-date
```


```
   Version:            1.7.2.1 Sat Dec  5 19:35:03 1992
   RCS Version:        1.7.2.1 /usr/local/cvsroot/yoyodyne/tc/driver.c,v
   Sticky Tag:         release-1-0-patches (branch: 1.7.2)
   Sticky Date:        (none)
   Sticky Options:     (none)

   Existing Tags:
       release-1-0-patches             (branch: 1.7.2)
       release-1-0                     (revision: 1.7)
```

The sticky tags will remain on your working files until you delete them with `cvs update -A'. The `-A' option retrieves the version of the file from the head of the trunk, and forgets any sticky tags, dates, or options.

Sticky tags are not just for branches. If you check out a certain revision (such as 1.4) it will also become sticky. Subsequent `cvs update' will not retrieve the latest revision until you reset the tag with `cvs update -A'. Likewise, use of the `-D' option to `update` or `checkout` sets a sticky date, which, similarly, causes that date to be used for future retrievals.

Many times you will want to retrieve an old version of a file without setting a sticky tag. The way to do that is with the `-p' option to `checkout` or `update`, which sends the contents of the file to standard output. For example, suppose you have a file named `file1' which existed as revision 1.1, and you then removed it (thus adding a dead revision 1.2). Now suppose you want to add it again, with the same contents it had previously. Here is how to do it:

```
$ cvs update -p -r 1.1 file1 >file1
===================================================================
Checking out file1
RCS:  /tmp/cvs-sanity/cvsroot/first-dir/Attic/file1,v
VERS: 1.1
***************
$ cvs add file1
cvs add: version 1.2 of `file1' will be resurrected
cvs add: use 'cvs commit' to add this file permanently
$ cvs commit -m test
Checking in file1;
/tmp/cvs-sanity/cvsroot/first-dir/file1,v  <--  file1
new revision: 1.3; previous revision: 1.2
done
$
```

Go to the previous, next section.

Go to the previous, next section.

Merging

You can include the changes made between any two revisions into your working copy, by merging. You can then commit that revision, and thus effectively copy the changes onto another branch.

Merging an entire branch

You can merge changes made on a branch into your working copy by giving the `-j branch' flag to the `update` command. With one `-j branch' option it merges the changes made between the point where the branch forked and newest revision on that branch (into your working copy).

The `-j' stands for "join".

Consider this revision tree:

```
+-----+    +-----+    +-----+    +-----+
! 1.1 !----! 1.2 !----! 1.3 !----! 1.4 !      <- The main trunk
+-----+    +-----+    +-----+    +-----+
                !
                !
                !   +---------+    +---------+
Branch R1fix -> +---! 1.2.2.1 !----! 1.2.2.2 !
                    +---------+    +---------+
```

The branch 1.2.2 has been given the tag (symbolic name) `R1fix'. The following example assumes that the module `mod' contains only one file, `m.c'.

```
$ cvs checkout mod                # Retrieve the latest revision, 1.4

$ cvs update -j R1fix m.c         # Merge all changes made on the branch,
                                  # i.e. the changes between revision 1.2
                                  # and 1.2.2.2, into your working copy
                                  # of the file.

$ cvs commit -m "Included R1fix" # Create revision 1.5.
```

A conflict can result from a merge operation. If that happens, you should resolve it before committing the new revision. See section Conflicts example.

The `checkout` command also supports the `-j branch' flag. The same effect as above could be achieved with this:

```
$ cvs checkout -j R1fix mod
$ cvs commit -m "Included R1fix"
```

Merging from a branch several times

Continuing our example, the revision tree now looks like this:

```
+-----+    +-----+    +-----+    +-----+    +-----+
! 1.1 !----! 1.2 !----! 1.3 !----! 1.4 !----! 1.5 !      <- The main trunk
+-----+    +-----+    +-----+    +-----+    +-----+
                !                             *
                !                           *
                !   +---------+    +---------+
Branch R1fix -> +---! 1.2.2.1 !----! 1.2.2.2 !
                    +---------+    +---------+
```

where the starred line represents the merge from the `R1fix' branch to the main trunk, as just discussed.

Now suppose that development continues on the `R1fix' branch:

```
+-----+    +-----+    +-----+    +-----+    +-----+
! 1.1 !----! 1.2 !----! 1.3 !----! 1.4 !----! 1.5 !      <- The main trunk
+-----+    +-----+    +-----+    +-----+    +-----+
                !                             *
                !                           *
                !   +---------+    +---------+    +---------+
Branch R1fix -> +---! 1.2.2.1 !----! 1.2.2.2 !----! 1.2.2.3 !
                    +---------+    +---------+    +---------+
```

and then you want to merge those new changes onto the main trunk. If you just use the `cvs update -j R1fix m.c` command again, CVS will attempt to merge again the changes which you have already merged, which can have undesirable side effects.

So instead you need to specify that you only want to merge the changes on the branch which have not yet been merged into the trunk. To do that you specify two `-j' options, and CVS merges the changes from the first revision to the second revision. For example, in this case the simplest way would be

```
cvs update -j 1.2.2.2 -j R1fix m.c    # Merge changes from 1.2.2.2 to the
                                      # head of the R1fix branch
```

The problem with this is that you need to specify the 1.2.2.2 revision manually. A slightly better approach might be to use the date the last merge was done:

```
cvs update -j R1fix:yesterday -j R1fix m.c
```

Better yet, tag the R1fix branch after every merge into the trunk, and then use that tag for subsequent merges:

```
cvs update -j merged_from_R1fix_to_trunk -j R1fix m.c
```

Merging differences between any two revisions

With two `-j revision' flags, the `update` (and `checkout`) command can merge the differences between any two revisions into your working file.

```
$ cvs update -j 1.5 -j 1.3 backend.c
```

will *remove* all changes made between revision 1.3 and 1.5. Note the order of the revisions!

If you try to use this option when operating on multiple files, remember that the numeric revisions will probably be very different between the various files that make up a module. You almost always use symbolic tags rather than revision numbers when operating on multiple files.

Go to the previous, next section.

Go to the previous, next section.

Recursive behavior

Almost all of the subcommands of CVS work recursively when you specify a directory as an argument. For instance, consider this directory structure:

```
$HOME
  |
  +--tc
  |   |
      +--CVS
      |      (internal CVS files)
      +--Makefile
      +--backend.c
      +--driver.c
      +--frontend.c
      +--parser.c
      +--man
      |    |
      |    +--CVS
      |    |  (internal CVS files)
      |    +--tc.1
      |
      +--testing
           |
           +--CVS
           |  (internal CVS files)
           +--testpgm.t
           +--test2.t
```

If `tc' is the current working directory, the following is true:

- `cvs update testing' is equivalent to `cvs update testing/testpgm.t testing/test2.t'
- `cvs update testing man' updates all files in the subdirectories
- `cvs update .' or just `cvs update' updates all files in the `tc` module

If no arguments are given to `update` it will update all files in the current working directory and all its subdirectories. In other words, `.' is a default argument to `update`. This is also true for most of the CVS subcommands, not only the `update` command.

The recursive behavior of the CVS subcommands can be turned off with the `-l' option.

```
$ cvs update -l         # Don't update files in subdirectories
```

TOC

Go to the previous, next section.

Go to the previous, next section.

Adding files to a module

To add a new file to a module, follow these steps.

- You must have a working copy of the module. See section Getting the source.
- Create the new file inside your working copy of the module.
- Use `cvs add filename' to tell CVS that you want to version control the file.
- Use `cvs commit filename' to actually check in the file into the repository. Other developers cannot see the file until you perform this step.
- If the file contains binary data it might be necessary to change the default keyword substitution. See section Keyword substitution. See section admin examples.

You can also use the `add` command to add a new directory inside a module.

Unlike most other commands, the `add` command is not recursive. You cannot even type `cvs add foo/bar'! Instead, you have to

```
$ cd foo
$ cvs add bar
```

See section add--Add a new file/directory to the repository, for a more complete description of the `add` command.

Go to the previous, next section.

Go to the previous, next section.

Removing files from a module

Modules change. New files are added, and old files disappear. Still, you want to be able to retrieve an exact copy of old releases of the module.

Here is what you can do to remove a file from a module, but remain able to retrieve old revisions:

- Make sure that you have not made any uncommitted modifications to the file. See section Viewing differences, for one way to do that. You can also use the `status` or `update` command. If you remove the file without committing your changes, you will of course not be able to retrieve the file as it was immediately before you deleted it.
- Remove the file from your working copy of the module. You can for instance use `rm`.
- Use `cvs remove filename' to tell CVS that you really want to delete the file.
- Use `cvs commit filename' to actually perform the removal of the file from the repository.

When you commit the removal of the file, CVS records the fact that the file no longer exists. It is possible for a file to exist on only some branches and not on others, or to re-add another file with the same name later. CVS will correctly create or not create the file, based on the `-r' and `-D' options specified to `checkout` or `update`.

Command: **cvs remove** *[-lR] files ...*

Schedule file(s) to be removed from the repository (files which have not already been removed from the working directory are not processed). This command does not actually remove the file from the repository until you commit the removal. The `-R' option (the default) specifies that it will recurse into subdirectories; `-l' specifies that it will not.

Here is an example of removing several files:

```
$ cd test
$ rm ?.c
$ cvs remove
cvs remove: Removing .
cvs remove: scheduling a.c for removal
cvs remove: scheduling b.c for removal
cvs remove: use 'cvs commit' to remove these files permanently
$ cvs ci -m "Removed unneeded files"
cvs commit: Examining .
cvs commit: Committing .
```

If you change your mind you can easily resurrect the file before you commit it, using the `add` command.

```
$ ls
```


```
CVS   ja.h  oj.c
$ rm oj.c
$ cvs remove oj.c
cvs remove: scheduling oj.c for removal
cvs remove: use 'cvs commit' to remove this file permanently
$ cvs add oj.c
U oj.c
cvs add: oj.c, version 1.1.1.1, resurrected
```

If you realize your mistake before you run the `remove` command you can use `update` to resurrect the file:

```
$ rm oj.c
$ cvs update oj.c
cvs update: warning: oj.c was lost
U oj.c
```

Go to the previous, next section.

Go to the previous, next section.

Tracking third-party sources

If you modify a program to better fit your site, you probably want to include your modifications when the next release of the program arrives. CVS can help you with this task.

In the terminology used in CVS, the supplier of the program is called a vendor. The unmodified distribution from the vendor is checked in on its own branch, the vendor branch. CVS reserves branch 1.1.1 for this use.

When you modify the source and commit it, your revision will end up on the main trunk. When a new release is made by the vendor, you commit it on the vendor branch and copy the modifications onto the main trunk.

Use the `import` command to create and update the vendor branch. After a successful `import` the vendor branch is made the `head' revision, so anyone that checks out a copy of the file gets that revision. When a local modification is committed it is placed on the main trunk, and made the `head' revision.

Importing a module for the first time

Use the `import` command to check in the sources for the first time. When you use the `import` command to track third-party sources, the vendor tag and release tags are useful. The vendor tag is a symbolic name for the branch (which is always 1.1.1, unless you use the `-b branch' flag---See section import options). The release tags are symbolic names for a particular release, such as `FSF_0_04'.

Suppose you use `wdiff` (a variant of `diff` that ignores changes that only involve whitespace), and are going to make private modifications that you want to be able to use even when new releases are made in the future. You start by importing the source to your repository:

```
$ tar xfz wdiff-0.04.tar.gz
$ cd wdiff-0.04
$ cvs import -m "Import of FSF v. 0.04" fsf/wdiff FSF_DIST WDIFF_0_04
```

The vendor tag is named `FSF_DIST' in the above example, and the only release tag assigned is `WDIFF_0_04'.

Updating a module with the import command

When a new release of the source arrives, you import it into the repository with the same `import` command that you used to set up the repository in the first place. The only difference is that you specify a different release tag this time.

```
$ tar xfz wdiff-0.05.tar.gz
```


```
$ cd wdiff-0.05
$ cvs import -m "Import of FSF v. 0.05" fsf/wdiff FSF_DIST WDIFF_0_
```

For files that have not been modified locally, the newly created revision becomes the head revision. If you have made local changes, `import` will warn you that you must merge the changes into the main trunk, and tell you to use `checkout -j' to do so.

```
$ cvs checkout -jFSF_DIST:yesterday -jFSF_DIST wdiff
```

The above command will check out the latest revision of `wdiff', merging the changes made on the vendor branch `FSF_DIST' since yesterday into the working copy. If any conflicts arise during the merge they should be resolved in the normal way (see section Conflicts example). Then, the modified files may be committed.

Using a date, as suggested above, assumes that you do not import more than one release of a product per day. If you do, you can always use something like this instead:

```
$ cvs checkout -jWDIFF_0_04 -jWDIFF_0_05 wdiff
```

In this case, the two above commands are equivalent.

Go to the previous, next section.

Go to the previous, next section.

Moving and renaming files

Moving files to a different directory or renaming them is not difficult, but some of the ways in which this works may be non-obvious. (Moving or renaming a directory is even harder. See section Moving and renaming directories).

The examples below assume that the file old is renamed to new.

The Normal way to Rename

The normal way to move a file is to copy old to new, and then issue the normal CVS commands to remove old from the repository, and add new to it. (Both old and new could contain relative paths, for example `foo/bar.c').

```
$ mv old new
$ cvs remove old
$ cvs add new
$ cvs commit -m "Renamed old to new" old new
```

This is the simplest way to move a file, it is not error-prone, and it preserves the history of what was done. Note that to access the history of the file you must specify the old or the new name, depending on what portion of the history you are accessing. For example, `cvs log old` will give the log up until the time of the rename.

When new is committed its revision numbers will start at 1.0 again, so if that bothers you, use the `-r rev' option to commit (see section commit options)

Moving the history file

This method is more dangerous, since it involves moving files inside the repository. Read this entire section before trying it out!

```
$ cd $CVSROOT/module
$ mv old,v new,v
```

Advantages:

- The log of changes is maintained intact.
- The revision numbers are not affected.

Disadvantages:

- Old releases of the module cannot easily be fetched from the repository. (The file will show up as

new even in revisions from the time before it was renamed).

- There is no log information of when the file was renamed.
- Nasty things might happen if someone accesses the history file while you are moving it. Make sure no one else runs any of the CVS commands while you move it.

Copying the history file

This way also involves direct modifications to the repository. It is safe, but not without drawbacks.

```
# Copy the RCS file inside the repository
$ cd $CVSROOT/module
$ cp old,v new,v
# Remove the old file
$ cd ~/module
$ rm old
$ cvs remove old
$ cvs commit old
# Remove all tags from new
$ cvs update new
$ cvs log new              # Remember the tag names
$ cvs tag -d tag1
$ cvs tag -d tag2
...
```

By removing the tags you will be able to check out old revisions of the module.

Advantages:

- Checking out old revisions works correctly, as long as you use `-rtag' and not `-Ddate' to retrieve the revisions.
- The log of changes is maintained intact.
- The revision numbers are not affected.

Disadvantages:

- You cannot easily see the history of the file across the rename.
- Unless you use the `-r rev' (see section commit options) flag when new is committed its revision numbers will start at 1.0 again.

Go to the previous, next section.

Go to the previous, next section.

Moving and renaming directories

If you want to be able to retrieve old versions of the module, you must move each file in the directory with the CVS commands. See section The Normal way to Rename. The old, empty directory will remain inside the repository, but it will not appear in your workspace when you check out the module in the future.

If you really want to rename or delete a directory, you can do it like this:

1. Inform everyone who has a copy of the module that the directory will be renamed. They should commit all their changes, and remove their working copies of the module, before you take the steps below.
2. Rename the directory inside the repository.

   ```
   $ cd $CVSROOT/module
   $ mv old-dir new-dir
   ```
3. Fix the CVS administrative files, if necessary (for instance if you renamed an entire module).
4. Tell everyone that they can check out the module and continue working.

If someone had a working copy of the module the CVS commands will cease to work for him, until he removes the directory that disappeared inside the repository.

It is almost always better to move the files in the directory instead of moving the directory. If you move the directory you are unlikely to be able to retrieve old releases correctly, since they probably depend on the name of the directories.

Go to the previous, next section.

Go to the previous, next section.

History browsing

Once you have used CVS to store a version control history--what files have changed when, how, and by whom, there are a variety of mechanisms for looking through the history.

Log messages

Whenever you commit a file you specify a log message.

To look through the log messages which have been specified for every revision which has been committed, use the `cvs log` command (see section log--Print out 'rlog' information for files).

The history database

You can use the history file (see section The history file) to log various CVS actions. To retrieve the information from the history file, use the `cvs history` command (see section history--Show status of files and users).

User-defined logging

You can customize CVS to log various kinds of actions, in whatever manner you choose. These mechanisms operate by executing a script at various times. The script might append a message to a file listing the information and the programmer who created it, or send mail to a group of developers, or, perhaps, post a message to a particular newsgroup. To log commits, use the `` `loginfo' `` file (see section Loginfo). To log commits, checkouts, exports, and tags, respectively, you can also use the `-i', `-o', `-e', and `-t' options in the modules file. For a more flexible way of giving notifications to various users, which requires less in the way of keeping centralized scripts up to date, use the `cvs watch add` command (see section Telling CVS to notify you); this command is useful even if you are not using `cvs watch on`.

The `` `taginfo' `` file defines programs to execute when someone executes a `tag` or `rtag` command. The `` `taginfo' `` file has the standard form for administrative files (see section Reference manual for the Administrative files), where each line is a regular expression followed by a command to execute. The arguments passed to the command are, in order, the tagname, operation (`add` for `tag`, `mov` for `tag -F`, and `del` for `tag -d`), repository, and any remaining are pairs of filename revision. A non-zero exit of the filter program will cause the tag to be aborted.

Annotate command

Command: **cvs annotate** *[- l] files ...*

For each file in files, print the head revision of the trunk, together with information on the last modification for each line. The `-l` option means to process the local directory only, not to recurse (see section Common command options). For example:

```
$ cvs annotate ssfile
Annotations for ssfile
***************
1.1          (mary     27-Mar-96): ssfile line 1
1.2          (joe      28-Mar-96): ssfile line 2
```

The file `` `ssfile' `` currently contains two lines. The `ssfile line 1` line was checked in by `mary` on March 27. Then, on March 28, `joe` added a line `ssfile line 2`, without modifying the `ssfile line 1` line. This report doesn't tell you anything about lines which have been deleted or replaced; you need to use `cvs diff` for that (see section diff--Run diffs between revisions).

Go to the previous, next section.

Go to the previous, next section.

Keyword substitution

As long as you edit source files inside your working copy of a module you can always find out the state of your files via `cvs status' and `cvs log'. But as soon as you export the files from your development environment it becomes harder to identify which revisions they are.

RCS uses a mechanism known as keyword substitution (or keyword expansion) to help identifying the files. Embedded strings of the form `$keyword$` and `$keyword:...$` in a file are replaced with strings of the form `$keyword:value$` whenever you obtain a new revision of the file.

RCS Keywords

This is a list of the keywords that RCS currently (in release 5.6.0.1) supports:

`$Author$`

The login name of the user who checked in the revision.

`$Date$`

The date and time (UTC) the revision was checked in.

`$Header$`

A standard header containing the full pathname of the RCS file, the revision number, the date (UTC), the author, the state, and the locker (if locked). Files will normally never be locked when you use CVS.

`$Id$`

Same as `$Header$`, except that the RCS filename is without a path.

`$Locker$`

The login name of the user who locked the revision (empty if not locked, and thus almost always useless when you are using CVS).

`$Log$`

The log message supplied during commit, preceded by a header containing the RCS filename, the revision number, the author, and the date (UTC). Existing log messages are *not* replaced. Instead, the new log message is inserted after `$Log:...$`. Each new line is prefixed with a comment leader which RCS guesses from the file name extension. It can be changed with `cvs admin -c`. See section admin options. This keyword is useful for accumulating a complete change log in a source file, but for several reasons it can be problematic. See section Problems with the Log keyword..

`$RCSfile$`

The name of the RCS file without a path.

`$Revision$`

The revision number assigned to the revision.

`$Source$`

The full pathname of the RCS file.

`$State$`

The state assigned to the revision. States can be assigned with `cvs admin -s`---See section admin options.

Using keywords

To include a keyword string you simply include the relevant text string, such as `$Id$`, inside the file, and commit the file. CVS will automatically expand the string as part of the commit operation.

It is common to embed Id string in the C source code. This example shows the first few lines of a typical file, after keyword substitution has been performed:

```
static char *rcsid="$Id: samp.c,v 1.5 1993/10/19 14:57:32 ceder Exp $";
/* The following lines will prevent gcc version 2.x
   from issuing an "unused variable" warning. */
#if __GNUC__ == 2
#define USE(var) static void * use_##var = (&use_##var, (void *) &var)
USE (rcsid);
#endif
```

Even though a clever optimizing compiler could remove the unused variable `rcsid`, most compilers tend to include the string in the binary. Some compilers have a `#pragma` directive to include literal text in the binary.

The `ident` command (which is part of the RCS package) can be used to extract keywords and their values from a file. This can be handy for text files, but it is even more useful for extracting keywords from binary files.

```
$ ident samp.c
samp.c:
     $Id: samp.c,v 1.5 1993/10/19 14:57:32 ceder Exp $
$ gcc samp.c
$ ident a.out
a.out:
     $Id: samp.c,v 1.5 1993/10/19 14:57:32 ceder Exp $
```

SCCS is another popular revision control system. It has a command, `what`, which is very similar to `ident` and used for the same purpose. Many sites without RCS have SCCS. Since `what` looks for the character sequence `@(#)` it is easy to include keywords that are detected by either command. Simply prefix the RCS keyword with the magic SCCS phrase, like this:

```
static char *id="@(#) $Id: ab.c,v 1.5 1993/10/19 14:57:32 ceder Exp $";
```

Avoiding substitution

Keyword substitution has its disadvantages. Sometimes you might want the literal text string `$'Author$ to appear inside a file without RCS interpreting it as a keyword and expanding it into something like `$'Author: ceder $.

There is unfortunately no way to selectively turn off keyword substitution. You can use `-ko' (see section Substitution modes) to turn off keyword substitution entirely.

In many cases you can avoid using RCS keywords in the source, even though they appear in the final product. For example, the source for this manual contains `$@asis{}Author$' whenever the text `$'Author$ should appear. In `nroff` and `troff` you can embed the null-character \& inside the keyword for a similar effect.

Substitution modes

Each file has a stored default substitution mode, and each working directory copy of a file also has a substitution mode. The former is set by the `-k' option to `cvs add` and `cvs admin`; the latter is set by the -k or -A options to `cvs checkout` or `cvs update`. `cvs diff` also has a `-k' option. For some examples, See section Handling binary files.

The modes available are:

`-kkv'

Generate keyword strings using the default form, e.g. $Revision: 5.7 $ for the `Revision` keyword.

`-kkvl'

Like `-kkv', except that a locker's name is always inserted if the given revision is currently locked. This option is normally not useful when CVS is used.

`-kk'

Generate only keyword names in keyword strings; omit their values. For example, for the `Revision` keyword, generate the string $Revision$ instead of $Revision: 5.7 $. This option is useful to ignore differences due to keyword substitution when comparing different revisions of a file.

`-ko'

Generate the old keyword string, present in the working file just before it was checked in. For example, for the `Revision` keyword, generate the string $Revision: 1.1 $ instead of $Revision: 5.7 $ if that is how the string appeared when the file was checked in.

`-kb'

Like `-ko', but also inhibit conversion of line endings between the canonical form in which they are stored in the repository (linefeed only), and the form appropriate to the operating system in use on the client. For systems, like unix, which use linefeed only to terminate lines, this is the same as `-ko'. For more information on binary files, see section Handling binary files.

`-kv'

Generate only keyword values for keyword strings. For example, for the `Revision` keyword, generate the string `5.7` instead of $Revision: 5.7 $. This can help generate files in programming languages where it is hard to strip keyword delimiters like $Revision: $ from a string. However, further keyword substitution cannot be performed once the keyword names are removed, so this option should be used with care.

One often would like to use `-kv' with `cvs export`---see section export--Export sources from CVS, similar to checkout. But be aware that doesn't handle an export containing binary files correctly.

Problems with the Log keyword.

The Log keyword is somewhat controversial. As long as you are working on your development system the information is easily accessible even if you do not use the Log keyword--just do a `cvs log`. Once you export the file the history information might be useless anyhow.

A more serious concern is that RCS is not good at handling Log entries when a branch is merged onto the main trunk. Conflicts often result from the merging operation.

People also tend to "fix" the log entries in the file (correcting spelling mistakes and maybe even factual errors). If that is done the information from `cvs log` will not be consistent with the information inside the file. This may or may not be a problem in real life.

It has been suggested that the Log keyword should be inserted *last* in the file, and not in the files header, if it is to be used at all. That way the long list of change messages will not interfere with everyday source file browsing.

Go to the previous, next section.

Go to the previous, next section.

Handling binary files

There are two issues with using CVS to store binary files. The first is that CVS by default convert line endings between the canonical form in which they are stored in the repository (linefeed only), and the form appropriate to the operating system in use on the client (for example, carriage return followed by line feed for Windows NT).

The second is that a binary file might happen to contain data which looks like a keyword (see section Keyword substitution), so keyword expansion must be turned off.

The `-kb' option available with some CVS commands insures that neither line ending conversion nor keyword expansion will be done. If you are using an old version of RCS without this option, and you are using an operating system, such as unix, which terminates lines with linefeeds only, you can use `-ko' instead; if you are on another operating system, upgrade to a version of RCS, such as 5.7 or later, which supports `-kb'.

Here is an example of how you can create a new file using the `-kb' flag:

```
$ echo '$Id$' > kotest
$ cvs add -kb -m"A test file" kotest
$ cvs ci -m"First checkin; contains a keyword" kotest
```

If a file accidentally gets added without `-kb', one can use the `cvs admin` command to recover. For example:

```
$ echo '$Id$' > kotest
$ cvs add -m"A test file" kotest
$ cvs ci -m"First checkin; contains a keyword" kotest
$ cvs admin -kb kotest
$ cvs update -A kotest
```

When you check in the file `kotest' the keywords are expanded. (Try the above example, and do a `cat kotest` after every command). The `cvs admin -kb` command sets the default keyword substitution method for this file, but it does not alter the working copy of the file that you have. The easiest way to get the unexpanded version of `kotest' is `cvs update -A`.

Go to the previous, next section.

Go to the previous, next section.

Revision management

If you have read this far, you probably have a pretty good grasp on what CVS can do for you. This chapter talks a little about things that you still have to decide.

If you are doing development on your own using CVS you could probably skip this chapter. The questions this chapter takes up become more important when more than one person is working in a repository.

When to commit?

Your group should decide which policy to use regarding commits. Several policies are possible, and as your experience with CVS grows you will probably find out what works for you.

If you commit files too quickly you might commit files that do not even compile. If your partner updates his working sources to include your buggy file, he will be unable to compile the code. On the other hand, other persons will not be able to benefit from the improvements you make to the code if you commit very seldom, and conflicts will probably be more common.

It is common to only commit files after making sure that they can be compiled. Some sites require that the files pass a test suite. Policies like this can be enforced using the commitinfo file (see section Commitinfo), but you should think twice before you enforce such a convention. By making the development environment too controlled it might become too regimented and thus counter-productive to the real goal, which is to get software written.

Go to the previous, next section.

Go to the previous, next section.

Reference manual for CVS commands

This appendix describes how to invoke CVS, and describes in detail those subcommands of CVS which are not fully described elsewhere. To look up a particular subcommand, see section Index.

Overall structure of CVS commands

The first release of CVS consisted of a number of shell-scripts. Today CVS is implemented as a single program that is a front-end to RCS and `diff`. The overall format of all CVS commands is:

```
cvs [ cvs_options ] cvs_command [ command_options ] [ command_args ]
```

`cvs`
: The program that is a front-end to RCS.

`cvs_options`
: Some options that affect all sub-commands of CVS. These are described below.

`cvs_command`
: One of several different sub-commands. Some of the commands have aliases that can be used instead; those aliases are noted in the reference manual for that command. There are only two situations where you may omit `cvs_command': `cvs -H' elicits a list of available commands, and `cvs -v' displays version information on CVS itself.

`command_options`
: Options that are specific for the command.

`command_args`
: Arguments to the commands.

There is unfortunately some confusion between `cvs_options` and `command_options`. `-l', when given as a `cvs_option`, only affects some of the commands. When it is given as a `command_option` is has a different meaning, and is accepted by more commands. In other words, do not take the above categorization too seriously. Look at the documentation instead.

Default options and the ~/.cvsrc file

There are some `command_options` that are used so often that you might have set up an alias or some other means to make sure you always specify that option. One example (the one that drove the implementation of the .cvsrc support, actually) is that many people find the default output of the `diff' command to be very hard to read, and that either context diffs or unidiffs are much easier to understand.

The `~/.cvsrc' file is a way that you can add default options to `cvs_commands` within cvs, instead

of relying on aliases or other shell scripts.

The format of the `~/.cvsrc' file is simple. The file is searched for a line that begins with the same name as the `cvs_command' being executed. If a match is found, then the remainder of the line is split up (at whitespace characters) into separate options and added to the command arguments *before* any options from the command line.

If a command has two names (e.g., `checkout' and `co'), the official name, not necessarily the one used on the command line, will be used to match against the file. So if this is the contents of the user's `~/.cvsrc' file:

```
log -N
diff -u
update -P
co -P
```

the command `cvs checkout foo' would have the `-P' option added to the arguments, as well as `cvs co foo'.

With the example file above, the output from `cvs diff foobar' will be in unidiff format. `cvs diff -c foobar' will provide context diffs, as usual. Getting "old" format diffs would be slightly more complicated, because `diff' doesn't have an option to specify use of the "old" format, so you would need `cvs -f diff foobar'.

In place of the command name you can use `cvs' to specify global options (see section Global options). For example the following line in `.cvsrc'

```
cvs -z6
```

causes CVS to use compression level 6

Global options

The available `cvs_options' (that are given to the left of `cvs_command') are:

`-b bindir`

> Use bindir as the directory where RCS programs are located. Overrides the setting of the `$RCSBIN` environment variable and any precompiled directory. This parameter should be specified as an absolute pathname.

`-d cvs_root_directory`

> Use cvs_root_directory as the root directory pathname of the repository. Overrides the setting of the `$CVSROOT` environment variable. See section The Repository.

`-e editor`

> Use editor to enter revision log information. Overrides the setting of the `$CVSEDITOR` and `$EDITOR` environment variables.

`-f`

Do not read the `~/.cvsrc' file. This option is most often used because of the non-orthogonality of the CVS option set. For example, the `cvs log' option `-N' (turn off display of tag names) does not have a corresponding option to turn the display on. So if you have `-N' in the `~/.cvsrc' entry for `diff', you may need to use `-f' to show the tag names. (1)

`-H`

Display usage information about the specified `cvs_command' (but do not actually execute the command). If you don't specify a command name, `cvs -H' displays a summary of all the commands available.

`-l`

Do not log the cvs_command in the command history (but execute it anyway). See section history--Show status of files and users, for information on command history.

`-n`

Do not change any files. Attempt to execute the `cvs_command', but only to issue reports; do not remove, update, or merge any existing files, or create any new files.

`-Q`

Cause the command to be really quiet; the command will only generate output for serious problems.

`-q`

Cause the command to be somewhat quiet; informational messages, such as reports of recursion through subdirectories, are suppressed.

`-r`

Make new working files files read-only. Same effect as if the $CVSREAD environment variable is set (see section All environment variables which affect CVS). The default is to make working files writable, unless watches are on (see section Mechanisms to track who is editing files).

`-s variable=value`

Set a user variable (see section Expansions in administrative files).

`-t`

Trace program execution; display messages showing the steps of CVS activity. Particularly useful with `-n' to explore the potential impact of an unfamiliar command.

`-v`

Display version and copyright information for CVS.

`-w`

Make new working files read-write. Overrides the setting of the $CVSREAD environment variable. Files are created read-write by default, unless $CVSREAD is set or `-r' is given.

`-z gzip-level`

Set the compression level. Only has an effect on the CVS client.

Common command options

This section describes the `command_options' that are available across several CVS commands. These options are always given to the right of `cvs_command'. Not all commands support all of these options; each option is only supported for commands where it makes sense. However, when a command has one of these options you can almost always count on the same behavior of the option as in other commands. (Other command options, which are listed with the individual commands, may have different behavior from one CVS command to the other).

Warning: the `history' command is an exception; it supports many options that conflict even with these standard options.

`-D date_spec`

Use the most recent revision no later than date_spec. date_spec is a single argument, a date description specifying a date in the past.

The specification is sticky when you use it to make a private copy of a source file; that is, when you get a working file using `-D', CVS records the date you specified, so that further updates in the same directory will use the same date (for more information on sticky tags/dates, see section Sticky tags).

A wide variety of date formats are supported by the underlying RCS facilities, similar to those described in co(1), but not exactly the same. The date_spec is interpreted as being in the local timezone, unless a specific timezone is specified. Examples of valid date specifications include:

```
1 month ago
2 hours ago
400000 seconds ago
last year
last Monday
yesterday
a fortnight ago
3/31/92 10:00:07 PST
January 23, 1987 10:05pm
22:00 GMT
```

`-D' is available with the `checkout`, `diff`, `export`, `history`, `rdiff`, `rtag`, and `update` commands. (The `history` command uses this option in a slightly different way; see section history options).

Remember to quote the argument to the `-D' flag so that your shell doesn't interpret spaces as argument separators. A command using the `-D' flag can look like this:

```
$ cvs diff -D "1 hour ago" cvs.texinfo
```

`-f`

When you specify a particular date or tag to CVS commands, they normally ignore files that do

not contain the tag (or did not exist prior to the date) that you specified. Use the `-f' option if you want files retrieved even when there is no match for the tag or date. (The most recent revision of the file will be used).

`-f' is available with these commands: `checkout`, `export`, `rdiff`, `rtag`, and `update`.

Warning: The `commit` command also has a `-f' option, but it has a different behavior for that command. See section commit options.

`-H`

Help; describe the options available for this command. This is the only option supported for all CVS commands.

`-k kflag`

Alter the default RCS processing of keywords. See section Keyword substitution, for the meaning of kflag. Your kflag specification is sticky when you use it to create a private copy of a source file; that is, when you use this option with the `checkout` or `update` commands, CVS associates your selected kflag with the file, and continues to use it with future update commands on the same file until you specify otherwise.

The `-k' option is available with the `add`, `checkout`, `diff` and `update` commands.

`-l`

Local; run only in current working directory, rather than recursing through subdirectories.

Warning: this is not the same as the overall `cvs -l' option, which you can specify to the left of a cvs command!

Available with the following commands: `checkout`, `commit`, `diff`, `export`, `log`, `remove`, `rdiff`, `rtag`, `status`, `tag`, and `update`.

`-m message`

Use message as log information, instead of invoking an editor.

Available with the following commands: `add`, `commit` and `import`.

`-n`

Do not run any checkout/commit/tag program. (A program can be specified to run on each of these activities, in the modules database (see section The modules file); this option bypasses it).

Warning: this is not the same as the overall `cvs -n' option, which you can specify to the left of a cvs command!

Available with the `checkout`, `commit`, `export`, and `rtag` commands.

`-P`

Prune (remove) directories that are empty after being updated, on `checkout`, or `update`. Normally, an empty directory (one that is void of revision-controlled files) is left alone. Specifying `-P' will cause these directories to be silently removed from your checked-out sources. This does not remove the directory from the repository, only from your checked out copy. Note that this option is implied by the `-r' or `-D' options of `checkout` and `export`.

`-p`

Pipe the files retrieved from the repository to standard output, rather than writing them in the current directory. Available with the `checkout` and `update` commands.

`-W`

Specify file names that should be filtered. You can use this option repeatedly. The spec can be a file name pattern of the same type that you can specify in the `.cvswrappers' file. Avaliable with the following commands: `import`, and `update`.

`-r tag`

Use the revision specified by the tag argument instead of the default head revision. As well as arbitrary tags defined with the `tag` or `rtag` command, two special tags are always available: `HEAD' refers to the most recent version available in the repository, and `BASE' refers to the revision you last checked out into the current working directory.

The tag specification is sticky when you use this option with `checkout` or `update` to make your own copy of a file: CVS remembers the tag and continues to use it on future update commands, until you specify otherwise (for more information on sticky tags/dates, see section Sticky tags). The tag can be either a symbolic or numeric tag. See section Tags--Symbolic revisions.

Specifying the `-q' global option along with the `-r' command option is often useful, to suppress the warning messages when the RCS history file does not contain the specified tag.

Warning: this is not the same as the overall `cvs -r' option, which you can specify to the left of a cvs command!

`-r' is available with the `checkout`, `commit`, `diff`, `history`, `export`, `rdiff`, `rtag`, and `update` commands.

add--Add a new file/directory to the repository

- Synopsis: add [-k kflag] [-m 'message'] files...
- Requires: repository, working directory.
- Changes: working directory.
- Synonym: new

Use the `add` command to create a new file or directory in the source repository. The files or directories specified with `add` must already exist in the current directory (which must have been created with the `checkout` command). To add a whole new directory hierarchy to the source repository (for example, files received from a third-party vendor), use the `import` command instead. See section import--Import sources into CVS, using vendor branches.

If the argument to `add` refers to an immediate sub-directory, the directory is created at the correct place in the source repository, and the necessary CVS administration files are created in your working directory. If the directory already exists in the source repository, `add` still creates the administration files

in your version of the directory. This allows you to use `add` to add a particular directory to your priv sources even if someone else created that directory after your checkout of the sources. You can do th following:

```
$ mkdir new_directory
$ cvs add new_directory
$ cvs update new_directory
```

An alternate approach using `update` might be:

```
$ cvs update -d new_directory
```

(To add any available new directories to your working directory, it's probably simpler to use `checkout` (see section checkout--Check out sources for editing) or `update -d' (see section update--Bring work tree in sync with repository)).

The added files are not placed in the source repository until you use `commit` to make the change permanent. Doing an `add` on a file that was removed with the `remove` command will resurrect the file, unless a `commit` command intervened. See section Removing files from a module, for an example.

Unlike most other commands `add` never recurses down directories. It cannot yet handle relative paths. Instead of

```
$ cvs add foo/bar.c
```

you have to do

```
$ cd foo
$ cvs add bar.c
```

add options

There are only two options you can give to `add':

`-k kflag`

> This option specifies the default way that this file will be checked out. The kflag argument (see section Substitution modes) is stored in the RCS file and can be changed with `admin -k` (see section admin options). See section Handling binary files, for information on using this option for binary files.

`-m description`

> Using this option, you can give a description for the file. This description appears in the history log (if it is enabled, see section The history file). It will also be saved in the RCS history file inside the repository when the file is committed. The `log` command displays this description.
>
> The description can be changed using `admin -t'. See section admin--Administration front end for rcs.

If you omit the `-m description' flag, an empty string will be used. You will not be prompted for a description.

add examples

To add the file `backend.c' to the repository, with a description, the following can be used.

```
$ cvs add -m "Optimizer and code generation passes." backend.c
$ cvs commit -m "Early version. Not yet compilable." backend.c
```

admin--Administration front end for rcs

- Requires: repository, working directory.
- Changes: repository.
- Synonym: rcs

This is the CVS interface to assorted administrative RCS facilities, documented in rcs(1). `admin` simply passes all its options and arguments to the `rcs` command; it does no filtering or other processing. This command *does* work recursively, however, so extreme care should be used.

If there is a group whose name matches a compiled in value which defaults to `cvsadmin`, only members of that group can use `cvs admin`. To disallow `cvs admin` for all users, create a group with no users in it.

admin options

Not all valid `rcs` options are useful together with CVS. Some even makes it impossible to use CVS until you undo the effect!

This description of the available options is based on the `rcs(1)' man page, but modified to suit readers that are more interrested in CVS than RCS.

`-Aoldfile`

Might not work together with CVS. Append the access list of oldfile to the access list of the RCS file.

`-alogins`

Might not work together with CVS. Append the login names appearing in the comma-separated list logins to the access list of the RCS file.

`-b[rev]`

When used with bare RCS, this option sets the default branch to rev; in CVS sticky tags (see section Sticky tags) are a better way to decide which branch you want to work on. With CVS, this option can be used to control behavior with respect to the vendor branch.

`-cstring`

Useful with CVS. Sets the comment leader to string. The comment leader is printed before eve log message line generated by the keyword Log (see section Keyword substitution). This is useful for programming languages without multi-line comments. RCS initially guesses the value of the comment leader from the file name extension when the file is first committed.

`-e[logins]`

Might not work together with CVS. Erase the login names appearing in the comma-separated list logins from the access list of the RCS file. If logins is omitted, erase the entire access list.

`-I`

Run interactively, even if the standard input is not a terminal.

`-i`

Useless with CVS. When using bare RCS, this is used to create and initialize a new RCS file, without depositing a revision.

`-ksubst`

Useful with CVS. Set the default keyword substitution to subst. See section Keyword substitution. Giving an explicit `-k' option to `cvs update`, `cvs export`, or `cvs checkout` overrides this default.

`-l[rev]`

Lock the revision with number rev. If a branch is given, lock the latest revision on that branch. If rev is omitted, lock the latest revision on the default branch.

This can be used in conjunction with the `` `rcslock.pl' `` script in the `` `contrib' `` directory of the CVS source distribution to provide reserved checkouts (where only one user can be editing a given file at a time). See the comments in that file for details (and see the `` `README' `` file in that directory for disclaimers about the unsupported nature of contrib). According to comments in that file, locking must set to strict (which is the default).

`-L`

Set locking to strict. Strict locking means that the owner of an RCS file is not exempt from locking for checkin. For use with CVS, strict locking must be set; see the discussion under the `-l' option above.

`-mrev:msg`

Replace the log message of revision rev with msg.

`-Nname[:[rev]]`

Act like `-n', except override any previous assignment of name.

`-nname[:[rev]]`

Associate the symbolic name name with the branch or revision rev. It is normally better to use `cvs tag' or `cvs rtag' instead. Delete the symbolic name if both `:' and rev are omitted; otherwise, print an error message if name is already associated with another number. If rev is symbolic, it is expanded before association. A rev consisting of a branch number followed by a `.' stands for the current latest revision in the branch. A `:' with an empty rev stands for the current latest revision on the default branch, normally the trunk. For example, `rcs -nname: RCS/*' associates name with the current latest revision of all the named RCS files; this contrasts with `rcs -nname:$ RCS/*' which

associates name with the revision numbers extracted from keyword strings in the correspondin working files.

`-orange`

Potentially useful, but dangerous, with CVS (see below). Deletes (outdates) the revisions given by range. A range consisting of a single revision number means that revision. A range consisting of a branch number means the latest revision on that branch. A range of the form `rev1:rev2' means revisions rev1 to rev2 on the same branch, `:rev' means from the beginning of the branch containing rev up to and including rev, and `rev:' means from revision rev to the end of the branch containing rev. None of the outdated revisions may have branches or locks.

Due to the way CVS handles branches rev cannot be specified symbolically if it is a branch. See section Magic branch numbers, for an explanation.

Make sure that no-one has checked out a copy of the revision you outdate. Strange things will happen if he starts to edit it and tries to check it back in. For this reason, this option is not a good way to take back a bogus commit; commit a new revision undoing the bogus change instead (see section Merging differences between any two revisions).

`-q`

Run quietly; do not print diagnostics.

`-sstate[:rev]`

Useful with CVS. Set the state attribute of the revision rev to state. If rev is a branch number, assume the latest revision on that branch. If rev is omitted, assume the latest revision on the default branch. Any identifier is acceptable for state. A useful set of states is `Exp' (for experimental), `Stab' (for stable), and `Rel' (for released). By default, the state of a new revision is set to `Exp' when it is created. The state is visible in the output from cvs log (see section log--Print out 'rlog' information for files), and in the `$'Log$ and `$'State$ keywords (see section Keyword substitution). Note that CVS uses the `dead` state for its own purposes; to take a file to or from the `dead` state use commands like `cvs remove` and `cvs add`, not `cvs admin -s`.

`-t[file]`

Useful with CVS. Write descriptive text from the contents of the named file into the RCS file, deleting the existing text. The file pathname may not begin with `-'. If file is omitted, obtain the text from standard input, terminated by end-of-file or by a line containing `.' by itself. Prompt for the text if interaction is possible; see `-I'. The descriptive text can be seen in the output from `cvs log' (see section log--Print out 'rlog' information for files).

`-t-string`

Similar to `-tfile'. Write descriptive text from the string into the RCS file, deleting the existing text.

`-U`

Set locking to non-strict. Non-strict locking means that the owner of a file need not lock a revision for checkin. For use with CVS, strict locking must be set; see the discussion under the `-l' option above.

`-u[rev]`

See the option `-l' above, for a discussion of using this option with CVS. Unlock the revision with

number rev. If a branch is given, unlock the latest revision on that branch. If rev is omitted, remove the latest lock held by the caller. Normally, only the locker of a revision may unlock it. Somebody else unlocking a revision breaks the lock. This causes a mail message to be sent to the original locker. The message contains a commentary solicited from the breaker. The commentary is terminated by end-of-file or by a line containing `.` by itself.

`-Vn`

Emulate RCS version n. Use -Vn to make an RCS file acceptable to RCS version n by discarding information that would confuse version n.

`-xsuffixes`

Useless with CVS. Use suffixes to characterize RCS files.

admin examples

Outdating is dangerous

First, an example of how *not* to use the `admin` command. It is included to stress the fact that this command can be quite dangerous unless you know *exactly* what you are doing.

The `-o' option can be used to outdate old revisions from the history file. If you are short on disc this option might help you. But think twice before using it--there is no way short of restoring the latest backup to undo this command!

The next line is an example of a command that you would *not* like to execute.

```
$ cvs admin -o:R_1_02 .
```

The above command will delete all revisions up to, and including, the revision that corresponds to the tag R_1_02. But beware! If there are files that have not changed between R_1_02 and R_1_03 the file will have *the same* numerical revision number assigned to the tags R_1_02 and R_1_03. So not only will it be impossible to retrieve R_1_02; R_1_03 will also have to be restored from the tapes!

Comment leaders

If you use the Log keyword and you do not agree with the guess for comment leader that CVS has done, you can enforce your will with `cvs admin -c`. This might be suitable for `nroff` source:

```
$ cvs admin -c'.\" ' *.man
$ rm *.man
$ cvs update
```

The two last steps are to make sure that you get the versions with correct comment leaders in your working files.

checkout--Check out sources for editing

- Synopsis: checkout [options] modules...
- Requires: repository.
- Changes: working directory.
- Synonyms: co, get

Make a working directory containing copies of the source files specified by modules. You must execute `checkout` before using most of the other CVS commands, since most of them operate on your working directory.

The modules part of the command are either symbolic names for some collection of source directories and files, or paths to directories or files in the repository. The symbolic names are defined in the `modules' file. See section The modules file.

Depending on the modules you specify, `checkout` may recursively create directories and populate them with the appropriate source files. You can then edit these source files at any time (regardless of whether other software developers are editing their own copies of the sources); update them to include new changes applied by others to the source repository; or commit your work as a permanent change to the source repository.

Note that `checkout` is used to create directories. The top-level directory created is always added to the directory where `checkout` is invoked, and usually has the same name as the specified module. In the case of a module alias, the created sub-directory may have a different name, but you can be sure that it will be a sub-directory, and that `checkout` will show the relative path leading to each file as it is extracted into your private work area (unless you specify the `-Q' global option).

The files created by `checkout` are created read-write, unless the `-r' option to CVS (see section Global options) is specified, the `CVSREAD` environment variable is specified (see section All environment variables which affect CVS), or a watch is in effect for that file (see section Mechanisms to track who is editing files).

Running `checkout` on a directory that was already built by a prior `checkout` is also permitted, and has the same effect as specifying the `-d' option to the `update` command, that is, any new directories that have been created in the repository will appear in your work area. See section update--Bring work tree in sync with repository.

checkout options

These standard options are supported by `checkout` (see section Common command options, for a complete description of them):

`-D date`

Use the most recent revision no later than date. This option is sticky, and implies `-P'. See section Sticky tags, for more information on sticky tags/dates.

`-f`

Only useful with the `-D date' or `-r tag' flags. If no matching revision is found, retrieve the most recent revision (instead of ignoring the file).

`-k kflag`

Process RCS keywords according to kflag. See co(1). This option is sticky; future updates of this file in this working directory will use the same kflag. The `status` command can be viewed to see the sticky options. See section status--Status info on the revisions.

`-l`

Local; run only in current working directory.

`-n`

Do not run any checkout program (as specified with the `-o' option in the modules file; see section The modules file).

`-P`

Prune empty directories.

`-p`

Pipe files to the standard output.

`-r tag`

Use revision tag. This option is sticky, and implies `-P'. See section Sticky tags, for more information on sticky tags/dates.

In addition to those, you can use these special command options with `checkout`:

`-A`

Reset any sticky tags, dates, or `-k' options. See section Sticky tags, for more information on sticky tags/dates.

`-c`

Copy the module file, sorted, to the standard output, instead of creating or modifying any files or directories in your working directory.

`-d dir`

Create a directory called dir for the working files, instead of using the module name. Unless you also use `-N', the paths created under dir will be as short as possible.

`-j tag`

With two `-j' options, merge changes from the revision specified with the first `-j' option to the revision specified with the second `j' option, into the working directory.

With one `-j' option, merge changes from the ancestor revision to the revision specified with the `-j' option, into the working directory. The ancestor revision is the common ancestor of the revision which the working directory is based on, and the revision specified in the `-j' option.

In addition, each -j option can contain an optional date specification which, when used with branches, can limit the chosen revision to one within a specific date. An optional date is specified by adding a colon (:) to the tag: `-jSymbolic_Tag:Date_Specifier'.

See section Merging.

`-N`

Only useful together with `-d dir'. With this option, CVS will not shorten module paths in your working directory. (Normally, CVS shortens paths as much as possible when you specify an explicit target directory).

`-s`

Like `-c', but include the status of all modules, and sort it by the status string. See section The modules file, for info about the `-s' option that is used inside the modules file to set the module status.

checkout examples

Get a copy of the module `tc':

```
$ cvs checkout tc
```

Get a copy of the module `tc' as it looked one day ago:

```
$ cvs checkout -D yesterday tc
```

commit--Check files into the repository

- Version 1.3 Synopsis: commit [-lnR] [-m 'log_message' | -f file] [-r revision] [files...]
- Version 1.3.1 Synopsis: commit [-lnRf] [-m 'log_message' | -F file] [-r revision] [files...]
- Requires: working directory, repository.
- Changes: repository.
- Synonym: ci

Warning: The `-f file' option will probably be renamed to `-F file', and `-f' will be given a new behavior in future releases of CVS.

Use `commit` when you want to incorporate changes from your working source files into the source repository.

If you don't specify particular files to commit, all of the files in your working current directory are examined. `commit` is careful to change in the repository only those files that you have really changed. By default (or if you explicitly specify the `-R' option), files in subdirectories are also examined and committed if they have changed; you can use the `-l' option to limit `commit` to the current directory only.

`commit` verifies that the selected files are up to date with the current revisions in the source repository; it will notify you, and exit without committing, if any of the specified files must be made current first with `update` (see section update--Bring work tree in sync with repository). `commit` does not call the

`update` command for you, but rather leaves that for you to do when the time is right.

When all is well, an editor is invoked to allow you to enter a log message that will be written to one or more logging programs (see section The modules file, and see section Loginfo) and placed in the RCS history file inside the repository. This log message can be retrieved with the `log` command; See section log--Print out 'rlog' information for files. You can specify the log message on the command line with the `-m message' option, and thus avoid the editor invocation, or use the `-f file' option to specify that the argument file contains the log message.

commit options

These standard options are supported by `commit` (see section Common command options, for a complete description of them):

`-l`

Local; run only in current working directory.

`-n`

Do not run any module program.

`-R`

Commit directories recursively. This is on by default.

`-r revision`

Commit to revision. revision must be either a branch, or a revision on the main trunk that is higher than any existing revision number. You cannot commit to a specific revision on a branch.

`commit` also supports these options:

`-F file`

This option is present in CVS releases 1.3-s3 and later. Read the log message from file, instead of invoking an editor.

`-f`

This option is present in CVS 1.3-s3 and later releases of CVS. Note that this is not the standard behavior of the `-f' option as defined in See section Common command options.

Force CVS to commit a new revision even if you haven't made any changes to the file. If the current revision of file is 1.7, then the following two commands are equivalent:

```
$ cvs commit -f file
$ cvs commit -r 1.8 file
```

`-f file`

This option is present in CVS releases 1.3, 1.3-s1 and 1.3-s2. Note that this is not the standard behavior of the `-f' option as defined in See section Common command options.

Read the log message from file, instead of invoking an editor.

`-m message`

Use message as the log message, instead of invoking an editor.

commit examples

New major release number

When you make a major release of your product, you might want the revision numbers to track your major release number. You should normally not care about the revision numbers, but this is a thing that many people want to do, and it can be done without doing any harm.

To bring all your files up to the RCS revision 3.0 (including those that haven't changed), you might do:

```
$ cvs commit -r 3.0
```

Note that it is generally a bad idea to try to make the RCS revision number equal to the current release number of your product. You should think of the revision number as an internal number that the CVS package maintains, and that you generally never need to care much about. Using the `tag` and `rtag` commands you can give symbolic names to the releases instead. See section tag--Add a symbolic tag to checked out version of RCS file and See section rtag--Add a tag to the RCS file.

Note that the number you specify with `-r' must be larger than any existing revision number. That is, if revision 3.0 exists, you cannot `cvs commit -r 1.3'.

Committing to a branch

You can commit to a branch revision (one that has an even number of dots) with the `-r' option. To create a branch revision, use the `-b' option of the `rtag` or `tag` commands (see section tag--Add a symbolic tag to checked out version of RCS file or see section rtag--Add a tag to the RCS file). Then, either `checkout` or `update` can be used to base your sources on the newly created branch. From that point on, all `commit` changes made within these working sources will be automatically added to a branch revision, thereby not disturbing main-line development in any way. For example, if you had to create a patch to the 1.2 version of the product, even though the 2.0 version is already under development, you might do:

```
$ cvs rtag -b -r FCS1_2 FCS1_2_Patch product_module
$ cvs checkout -r FCS1_2_Patch product_module
$ cd product_module
[[ hack away ]]
$ cvs commit
```

This works automatically since the `-r' option is sticky.

Creating the branch after editing

Say you have been working on some extremely experimental software, based on whatever revision you happened to checkout last week. If others in your group would like to work on this software with you,

but without disturbing main-line development, you could commit your change to a new branch. Others can then checkout your experimental stuff and utilize the full benefit of CVS conflict resolution. The scenario might look like:

```
[[ hacked sources are present ]]
$ cvs tag -b EXPR1
$ cvs update -r EXPR1
$ cvs commit
```

The `update` command will make the `-r EXPR1' option sticky on all files. Note that your changes to the files will never be removed by the `update` command. The `commit` will automatically commit to the correct branch, because the `-r' is sticky. You could also do like this:

```
[[ hacked sources are present ]]
$ cvs tag -b EXPR1
$ cvs commit -r EXPR1
```

but then, only those files that were changed by you will have the `-r EXPR1' sticky flag. If you hack away, and commit without specifying the `-r EXPR1' flag, some files may accidentally end up on the main trunk.

To work with you on the experimental change, others would simply do

```
$ cvs checkout -r EXPR1 whatever_module
```

diff--Run diffs between revisions

- Synopsis: diff [-l] [rcsdiff_options] [[-r rev1 | -D date1] [-r rev2 | -D date2]] [files...]
- Requires: working directory, repository.
- Changes: nothing.

The `diff` command is used to compare different revisions of files. The default action is to compare your working files with the revisions they were based on, and report any differences that are found.

If any file names are given, only those files are compared. If any directories are given, all files under them will be compared.

The exit status will be 0 if no differences were found, 1 if some differences were found, and 2 if any error occurred.

diff options

These standard options are supported by `diff` (see section Common command options, for a complete description of them):

`-D date`

Use the most recent revision no later than date. See `-r' for how this affects the comparison.

CVS can be configured to pass the `-D' option through to `rcsdiff` (which in turn passes it onto `diff`. GNU diff uses `-D' as a way to put `cpp`-style `#define' statements around the output differences. There is no way short of testing to figure out how CVS was configured. In the default configuration CVS will use the `-D date' option.

`-k kflag`

Process RCS keywords according to kflag. See co(1).

`-l`

Local; run only in current working directory.

`-R`

Examine directories recursively. This option is on by default.

`-r tag`

Compare with revision tag. Zero, one or two `-r' options can be present. With no `-r' option, the working file will be compared with the revision it was based on. With one `-r', that revision will be compared to your current working file. With two `-r' options those two revisions will be compared (and your working file will not affect the outcome in any way).

One or both `-r' options can be replaced by a `-D date' option, described above.

Any other options that are found are passed through to `rcsdiff`, which in turn passes them to `diff`. The exact meaning of the options depends on which `diff` you are using. The long options introduced in GNU diff 2.0 are not yet supported in CVS. See the documentation for your `diff` to see which options are supported.

diff examples

The following line produces a Unidiff (`-u' flag) between revision 1.14 and 1.19 of `` `backend.c' ``. Due to the `-kk' flag no keywords are substituted, so differences that only depend on keyword substitution are ignored.

```
$ cvs diff -kk -u -r 1.14 -r 1.19 backend.c
```

Suppose the experimental branch EXPR1 was based on a set of files tagged RELEASE_1_0. To see what has happened on that branch, the following can be used:

```
$ cvs diff -r RELEASE_1_0 -r EXPR1
```

A command like this can be used to produce a context diff between two releases:

```
$ cvs diff -c -r RELEASE_1_0 -r RELEASE_1_1 > diffs
```

If you are maintaining ChangeLogs, a command like the following just before you commit your changes may help you write the ChangeLog entry. All local modifications that have not yet been committed will be printed.


```
$ cvs diff -u | less
```

export--Export sources from CVS, similar to checkout

- Synopsis: export [-flNn] [-r rev|-D date] [-k subst] [-d dir] module...
- Requires: repository.
- Changes: current directory.

This command is a variant of `checkout`; use it when you want a copy of the source for module without the CVS administrative directories. For example, you might use `export` to prepare source for shipment off-site. This command requires that you specify a date or tag (with `-D' or `-r'), so that you can count on reproducing the source you ship to others.

One often would like to use `-kv' with `cvs export`. This causes any RCS keywords to be expanded such that an import done at some other site will not lose the keyword revision information. But be aware that doesn't handle an export containing binary files correctly. Also be aware that after having used `-kv', one can no longer use the `ident` command (which is part of the RCS suite--see ident(1)) which looks for RCS keyword strings. If you want to be able to use `ident` you must not use `-kv'.

export options

These standard options are supported by `export` (see section Common command options, for a complete description of them):

`-D date`
: Use the most recent revision no later than date.

`-f`
: If no matching revision is found, retrieve the most recent revision (instead of ignoring the file).

`-l`
: Local; run only in current working directory.

`-n`
: Do not run any checkout program.

`-R`
: Export directories recursively. This is on by default.

`-r tag`
: Use revision tag.

In addition, these options (that are common to `checkout` and `export`) are also supported:

`-d dir`
: Create a directory called dir for the working files, instead of using the module name. Unless you

also use `-N', the paths created under dir will be as short as possible.

`-k subst`

Set keyword expansion mode (see section Substitution modes).

`-N`

Only useful together with `-d dir'. With this option, CVS will not shorten module paths in your working directory. (Normally, CVS shortens paths as much as possible when you specify an explicit target directory.)

history--Show status of files and users

- Synopsis: history [-report] [-flags] [-options args] [files...]
- Requires: the file `$CVSROOT/CVSROOT/history'
- Changes: nothing.

CVS can keep a history file that tracks each use of the `checkout`, `commit`, `rtag`, `update`, and `release` commands. You can use `history` to display this information in various formats.

Logging must be enabled by creating the file `$CVSROOT/CVSROOT/history'.

Warning: `history` uses `-f', `-l', `-n', and `-p' in ways that conflict with the normal use inside CVS (see section Common command options).

history options

Several options (shown above as `-report') control what kind of report is generated:

`-c`

Report on each time commit was used (i.e., each time the repository was modified).

`-e`

Everything (all record types); equivalent to specifying `-xMACFROGWUT'.

`-m module`

Report on a particular module. (You can meaningfully use `-m' more than once on the command line.)

`-o`

Report on checked-out modules.

`-T`

Report on all tags.

`-x type`

Extract a particular set of record types type from the CVS history. The types are indicated by single letters, which you may specify in combination.

Certain commands have a single record type:

`F`
: release

`O`
: checkout

`T`
: rtag

One of four record types may result from an update:

`C`
: A merge was necessary but collisions were detected (requiring manual merging).

`G`
: A merge was necessary and it succeeded.

`U`
: A working file was copied from the repository.

`W`
: The working copy of a file was deleted during update (because it was gone from the repository).

One of three record types results from commit:

`A`
: A file was added for the first time.

`M`
: A file was modified.

`R`
: A file was removed.

The options shown as `-flags' constrain or expand the report without requiring option arguments:

`-a`
: Show data for all users (the default is to show data only for the user executing `history`).

`-l`
: Show last modification only.

`-w`
: Show only the records for modifications done from the same working directory where `history` is executing.

The options shown as `-options args' constrain the report based on an argument:

`-b str`
: Show data back to a record containing the string str in either the module name, the file name, or the repository path.

`-D date`

Show data since date. This is slightly different from the normal use of `-D date', which selects newest revision older than date.

`-p repository`

Show data for a particular source repository (you can specify several `-p' options on the same command line).

`-r rev`

Show records referring to revisions since the revision or tag named rev appears in individual RCS files. Each RCS file is searched for the revision or tag.

`-t tag`

Show records since tag tag was last added to the the history file. This differs from the `-r' flag above in that it reads only the history file, not the RCS files, and is much faster.

`-u name`

Show records for user name.

import--Import sources into CVS, using vendor branches

- Synopsis: import [-options] repository vendortag releasetag...
- Requires: Repository, source distribution directory.
- Changes: repository.

Use `import` to incorporate an entire source distribution from an outside source (e.g., a source vendor) into your source repository directory. You can use this command both for initial creation of a repository, and for wholesale updates to the module from the outside source. See section Tracking third-party sources, for a discussion on this subject.

The repository argument gives a directory name (or a path to a directory) under the CVS root directory for repositories; if the directory did not exist, import creates it.

When you use import for updates to source that has been modified in your source repository (since a prior import), it will notify you of any files that conflict in the two branches of development; use `checkout -j' to reconcile the differences, as import instructs you to do.

If CVS decides a file should be ignored (see section Ignoring files via cvsignore), it does not import it and prints `I ' followed by the filename

If the file `$CVSROOT/CVSROOT/cvswrappers' exists, any file whose names match the specifications in that file will be treated as packages and the appropriate filtering will be performed on the file/directory before being imported, See section The cvswrappers file.

The outside source is saved in a first-level RCS branch, by default 1.1.1. Updates are leaves of this branch; for example, files from the first imported collection of source will be revision 1.1.1.1, then files from the first imported update will be revision 1.1.1.2, and so on.

At least three arguments are required. repository is needed to identify the collection of source. vendo is a tag for the entire branch (e.g., for 1.1.1). You must also specify at least one releasetag to identify files at the leaves created each time you execute `import`.

import options

This standard option is supported by `import` (see section Common command options, for a complete description):

`-m message`

Use message as log information, instead of invoking an editor.

There are three additional special options.

`-b branch`

Specify a first-level branch other than 1.1.1. Unless the `-b branch' flag is given, revisions will *always* be made to the branch 1.1.1--even if a vendortag that matches another branch is given! What happens in that case, is that the tag will be reset to 1.1.1. Warning: This behavior might change in the future.

`-k subst`

Indicate the RCS keyword expansion mode desired. This setting will apply to all files created during the import, but not to any files that previously existed in the repository. See section Substitution modes for a list of valid `-k' settings.

`-I name`

Specify file names that should be ignored during import. You can use this option repeatedly. To avoid ignoring any files at all (even those ignored by default), specify `-I !'.

name can be a file name pattern of the same type that you can specify in the `.cvsignore' file. See section Ignoring files via cvsignore.

`-W spec`

Specify file names that should be filtered during import. You can use this option repeatedly.

spec can be a file name pattern of the same type that you can specify in the `.cvswrappers' file. See section The cvswrappers file.

import examples

See section Tracking third-party sources, and See section Creating a module from a number of files.

log--Print out 'rlog' information for files

- Synopsis: log [-l] rlog-options [files...]
- Requires: repository, working directory.
- Changes: nothing.

- Synonym: rlog

Display log information for files. `log` calls the RCS utility `rlog`, which prints all available information about the RCS history file. This includes the location of the RCS file, the head revision (the latest revision on the trunk), all symbolic names (tags) and some other things. For each revision, the revision number, the author, the number of lines added/deleted and the log message are printed. All times are displayed in Coordinated Universal Time (UTC). (Other parts of CVS print times in the local timezone).

log options

Only one option is interpreted by CVS and not passed on to `rlog`:

`-l`

Local; run only in current working directory. (Default is to run recursively).

By default, `rlog` prints all information that is available. All other options (including those that normally behave differently) are passed through to `rlog` and restrict the output. See rlog(1) for a complete description of options. This incomplete list (which is a slightly edited extract from rlog(1)) lists all options that are useful in conjunction with CVS.

Please note: There can be no space between the option and its argument, since `rlog` parses its options in a different way than CVS.

`-b`

Print information about the revisions on the default branch, normally the highest branch on the trunk.

`-ddates`

Print information about revisions with a checkin date/time in the range given by the semicolon-separated list of dates. The following table explains the available range formats:

`d1<d2`

`d2>d1`

Select the revisions that were deposited between d1 and d2 inclusive.

`<d`

`d>`

Select all revisions dated d or earlier.

`d<`

`>d`

Select all revisions dated d or later.

`d`

Select the single, latest revision dated d or earlier.

The date/time strings d, d1, and d2 are in the free format explained in co(1). Quoting is normally necessary, especially for < and >. Note that the separator is a semicolon (;).

- -h Print only the RCS pathname, working pathname, head, default branch, access list, locks, symbolic

names, and suffix.

- -N Do not print the list of tags for this file. This option can be very useful when your site uses a lot of tags, so rather than "more"'ing over 3 pages of tag information, the log information is presented without tags at all.
- -R Print only the name of the RCS history file.
- -rrevisions Print information about revisions given in the comma-separated list revisions of revisions and ranges. The following table explains the available range formats:

`rev1:rev2`

Revisions rev1 to rev2 (which must be on the same branch).

`:rev`

Revisions from the beginning of the branch up to and including rev.

`rev:`

Revisions starting with rev to the end of the branch containing rev.

`branch`

An argument that is a branch means all revisions on that branch. You can unfortunately not specify a symbolic branch here. You must specify the numeric branch number. See section Magic branch numbers, for an explanation.

`branch1:branch2`

A range of branches means all revisions on the branches in that range.

`branch.`

The latest revision in branch.

A bare `-r' with no revisions means the latest revision on the default branch, normally the trunk.

- -sstates Print information about revisions whose state attributes match one of the states given in the comma-separated list states.
- -t Print the same as `-h', plus the descriptive text.
- -wlogins Print information about revisions checked in by users with login names appearing in the comma-separated list logins. If logins is omitted, the user's login is assumed.

`rlog` prints the intersection of the revisions selected with the options `-d', `-l', `-s', and `-w', intersected with the union of the revisions selected by `-b' and `-r'.

log examples

Contributed examples are gratefully accepted.

rdiff---'patch' format diffs between releases

- rdiff [-flags] [-V vn] [-r t|-D d [-r t2|-D d2]] modules...
- Requires: repository.

- Changes: nothing.
- Synonym: patch

Builds a Larry Wall format patch(1) file between two releases, that can be fed directly into the patch program to bring an old release up-to-date with the new release. (This is one of the few CVS commands that operates directly from the repository, and doesn't require a prior checkout.) The diff output is sent to the standard output device.

You can specify (using the standard `-r' and `-D' options) any combination of one or two revisions or dates. If only one revision or date is specified, the patch file reflects differences between that revision or date and the current head revisions in the RCS file.

Note that if the software release affected is contained in more than one directory, then it may be necessary to specify the `-p' option to the patch command when patching the old sources, so that patch is able to find the files that are located in other directories.

rdiff options

These standard options are supported by `rdiff` (see section Common command options, for a complete description of them):

`-D date`
: Use the most recent revision no later than date.

`-f`
: If no matching revision is found, retrieve the most recent revision (instead of ignoring the file).

`-l`
: Local; don't descend subdirectories.

`-r tag`
: Use revision tag.

In addition to the above, these options are available:

`-c`
: Use the context diff format. This is the default format.

`-s`
: Create a summary change report instead of a patch. The summary includes information about files that were changed or added between the releases. It is sent to the standard output device. This is useful for finding out, for example, which files have changed between two dates or revisions.

`-t`
: A diff of the top two revisions is sent to the standard output device. This is most useful for seeing what the last change to a file was.

`-u`
: Use the unidiff format for the context diffs. This option is not available if your diff does not support the unidiff format. Remember that old versions of the `patch` program can't handle the

unidiff format, so if you plan to post this patch to the net you should probably not use `-u'.

`-V vn`

Expand RCS keywords according to the rules current in RCS version vn (the expansion format changed with RCS version 5).

rdiff examples

Suppose you receive mail from `foo@bar.com` asking for an update from release 1.2 to 1.4 of the tc compiler. You have no such patches on hand, but with CVS that can easily be fixed with a command such as this:

```
$ cvs rdiff -c -r FOO1_2 -r FOO1_4 tc | \
$$ Mail -s 'The patches you asked for' foo@bar.com
```

Suppose you have made release 1.3, and forked a branch called `R_1_3fix' for bugfixes. `R_1_3_1' corresponds to release 1.3.1, which was made some time ago. Now, you want to see how much development has been done on the branch. This command can be used:

```
$ cvs patch -s -r R_1_3_1 -r R_1_3fix module-name
cvs rdiff: Diffing module-name
File ChangeLog,v changed from revision 1.52.2.5 to 1.52.2.6
File foo.c,v changed from revision 1.52.2.3 to 1.52.2.4
File bar.h,v changed from revision 1.29.2.1 to 1.2
```

release--Indicate that a Module is no longer in use

- release [-d] directories...
- Requires: Working directory.
- Changes: Working directory, history log.

This command is meant to safely cancel the effect of `cvs checkout'. Since CVS doesn't lock files, it isn't strictly necessary to use this command. You can always simply delete your working directory, if you like; but you risk losing changes you may have forgotten, and you leave no trace in the CVS history file (see section The history file) that you've abandoned your checkout.

Use `cvs release' to avoid these problems. This command checks that no uncommitted changes are present; that you are executing it from immediately above a CVS working directory; and that the repository recorded for your files is the same as the repository defined in the module database.

If all these conditions are true, `cvs release' leaves a record of its execution (attesting to your intentionally abandoning your checkout) in the CVS history log.

release options

The `release` command supports one command option:

`-d`

Delete your working copy of the file if the release succeeds. If this flag is not given your files will remain in your working directory.

Warning: The `release` command uses `rm -r `module'' to delete your file. This has the very serious side-effect that any directory that you have created inside your checked-out sources, and not added to the repository (using the `add` command; see section add--Add a new file/directory to the repository) will be silently deleted--even if it is non-empty!

release output

Before `release` releases your sources it will print a one-line message for any file that is not up-to-date.

Warning: Any new directories that you have created, but not added to the CVS directory hierarchy with the `add` command (see section add--Add a new file/directory to the repository) will be silently ignored (and deleted, if `-d' is specified), even if they contain files.

`U file`

There exists a newer revision of this file in the repository, and you have not modified your local copy of the file.

`A file`

The file has been added to your private copy of the sources, but has not yet been committed to the repository. If you delete your copy of the sources this file will be lost.

`R file`

The file has been removed from your private copy of the sources, but has not yet been removed from the repository, since you have not yet committed the removal. See section commit--Check files into the repository.

`M file`

The file is modified in your working directory. There might also be a newer revision inside the repository.

`? file`

file is in your working directory, but does not correspond to anything in the source repository, and is not in the list of files for CVS to ignore (see the description of the `-I' option, and see section Ignoring files via cvsignore). If you remove your working sources, this file will be lost.

Note that no warning message like this is printed for spurious directories that CVS encounters. The directory, and all its contents, are silently ignored.

release examples

Release the module, and delete your local working copy of the files.

```
$ cd ..          # You must stand immediately above the
                 # sources when you issue `cvs release'.
$ cvs release -d tc
You have [0] altered files in this repository.
Are you sure you want to release (and delete) module `tc': y
$
```

rtag--Add a tag to the RCS file

- rtag [-falnR] [-b] [-d] [-r tag | -Ddate] symbolic_tag modules...
- Requires: repository.
- Changes: repository.
- Synonym: rfreeze

You can use this command to assign symbolic tags to particular, explicitly specified source revisions in the repository. `rtag` works directly on the repository contents (and requires no prior checkout). Use `tag` instead (see section tag--Add a symbolic tag to checked out version of RCS file), to base the selection of revisions on the contents of your working directory.

If you attempt to use a tag name that already exists, CVS will complain and not overwrite that tag. Use the `-F' option to force the new tag value.

rtag options

These standard options are supported by `rtag` (see section Common command options, for a complete description of them):

`-D date`

Tag the most recent revision no later than date.

`-f`

Only useful with the `-D date' or `-r tag' flags. If no matching revision is found, use the most recent revision (instead of ignoring the file).

`-F`

Overwrite an existing tag of the same name on a different revision. This option is new in CVS 1.4. The old behavior is matched by `cvs tag -F'.

`-l`

Local; run only in current working directory.

`-n`

Do not run any tag program that was specified with the `-t' flag inside the `modules' file. (s section The modules file).

`-R`

Commit directories recursively. This is on by default.

`-r tag`

Only tag those files that contain tag. This can be used to rename a tag: tag only the files identified by the old tag, then delete the old tag, leaving the new tag on exactly the same files as the old tag.

In addition to the above common options, these options are available:

`-a`

Use the `-a' option to have `rtag` look in the `Attic' (see section Removing files from a module) for removed files that contain the specified tag. The tag is removed from these files, which makes it convenient to re-use a symbolic tag as development continues (and files get removed from the up-coming distribution).

`-b`

Make the tag a branch tag. See section Branches.

`-d`

Delete the tag instead of creating it.

In general, tags (often the symbolic names of software distributions) should not be removed, but the `-d' option is available as a means to remove completely obsolete symbolic names if necessary (as might be the case for an Alpha release, or if you mistagged a module).

status--Status info on the revisions

- status [-lR] [-v] [files...]
- Requires: working directory, repository.
- Changes: nothing.

Display a brief report on the current status of files with respect to the source repository, including any sticky tags, dates, or `-k' options.

You can also use this command to determine the potential impact of a `cvs update' on your working source directory--but remember that things might change in the repository before you run `update`.

status options

These standard options are supported by `status` (see section Common command options, for a complete description of them):

`-l`

Local; run only in current working directory.

`-R`

Commit directories recursively. This is on by default.

There is one additional option:

`-v`

Verbose. In addition to the information normally displayed, print all symbolic tags, together with the numerical value of the revision or branch they refer to.

tag--Add a symbolic tag to checked out version of RCS file

- tag [-lR] [-b] [-d] symbolic_tag [files...]
- Requires: working directory, repository.
- Changes: repository.
- Synonym: freeze

Use this command to assign symbolic tags to the nearest repository versions to your working sources. The tags are applied immediately to the repository, as with `rtag`, but the versions are supplied implicitly by the CVS records of your working files' history rather than applied explicitly.

One use for tags is to record a snapshot of the current sources when the software freeze date of a project arrives. As bugs are fixed after the freeze date, only those changed sources that are to be part of the release need be re-tagged.

The symbolic tags are meant to permanently record which revisions of which files were used in creating a software distribution. The `checkout` and `update` commands allow you to extract an exact copy of a tagged release at any time in the future, regardless of whether files have been changed, added, or removed since the release was tagged.

This command can also be used to delete a symbolic tag, or to create a branch. See the options section below.

If you attempt to use a tag name that already exists, CVS will complain and not overwrite that tag. Use the `-F' option to force the new tag value.

tag options

These standard options are supported by `tag` (see section Common command options, for a complete description of them):

`-F`

Overwrite an existing tag of the same name on a different revision. This option is new in CVS 1.4. The old behavior is matched by `cvs tag -F'.

`-l`

Local; run only in current working directory.

`-R`

Commit directories recursively. This is on by default.

Two special options are available:

`-b`

The -b option makes the tag a branch tag (see section Branches), allowing concurrent, isolated development. This is most useful for creating a patch to a previously released software distribution.

`-d`

Delete a tag.

If you use `cvs tag -d symbolic_tag', the symbolic tag you specify is deleted instead of being added. Warning: Be very certain of your ground before you delete a tag; doing this permanently discards some historical information, which may later turn out to be valuable.

update--Bring work tree in sync with repository

- update [-AdflPpR] [-d] [-r tag|-D date] files...
- Requires: repository, working directory.
- Changes: working directory.

After you've run checkout to create your private copy of source from the common repository, other developers will continue changing the central source. From time to time, when it is convenient in your development process, you can use the `update` command from within your working directory to reconcile your work with any revisions applied to the source repository since your last checkout or update.

update options

These standard options are available with `update` (see section Common command options, for a complete description of them):

`-D date`

Use the most recent revision no later than date. This option is sticky, and implies `-P'. See section Sticky tags, for more information on sticky tags/dates.

`-f`

Only useful with the `-D date' or `-r tag' flags. If no matching revision is found, retrieve the most recent revision (instead of ignoring the file).

`-k kflag`

Process RCS keywords according to kflag. See co(1). This option is sticky; future updates of this file in this working directory will use the same kflag. The `status` command can be viewed to see the sticky options. See section status--Status info on the revisions.

`-l`

Local; run only in current working directory. See section Recursive behavior.

`-P`

Prune empty directories.

`-p`

Pipe files to the standard output.

`-R`

Operate recursively. This is on by default. See section Recursive behavior.

`-r tag`

Retrieve revision tag. This option is sticky, and implies `-P'. See section Sticky tags, for more information on sticky tags/dates.

These special options are also available with `update`.

`-A`

Reset any sticky tags, dates, or `-k' options. See section Sticky tags, for more information on sticky tags/dates.

`-d`

Create any directories that exist in the repository if they're missing from the working directory. Normally, `update` acts only on directories and files that were already enrolled in your working directory.

This is useful for updating directories that were created in the repository since the initial checkout; but it has an unfortunate side effect. If you deliberately avoided certain directories in the repository when you created your working directory (either through use of a module name or by listing explicitly the files and directories you wanted on the command line), then updating with `-d' will create those directories, which may not be what you want.

`-I name`

Ignore files whose names match name (in your working directory) during the update. You can specify `-I' more than once on the command line to specify several files to ignore. Use `-I !' to avoid ignoring any files at all. See section Ignoring files via cvsignore, for other ways to make CVS ignore some files.

`-Wspec`

Specify file names that should be filtered during update. You can use this option repeatedly.

spec can be a file name pattern of the same type that you can specify in the `.cvswrappers' file. See section The cvswrappers file.

`-jrevision`

With two `-j' options, merge changes from the revision specified with the first `-j' option to the revision specified with the second `j' option, into the working directory.

With one `-j' option, merge changes from the ancestor revision to the revision specified with the `-j' option, into the working directory. The ancestor revision is the common ancestor of the revision which the working directory is based on, and the revision specified in the `-j' option.

In addition, each -j option can contain an optional date specification which, when used with branches, can limit the chosen revision to one within a specific date. An optional date is specified by adding a colon (:) to the tag: `-jSymbolic_Tag:Date_Specifier'.

See section Merging.

update output

`update` keeps you informed of its progress by printing a line for each file, preceded by one character indicating the status of the file:

`U file`

The file was brought up to date with respect to the repository. This is done for any file that exists in the repository but not in your source, and for files that you haven't changed but are not the most recent versions available in the repository.

`A file`

The file has been added to your private copy of the sources, and will be added to the source repository when you run `commit` on the file. This is a reminder to you that the file needs to be committed.

`R file`

The file has been removed from your private copy of the sources, and will be removed from the source repository when you run `commit` on the file. This is a reminder to you that the file needs to be committed.

`M file`

The file is modified in your working directory.

`M' can indicate one of two states for a file you're working on: either there were no modifications to the same file in the repository, so that your file remains as you last saw it; or there were modifications in the repository as well as in your copy, but they were merged successfully, without conflict, in your working directory.

CVS will print some messages if it merges your work, and a backup copy of your working file (as it looked before you ran `update`) will be made. The exact name of that file is printed while `update` runs.

`C file`

A conflict was detected while trying to merge your changes to file with changes from the source repository. file (the copy in your working directory) is now the output of the rcsmerge(1) command on the two revisions; an unmodified copy of your file is also in your working directory, with the name `.#file.revision' where revision is the RCS revision that your modified file started from. (Note that some systems automatically purge files that begin with `.#' if they have not been accessed for a few days. If you intend to keep a copy of your original file, it is a very good idea to rename it.)

`? file`

file is in your working directory, but does not correspond to anything in the source repository, and

is not in the list of files for CVS to ignore (see the description of the `-I' option, and see section Ignoring files via cvsignore).

Note that no warning message like this is printed for spurious directories that CVS encounters. The directory, and all its contents, are silently ignored.

update examples

The following line will display all files which are not up-to-date without actually change anything in your working directory. It can be used to check what has been going on with the project.

```
$ cvs -n -q update
```

Go to the previous, next section.

Go to the previous, next section.

Reference manual for the Administrative files

Inside the repository, in the directory `$CVSROOT/CVSROOT', there are a number of supportive files for CVS. You can use CVS in a limited fashion without any of them, but if they are set up properly they can help make life easier.

The most important of these files is the `modules' file, which defines the modules inside the repository.

The modules file

The `modules' file records your definitions of names for collections of source code. CVS will use these definitions if you use CVS to update the modules file (use normal commands like `add`, `commit`, etc).

The `modules' file may contain blank lines and comments (lines beginning with `#') as well as module definitions. Long lines can be continued on the next line by specifying a backslash (`\') as the last character on the line.

A module definition is a single line of the `modules' file, in either of two formats. In both cases, mname represents the symbolic module name, and the remainder of the line is its definition.

`mname -a aliases...`

This represents the simplest way of defining a module mname. The `-a' flags the definition as a simple alias: CVS will treat any use of mname (as a command argument) as if the list of names aliases had been specified instead. aliases may contain either other module names or paths. When you use paths in aliases, `checkout` creates all intermediate directories in the working directory, just as if the path had been specified explicitly in the CVS arguments.

`mname [ options ] dir [ files... ] [ &module... ]`

In the simplest case, this form of module definition reduces to `mname dir'. This defines all the files in directory dir as module mname. dir is a relative path (from `$CVSROOT`) to a directory of source in the source repository. In this case, on checkout, a single directory called mname is created as a working directory; no intermediate directory levels are used by default, even if dir was a path involving several directory levels.

By explicitly specifying files in the module definition after dir, you can select particular files from directory dir. The sample definition for `modules' is an example of a module defined with a single file from a particular directory. Here is another example:

```
m4test  unsupported/gnu/m4 foreach.m4 forloop.m4
```

With this definition, executing `cvs checkout m4test' will create a single working directory `m4test' containing the two files listed, which both come from a common directory several levels deep in the CVS source repository.

A module definition can refer to other modules by including `&module' in its definition. `checkou` creates a subdirectory for each such module, in your working directory.

`-d name`

Name the working directory something other than the module name.

`-e prog`

Specify a program prog to run whenever files in a module are exported. prog runs with a single argument, the module name.

`-i prog`

Specify a program prog to run whenever files in a module are committed. prog runs with a single argument, the full pathname of the affected directory in a source repository. The `` `commitinfo' ``, `` `loginfo' ``, and `` `editinfo' `` files provide other ways to call a program on commit.

`-o prog`

Specify a program prog to run whenever files in a module are checked out. prog runs with a single argument, the module name.

`-s status`

Assign a status to the module. When the module file is printed with `cvs checkout -s' the modules are sorted according to primarily module status, and secondarily according to the module name. This option has no other meaning. You can use this option for several things besides status: for instance, list the person that is responsible for this module.

`-t prog`

Specify a program prog to run whenever files in a module are tagged with `rtag`. prog runs with two arguments: the module name and the symbolic tag specified to `rtag`. There is no way to specify a program to run when `tag` is executed.

`-u prog`

Specify a program prog to run whenever `cvs update' is executed from the top-level directory of the checked-out module. prog runs with a single argument, the full path to the source repository for this module.

The cvswrappers file

Wrappers allow you to set a hook which transforms files on their way in and out of CVS. Most or all of the wrappers features do not work with client/server CVS.

The file `` `cvswrappers' `` defines the script that will be run on a file when its name matches a regular expresion. There are two scripts that can be run on a file or directory. One script is executed on the file/directory before being checked into the repository (this is denoted with the `-t` flag) and the other when the file is checked out of the repository (this is denoted with the `-f` flag)

The `` `cvswrappers' `` also has a `-m' option to specify the merge methodology that should be used when the file is updated. `MERGE` means the usual CVS behavior: try to merge the files (this generally will not work for binary files). `COPY` means that `cvs update` will merely copy one version over the other, and require the user using mechanisms outside CVS, to insert any necessary changes. The `-m' wrapper option only affects behavior when merging is done on update; it does not affect how files are stored. See See section Handling

binary files, for more on binary files.

The basic format of the file `cvswrappers' is:

```
wildcard     [option value][option value]...

where option is one of
-f           from cvs filter         value: path tofilter
-t           to cvs filter           value: path to filter
-m           update methodology      value: MERGE or COPY

and value is a single-quote delimited value.

*.nib     -f 'unwrap %s' -t 'wrap %s %s' -m 'COPY'
*.c       -t 'indent %s %s'
```

The above example of a `cvswrappers' file states that all files/directories that end with a `.nib' should be filtered with the `wrap' program before checking the file into the repository. The file should be filtered though the `unwrap' program when the file is checked out of the repository. The `cvswrappers' file also states that a `COPY' methodology should be used when updating the files in the repository (that is no merging should be performed).

The last example line says that all files that end with a `*.c' should be filtered with `indent' before being checked into the repository. Unlike the previous example no filtering of the `*.c' file is done when it is checked out of the repository. The `-t' filter is called with two arguments, the first is the name of the file/directory to filter and the second is the pathname to where the resulting filtered file should be placed.

The `-f' filter is called with one argument, which is the name of the file to filter from. The end result of this filter will be a file in the users directory that they can work on as they normally would.

The commit support files

The `-i' flag in the `modules' file can be used to run a certain program whenever files are committed (see section The modules file). The files described in this section provide other, more flexible, ways to run programs whenever something is committed.

There are three kind of programs that can be run on commit. They are specified in files in the repository, as described below. The following table summarizes the file names and the purpose of the corresponding programs.

`commitinfo'
: The program is responsible for checking that the commit is allowed. If it exits with a non-zero exit status the commit will be aborted.

`editinfo'
: The specified program is used to edit the log message, and possibly verify that it contains all required fields. This is most useful in combination with the `rcsinfo' file, which can hold a log message template (see section Rcsinfo).

`loginfo'

The specified program is called when the commit is complete. It receives the log message and some additional information and can store the log message in a file, or mail it to appropriate persons, or maybe post it to a local newsgroup, or... Your imagination is the limit!

The common syntax

The four files `commitinfo', `loginfo', `rcsinfo' and `editinfo' all have a common format. The purpose of the files are described later on. The common syntax is described here.

Each line contains the following:

- A regular expression
- A whitespace separator--one or more spaces and/or tabs.
- A file name or command-line template.

Blank lines are ignored. Lines that start with the character `#' are treated as comments. Long lines unfortunately can *not* be broken in two parts in any way.

The first regular expression that matches the current directory name in the repository is used. The rest of the line is used as a file name or command-line as appropriate.

Commitinfo

The `commitinfo' file defines programs to execute whenever `cvs commit' is about to execute. These programs are used for pre-commit checking to verify that the modified, added and removed files are really ready to be committed. This could be used, for instance, to verify that the changed files conform to to your site's standards for coding practice.

As mentioned earlier, each line in the `commitinfo' file consists of a regular expression and a command-line template. The template can include a program name and any number of arguments you wish to supply to it. The full path to the current source repository is appended to the template, followed by the file names of any files involved in the commit (added, removed, and modified files).

The first line with a regular expression matching the relative path to the module will be used. If the command returns a non-zero exit status the commit will be aborted.

If the repository name does not match any of the regular expressions in this file, the `DEFAULT' line is used, if it is specified.

All occurances of the name `ALL' appearing as a regular expression are used in addition to the first matching regular expression or the name `DEFAULT'.

Note: when CVS is accessing a remote repository, `commitinfo' will be run on the *remote* (i.e., server) side, not the client side (see section Remote repositories).

TOC

Editinfo

If you want to make sure that all log messages look the same way, you can use the `editinfo' file to specify a program that is used to edit the log message. This program could be a custom-made editor that always enforces a certain style of the log message, or maybe a simple shell script that calls an editor, and checks that the entered message contains the required fields.

If no matching line is found in the `editinfo' file, the editor specified in the environment variable $CVSEDITOR is used instead. If that variable is not set, then the environment variable $EDITOR is used instead. If that variable is not set a precompiled default, normally vi, will be used.

The `editinfo' file is often most useful together with the `rcsinfo' file, which can be used to specify a log message template.

Each line in the `editinfo' file consists of a regular expression and a command-line template. The template must include a program name, and can include any number of arguments. The full path to the current log message template file is appended to the template.

One thing that should be noted is that the `ALL' keyword is not supported. If more than one matching line is found, the first one is used. This can be useful for specifying a default edit script in a module, and then overriding it in a subdirectory.

If the repository name does not match any of the regular expressions in this file, the `DEFAULT' line is used, if it is specified.

If the edit script exits with a non-zero exit status, the commit is aborted.

Note: when CVS is accessing a remote repository, `editinfo' will be run on the *remote* (i.e., server) side, not the client side (see section Remote repositories).

Editinfo example

The following is a little silly example of a `editinfo' file, together with the corresponding `rcsinfo' file, the log message template and an editor script. We begin with the log message template. We want to always record a bug-id number on the first line of the log message. The rest of log message is free text. The following template is found in the file `/usr/cvssupport/tc.template'.

```
BugId:
```

The script `/usr/cvssupport/bugid.edit' is used to edit the log message.

```
#!/bin/sh
#
#       bugid.edit filename
#
#  Call $EDITOR on FILENAME, and verify that the
#  resulting file contains a valid bugid on the first
#  line.
if [ "x$EDITOR" = "x" ]; then EDITOR=vi; fi
```


```
if [ "x$CVSEDITOR" = "x" ]; then CVSEDITOR=$EDITOR; fi
$CVSEDITOR $1
until head -1|grep '^BugId:[ ]*[0-9][0-9]*$' < $1
do  echo -n  "No BugId found.  Edit again? ([y]/n)"
    read ans
    case ${ans} in
        n*) exit 1;;
    esac
    $CVSEDITOR $1
done
```

The `editinfo' file contains this line:

```
^tc     /usr/cvssupport/bugid.edit
```

The `rcsinfo' file contains this line:

```
^tc     /usr/cvssupport/tc.template
```

Loginfo

The `loginfo' file is used to control where `cvs commit' log information is sent. The first entry on a line is a regular expression which is tested against the directory that the change is being made to, relative to the $CVSROOT. If a match is found, then the remainder of the line is a filter program that should expect log information on its standard input.

The filter program may use one and only one % modifier (a la printf). If `%s' is specified in the filter program, a brief title is included (enclosed in single quotes) showing the modified file names.

If the repository name does not match any of the regular expressions in this file, the `DEFAULT' line is used, if it is specified.

All occurances of the name `ALL' appearing as a regular expression are used in addition to the first matching regular expression or `DEFAULT'.

The first matching regular expression is used.

See section The commit support files, for a description of the syntax of the `loginfo' file.

Note: when CVS is accessing a remote repository, `loginfo' will be run on the *remote* (i.e., server) side, not the client side (see section Remote repositories).

Loginfo example

The following `loginfo' file, together with the tiny shell-script below, appends all log messages to the file `$CVSROOT/CVSROOT/commitlog', and any commits to the administrative files (inside the `CVSROOT' directory) are also logged in `/usr/adm/cvsroot-log'.

```
ALL             /usr/local/bin/cvs-log $CVSROOT/CVSROOT/commitlog
```


```
^CVSROOT        /usr/local/bin/cvs-log /usr/adm/cvsroot-log
```

The shell-script `/usr/local/bin/cvs-log' looks like this:

```
#!/bin/sh
(echo "------------------------------------------------------------";
 echo -n $USER"  ";
 date;
 echo;
 sed '1s+'${CVSROOT}'++') >> $1
```

Rcsinfo

The `rcsinfo' file can be used to specify a form to edit when filling out the commit log. The `rcsinfo' file has a syntax similar to the `editinfo', `commitinfo' and `loginfo' files. See section The common syntax. Unlike the other files the second part is *not* a command-line template. Instead, the part after the regular expression should be a full pathname to a file containing the log message template.

If the repository name does not match any of the regular expressions in this file, the `DEFAULT' line is used, if it is specified.

All occurances of the name `ALL' appearing as a regular expression are used in addition to the first matching regular expression or `DEFAULT'.

The log message template will be used as a default log message. If you specify a log message with `cvs commit -m message' or `cvs commit -f file' that log message will override the template.

See section Editinfo example, for an example `rcsinfo' file.

When CVS is accessing a remote repository, the contents of `rcsinfo' at the time a directory is first checked out will specify a template which does not then change. If you edit `rcsinfo' or its templates, you may need to check out a new working directory.

Ignoring files via cvsignore

There are certain file names that frequently occur inside your working copy, but that you don't want to put under CVS control. Examples are all the object files that you get while you compile your sources. Normally, when you run `cvs update', it prints a line for each file it encounters that it doesn't know about (see section update output).

CVS has a list of files (or sh(1) file name patterns) that it should ignore while running `update', `import' and `release'. This list is constructed in the following way.

- The list is initialized to include certain file name patterns: names associated with CVS administration, or with other common source control systems; common names for patch files, object files, archive files, and editor backup files; and other names that are usually artifacts of assorted utilities. Currently, the default list of ignored file name patterns is:

```
RCS     SCCS    CVS     CVS.adm
```


```
RCSLOG  cvslog.*
tags    TAGS
.make.state     .nse_depinfo
*~      #*      .#*     ,*      _$*     *$
*.old   *.bak   *.BAK   *.orig  *.rej   .del-*
*.a     *.olb   *.o     *.obj   *.so    *.exe
*.Z     *.elc   *.ln
core
```

- The per-repository list in `$CVSROOT/CVSROOT/cvsignore' is appended to the list, if that file exists.
- The per-user list in `.cvsignore' in your home directory is appended to the list, if it exists.
- Any entries in the environment variable $CVSIGNORE is appended to the list.
- Any `-I' options given to CVS is appended.
- As CVS traverses through your directories, the contents of any `.cvsignore' will be appended to the list. The patterns found in `.cvsignore' are only valid for the directory that contains them, not for any sub-directories.

In any of the 5 places listed above, a single exclamation mark (`!') clears the ignore list. This can be used if you want to store any file which normally is ignored by CVS.

The history file

The file `$CVSROOT/CVSROOT/history' is used to log information for the `history' command (see section history--Show status of files and users). This file must be created to turn on logging. This is done automatically if the `cvs init' command is used to set up the repository (see section Setting up the repository).

The file format of the `history' file is documented only in comments in the CVS source code, but generally programs should use the `cvs history' command to access it anyway, in case the format changes with future releases of CVS.

Setting up the repository

To set up a CVS repository, choose a directory with ample disk space available for the revision history of the source files. It should be accessable (directly or via a networked file system) from all machines which want to use CVS in server or local mode; the client machines need not have any access to it other than via the CVS protocol.

To create a repository, run the `cvs init' command. It will set up an empty repository in the CVS root specified in the usual way (see section The Repository). For example,

```
cvs -d /usr/local/cvsroot init
```

`cvs init' is careful to never overwrite any existing files in the repository, so no harm is done if you run `cvs init' on an already set-up repository.

`cvs init` will enable history logging; if you don't want that, remove the history file after running `cvs init`. See section The history file.

Expansions in administrative files

Sometimes in writing an administrative file, you might want the file to be able to know various things based on environment CVS is running in. There are several mechanisms to do that.

To find the home directory of the user running CVS (from the `HOME` environment variable), use `~' followed by `/' or the end of the line. Likewise for the home directory of user, use `~user'. These variables are expanded on the server machine, and don't get any resonable expansion if pserver (see section Direct connection with password authentication) is in used; therefore user variables (see below) may be a better choice to customize behavior based on the user running CVS.

One may want to know about various pieces of information internal to CVS. A CVS internal variable has the syntax `${variable}`, where variable starts with a letter and consists of alphanumberic characters and `_'. If the character following variable is a non-alphanumeric character other than `_', the `{' and `}' can be omitted. The CVS internal variables are:

`CVSROOT`

This is the value of the CVS root in use. See section The Repository, for a description of the various ways to specify this.

`RCSBIN`

This is the value CVS is using for where to find RCS binaries. See section Global options, for a description of how to specify this.

`CVSEDITOR`

`VISUAL`

`EDITOR`

These all expand to the same value, which is the editor that CVS is using. See section Global options, for how to specify this.

`USER`

Username of the user running CVS (on the CVS server machine).

If you want to pass a value to the administrative files which the user that is running CVS can specify, use a user variable. To expand a user variable, the administrative file contains `${=variable}`. To set a user variable, specify the global option `-s' to CVS, with argument `variable=value`. It may be particularly useful to specify this option via `.cvsrc' (see section Default options and the ~/.cvsrc file).

For example, if you want the administrative file to refer to a test directory you might create a user variable `TESTDIR`. Then if CVS is invoked as `cvs -s TESTDIR=/work/local/tests`, and the administrative file contains `sh ${=TESTDIR}/runtests`, then that string is expanded to `sh /work/local/tests/runtests`.

All other strings containing `$' are reserved; there is no way to quote a `$' character so that `$' represents itself.

Go to the previous, next section.

Go to the previous, next section.

All environment variables which affect CVS

This is a complete list of all environment variables that affect CVS.

`$CVSIGNORE`

A whitespace-separated list of file name patterns that CVS should ignore. See section Ignoring files via cvsignore.

`$CVSWRAPPERS`

A whitespace-separated list of file name patterns that CVS should treat as wrappers. See section The cvswrappers file.

`$CVSREAD`

If this is set, `checkout` and `update` will try hard to make the files in your working directory read-only. When this is not set, the default behavior is to permit modification of your working files.

`$CVSROOT`

Should contain the full pathname to the root of the CVS source repository (where the RCS history files are kept). This information must be available to CVS for most commands to execute; if `$CVSROOT` is not set, or if you wish to override it for one invocation, you can supply it on the command line: `cvs -d cvsroot cvs_command...' Once you have checked out a working directory, CVS stores the appropriate root (in the file `` `CVS/Root' ``), so normally you only need to worry about this when initially checking out a working directory.

`$EDITOR`

`$CVSEDITOR`

Specifies the program to use for recording log messages during commit. If not set, the default is `/usr/ucb/vi'. `$CVSEDITOR` overrides `$EDITOR`. `$CVSEDITOR` does not exist in CVS 1.3, but the next release will probably include it.

`$PATH`

If `$RCSBIN` is not set, and no path is compiled into CVS, it will use `$PATH` to try to find all programs it uses.

`$RCSBIN`

Specifies the full pathname of the location of RCS programs, such as co(1) and ci(1). If not set, a compiled-in value is used, or your `$PATH` is searched.

`$HOME`

`$HOMEPATH`

Used to locate the directory where the `` `.cvsrc' `` file is searched (`$HOMEPATH` is used for Windows-NT). see section Default options and the ~/.cvsrc file

`$CVS_RSH`

Used in client-server mode when accessing a remote repository using RSH. The default value is `rsh`. You can set it to use another program for accssing the remote server (e.g. for HP-UX 9, you should set it to `remsh` because `rsh` invokes the restricted shell). see section Connecting with rsh

`$CVS_SERVER`

Used in client-server mode when accessing a remote repository using RSH. It specifies the name of the program to start on the server side when accessing a remote repository using RSH. The default value is `cvs`. see section Connecting with rsh

`$CVS_PASSFILE`

Used in client-server mode when accessing the `cvs login server`. Default value is `` `$HOME/.cvspass' ``. see section Using the client with password authentication

`$CVS_PASSWORD`

Used in client-server mode when accessing the `cvs login server`. see section Using the client with password authentication

`$CVS_CLIENT_PORT`

Used in client-server mode when accessing the server via Kerberos. see section Direct connection with kerberos

`$CVS_RCMD_PORT`

Used in client-server mode. If set, specifies the port number to be used when accessing the RCMD demon on the server side. (Currently not used for Unix clients).

`$CVS_CLIENT_LOG`

Used for debugging only in client-server mode. If set, everything send to the server is logged into `` `$CVS_CLIENT_LOG.in' `` and everything send from the server is logged into `` `$CVS_CLIENT_LOG.out' ``.

`$CVS_SERVER_SLEEP`

Used only for debugging the server side in client-server mode. If set, delays the start of the server child process the the specified amount of seconds so that you can attach to it with a debugger.

`$CVS_IGNORE_REMOTE_ROOT`

(What is the purpose of this variable?)

`$COMSPEC`

Used under OS/2 only. It specifies the name of the command interpreter and defaults to CMD.EXE.

CVS is a front-end to RCS. The following environment variables affect RCS. Note that if you are using the client/server CVS, these variables need to be set on the server side (which may or not may be possible depending on how you are connecting). There is probably not any need to set any of them, however.

`$LOGNAME`

`$USER`

If set, they affect who RCS thinks you are. If you have trouble checking in files it might be because your login name differs from the setting of e.g. `$LOGNAME`.

`$RCSINIT`

Options prepended to the argument list, separated by spaces. A backslash escapes spaces within an option. The `$RCSINIT` options are prepended to the argument lists of most RCS commands.

`$TMPDIR`

`$TMP`

`$TEMP`

Name of the temporary directory. The environment variables are inspected in the order they appear above and the first value found is taken; if none of them are set, a host-dependent default is used, typically `` `/tmp' ``.

Go to the previous, next section.

Go to the previous, next section.

Troubleshooting

Magic branch numbers

Externally, branch numbers consist of an odd number of dot-separated decimal integers. See section Revision numbers. That is not the whole truth, however. For efficiency reasons CVS sometimes inserts an extra 0 in the second rightmost position (1.2.3 becomes 1.2.0.3, 8.9.10.11.12 becomes 8.9.10.11.0.12 and so on).

CVS does a pretty good job at hiding these so called magic branches, but in at least four places the hiding is incomplete.

- The magic branch can appear in the output from `cvs status` in vanilla CVS 1.3. This is fixed in CVS 1.3-s2.
- The magic branch number appears in the output from `cvs log`. This is much harder to fix, since `cvs log` runs `rlog` (which is part of the RCS distribution), and modifying `rlog` to know about magic branches would probably break someone's habits (if they use branch 0 for their own purposes).
- You cannot specify a symbolic branch name to `cvs log`.
- You cannot specify a symbolic branch name to `cvs admin`.

You can use the `admin` command to reassign a symbolic name to a branch the way RCS expects it to be. If `R4patches` is assigned to the branch 1.4.2 (magic branch number 1.4.0.2) in file `` `numbers.c' `` you can do this:

```
$ cvs admin -NR4patches:1.4.2 numbers.c
```

It only works if at least one revision is already committed on the branch. Be very careful so that you do not assign the tag to the wrong number. (There is no way to see how the tag was assigned yesterday).

Go to the previous, next section.

TOC

Go to the previous, next section.

GNU GENERAL PUBLIC LICENSE

Go to the previous, next section.

Go to the previous section.

Index

TOC

a

b

C

d

e

f

g

h

i

j

k

l

m

n

o

p

r

S

t

u

v

w

Go to the previous section.

CVS--Concurrent Versions System

TOC

(1)

Yes, this really should be fixed, and it's being worked on

Dvips: A DVI-to-PostScript Translator

for version 5.526b

February 1994

Tomas Rokicki (edited for Dvipsk by Karl Berry)

Dvips: A DVI-to-PostScript Translator

@defcodeindex fl @defcodeindex op

This document is based on `dvips.tex' by Tomas Rokicki. It is in the public domain.

Why Use dvips?

The `dvips' program has a number of features that set it apart from other PostScript drivers for TeX. This rather long section describes the advantages of using dvips, and may be skipped if you are just interested in learning how to use the program. See section Installation, for details of compilation and installation.

The dvips driver generates excellent, standard PostScript, that can be included in other documents as figures or printed through a variety of spoolers. The generated PostScript requires very little printer memory, so very complex documents with a lot of fonts can easily be printed even on PostScript printers without much memory, such as the original Apple LaserWriter. The PostScript output is also compact, requiring less disk space to store and making it feasible as a transfer format.

Even those documents that are too complex to print in their entirety on a particular printer can be printed, since dvips will automatically split such documents into pieces, reclaiming the printer memory between each piece.

The dvips program supports graphics in a natural way, allowing PostScript graphics to be included and automatically scaled and positioned in a variety of ways.

Printers with resolutions other than 300 dpi are also supported, even if they have different resolutions in the horizontal and vertical directions. High resolution output is supported for typesetters, including an option that compresses the bitmapped fonts so that typesetter virtual memory is not exhausted. This option also significantly reduces the size of the PostScript file and decoding in the printer is very fast.

Missing fonts can be automatically generated if METAFONT exists on the system, or fonts can be converted from `gf' to `pk' format on demand. If a font cannot be generated, a scaled version of the same font at a different size can be used instead, although dvips will complain loudly about the poor aesthetics of the resulting output.

Users will appreciate features such as collated copies and support for `tpic', `psfig', `emtex', and `METAPOST'; system administrators will love the support for multiple printers, each with their own configuration file, and the ability to pipe the output directly to a program such as `lpr'. Support for MS-DOS, OS/2, and VMS in addition to UNIX is provided in the standard distribution, and porting to other systems is easy.

One of the most important features is the support of virtual fonts, which add an entirely new level of flexibility to TeX. Virtual fonts are used to give dvips its excellent PostScript font support, handling all the font remapping in a natural, portable, elegant, and extensible way. The dvips `afm2tfm' driver even comes with its own `afm2tfm' program that creates the necessary virtual fonts and TeX font metric files automatically from the Adobe font metric files.

Source is provided and freely distributable, so adding a site-specific feature is possible. Adding such features is made easier by the highly modular structure of the program.

There is really no reason to use another driver, and the more people use dvips, the less time will be spent fighting with PostScript and the more time will be available to create beautiful documents. So if you don't use dvips on your system, get it today.

Using dvips

To use dvips, simply type

```
dvips foo
```

where `foo.dvi' is the output of TeX that you want to print. If dvips has been installed correctly, the document should come out of your default printer.

If you use fonts that have not been used on your system before, they may be automatically generated; this process can take a few minutes. The next time that document is printed, these fonts will already exist, so printing will go much faster.

Many options are available; they are described in a later section. For a brief summary of available options, just type

```
dvips
```

Paper Size and Landscape Mode

Most TeX documents at a particular site are designed to use the standard paper size (for example, letter size in the United States or A4 in Europe.) The dvips program defaults to these paper sizes and can be customized for the defaults at each site or on each printer.

But many documents are designed for other paper sizes. For instance, you may want to design a document that has the long edge of the paper horizontal. This can be useful when typesetting booklets, brochures, complex tables, or many other documents. This type of paper orientation is called landscape orientation (the `normal' orientation is portrait). Alternatively, a document might be designed for ledger or A3 paper.

Since the intended paper size is a document design decision, and not a decision that is made at printing time, such information should be given in the TeX file and not on the dvips command line. For this reason, dvips supports a `papersize' special. It is hoped that this special will become standard over time for TeX previewers and other printer drivers.

The format of the `papersize' special is

```
\special{papersize=8.5in,11in}
```

where the dimensions given above are for a standard letter sheet. The first dimension given is the horizontal

size of the page, and the second is the vertical size. The dimensions supported are the same as for TeX; namely, in (inches), cm (centimeters), mm (millimeters), pt (points), sp (scaled points), bp (big points, same as the default PostScript unit), pc (picas), dd (didot points), and cc (ciceros).

For a landscape document, the `papersize' comment would be given as

```
\special{papersize=11in,8.5in}
```

An alternate specification of `landscape' is to have a special of the form

```
\special{landscape}
```

This is supported for backward compatibility, but it is hoped that eventually the `papersize' comment will dominate.

Of course, using such a command only informs dvips of the desired paper size; you must still adjust the `hsize' and `vsize' in your TeX document to actually use the full page.

The `papersize' special must occur somewhere on the first page of the document.

Including PostScript Graphics

Scaling and including PostScript graphics is a breeze--if the PostScript file is correctly formed. Even if it is not, however, the file can usually be accommodated with just a little more work. The most important feature of a good PostScript file--from the standpoint of including it in another document--is an accurate bounding box comment.

The Bounding Box Comment

Every well-formed PostScript file has a comment describing where on the page the graphic is located, and how big that graphic is. This information is given in terms of the lower left and upper right corners of a box just enclosing the graphic, and is thus referred to as a bounding box. These coordinates are given in PostScript units (there are precisely 72 PostScript units to the inch) with respect to the lower left corner of the sheet of paper.

To see if a PostScript file has a bounding box comment, just look at the first few lines of the file. (PostScript is standard ASCII, so you can use any text editor to do this.) If within the first few dozen lines there is a line of the form

```
%%BoundingBox: 0 1 2 3
```

(with any numbers), chances are very good that the file is Encapsulated PostScript and will work easily with dvips. If the file contains instead a line like

```
%%BoundingBox: (atend)
```

the file is still probably Encapsulated PostScript, but the bounding box (that dvips needs to position the graphic) is at the end of the file and should be moved to the position of the line above. This can be done with that same text editor, or with a simple Perl script.

If the document lacks a bounding box altogether, one can easily be added. Simply print the file. Now, take a ruler, and make the following measurements. All measurements should be in PostScript units, so measure it in inches and multiply by 72. Alternatively, the `bbfig' program distributed with dvips in the `contrib' directory can be used to automate this process.

From the left edge of the paper to the leftmost mark on the paper is llx, the first number. From the bottom edge of the paper to the bottommost mark on the paper is lly, the second number. From the left edge of the paper to the rightmost mark on the paper is urx, the third number. The fourth and final number, ury, is the distance from the bottom of the page to the uppermost mark on the paper.

Now, add a comment of the following form as the second line of the document. (The first line should already be a line starting with the two characters `%!'; if it is not, the file probably isn't PostScript.)

```
%%BoundingBox: llx lly urx ury
```

Or, if you don't want to modify the file, you can simply write these numbers down in a convenient place and use them when you import the graphic.

If the document does not have such a bounding box, or if the bounding box is given at the end of the document, please complain to the authors of the software package that generated the file; without such a line, including PostScript graphics can be tedious.

Using the EPSF Macros

Now you are ready to include the graphic into a TeX file. Simply add to the top of your TeX file a line like

@flindex epsf.tex

```
\input epsf
```

(or, if your document is in LaTeX or SliTeX, add the `epsf' style option, as was done to the following line).

```
\documentstyle[12pt,epsf]{article}
```

This only needs to be done once, no matter how many figures you plan to include. Now, at the point you want to include the file, enter a line such as

```
\epsffile{foo.ps}
```

If you are using LaTeX or SliTeX, you may need to add a `\leavevmode' command immediately before the `\epsffile' command to get certain environments to work correctly. If your file did not (or does not currently) have a bounding box comment, you should supply those numbers you wrote down as in the following example:


```
\epsffile[100 100 500 500]{foo.ps}
```

(in the same order they would have been in a normal bounding box comment). Now, save your changes and run TeX and dvips; the output should have your graphic positioned at precisely the point you indicated, with the proper amount of space reserved.

The effect of the `epsffile' macro is to typeset the figure as a TeX `vbox' at the point of the page that the command is executed. By default, the graphic will have its `natural' width (namely, the width of its bounding box). The TeX box will have depth zero and a `natural' height. The graphic will be scaled by any `dvi' magnification in effect at the time.

Any PostScript graphics included by any method in this document (except `bop-hook' and its ilk) are scaled by the current `dvi' magnification. For graphics included with `epsffile' where the size is given in TeX dimensions, this scaling will produce the correct, or expected, results. For compatibility with old PostScript drivers, it is possible to turn this scaling off with the following TeX command:

```
\special{! /magscale false def}
```

Use of this command is not recommended because it will make the `epsffile' graphics the wrong size if global magnification is used in a `dvi' document, and it will cause any PostScript graphics to appear improperly scaled and out of position if a `dvi' to `dvi' program is used to scale or otherwise modify the document.

You can enlarge or reduce the figure by putting

```
\epsfxsize=dimen
```

right before the call to `epsffile'. Then the width of the TeX box will be dimen and its' height will be scaled proportionately. Alternatively you can force the vertical size to a particular size with

```
\epsfysize=dimen
```

in which case the height will be set and the width will be scaled proportionally. If you set both, the aspect ratio of the included graphic will be distorted but both size specifications will be honored.

By default, clipping is disabled for included EPSF images. This is because clipping to the bounding box dimensions often cuts off a small portion of the figure, due to slightly inaccurate bounding box arguments. The problem might be subtle; lines around the boundary of the image might be half their intended width, or the tops or bottoms of some text annotations might be sliced off. If you want to turn clipping on, just use the command

```
\epsfclipon
```

and to turn clipping back off, use

```
\epsfclipoff
```

A more general facility for sizing is available by defining the `epsfsize' macro. You can redefine this macro to do almost anything. This TeX macro is passed two parameters by `epsffile'. The first parameter is the

natural horizontal size of the PostScript graphic, and the second parameter is the natural vertical size. 'I macro is responsible for returning the desired horizontal size of the graph (the same as assigning `epsfx above).

In the definitions given below, only the body is given; it should be inserted in

```
\def\epsfsize#1#2{body}
```

Some common definitions are:

`epsfxsize'

This definition (the default) enables the default features listed above, by setting `epsfxsize' to the same value it had before the macro was called.

`0pt'

This definition forces natural sizes for all graphics by setting the width to zero, which turns on horizontal scaling.

`#1'

This forces natural sizes too, by returning the first parameter only (the natural width) and setting the width to it.

`hsize'

This forces all graphics to be scaled so they are as wide as the current horizontal size. (In LaTeX, use `textwidth' instead of `hsize'.)

`0.5#1'

This scales all figures to half of their natural size.

`\ifnum#1>\hsize\hsize\else#1\fi'

This keeps graphics at their natural size, unless the width would be wider than the current `hsize', in which case the graphic is scaled down to `hsize'.

If you want TeX to report the size of the figure as a message on your terminal when it processes each figure, give the command

```
\epsfverbosetrue
```

Header Files

Often in order to get a particular graphic file to work, a certain header file might need to be sent first. Sometimes this is even desirable, since the size of the header macros can dominate the size of certain PostScript graphics files. The dvips program provides support for this with the `header=' special command. For instance, to ensure that `foo.ps' gets downloaded as part of the header material, the following command should be added to the TeX file:

```
\special{header=foo.ps}
```

The dictionary stack will be at the `userdict' level when these header files are included.

For these and all other header files (including the headers required by dvips itself and any downloaded fonts), the printer VM budget is debited by some value. If the header file has, in its first 1024 bytes, a l of the form

```
%%VMusage: min max
```

then the maximum value is used. If it doesn't, then the total size of the header file in bytes is used as an approximation of the memory requirements.

Literal PostScript

For simple graphics, or just for experimentation, literal PostScript graphics can be included. Simply use a special command that starts with a double quote (`"').

For instance, the following (simple) graphic: was created by typing:

```
\vbox to 100bp{\vss % a bp is the same as a PostScript unit
\special{" newpath 0 0 moveto 100 100 lineto 394 0 lineto
closepath gsave 0.8 setgray fill grestore stroke}}
```

(You are responsible for leaving space for such literal graphics.) Literal graphics are discouraged because of their nonportability.

Literal Headers

Similarly, you can define your own macros for use in such literal graphics through the use of literal macros. Literal macros are defined just like literal graphics, only you begin the special with an exclamation point instead of a double quote. These literal macros are included as part of the header material in a special dictionary called `SDict'. This dictionary is the first one on the PostScript dictionary stack when any PostScript graphic is included, whether by literal inclusion or through the `epsffile' macros.

Other Graphics Support

There are other ways to include graphics with dvips. One is to use an existing package, such as `emtex', `psfig', `tpic', or `METAPOST', all of which `dvips' supports.

Other facilities are available for historical reasons, but their use is discouraged, in hope that some `sane' form of PostScript inclusion shall become standard. The main advantage of the `epsffile' macros is that they can be adapted for whatever form of special eventually becomes standard, and thus only minor modifications to that one file need to be made, rather than revising an entire library of TeX documents.

Most of these specials use a flexible key and value scheme:

```
\special{psfile=filename.ps[key=value]*}
```

This will download the PostScript file called `filename.ps' such that the current point will be the

origin of the PostScript coordinate system. The optional key/value assignments allow you to specify transformations on the PostScript.

The possible keys are:

`hoffset'
: The horizontal offset (default 0)

`voffset'
: The vertical offset (default 0)

`hsize'
: The horizontal clipping size (default 612)

`vsize'
: The vertical clipping size (default 792)

`hscale'
: The horizontal scaling factor (default 100)

`vscale'
: The vertical scaling factor (default 100)

`angle'
: The rotation (default 0)

`clip'
: Enable clipping to the bounding box

The dimension parameters are all given in PostScript units. The `hscale' and `vscale' are given in non-dimensioned percentage units, and the rotation value is specified in degrees. Thus

```
\special{psfile=foo.ps hoffset=72 hscale=90 vscale=90}
```

will shift the graphics produced by file `foo.ps' right by one inch and will draw it at 0.9 times normal size. Offsets are given relative to the point of the special command, and are unaffected by scaling or rotation. Rotation is counterclockwise about the origin. The order of operations is to rotate the figure, scale it, then offset it.

For compatibility with older PostScript drivers, it is possible to change the units that `hscale' and `vscale' are given in. This can be done by redefining `@scaleunit' in `SDict' by a TeX command such as

```
\special{! /@scaleunit 1 def}
```

The `@scaleunit' variable, which is by default 100, is what `hscale' and `vscale' are divided by to yield an absolute scale factor.

All of the methods for including graphics we have described so far enclose the graphic in a PostScript save/restore pair, guaranteeing that the figure will have no effect on the rest of the document. Another type of special command allows literal PostScript instructions to be inserted without enclosing them in this protective shield; users of this feature are supposed to understand what they are doing (and they shouldn't change the PostScript graphics state unless they are willing to take the consequences). This command can

take many forms, because it has had a torturous history; any of the following will work:

```
\special{ps:text}
\special{ps::text}
\special{ps::[begin]text}
\special{ps::[end]text}
```

(with longer forms taking precedence over shorter forms, when they are used). Note that `ps::' and `ps::[end]' do not do any positioning, so they can be used to continue PostScript literals started with `ps:' or `ps::[begin]'. There is also the command

```
\special{ps: plotfile filename}
```

which will copy the commands from filename verbatim into the output (but omitting lines that begin with %). An example of the proper use of literal specials can be found in the file `rotate.tex' which makes it easy to typeset text turned 90 degrees.

To finish off this section, the following examples are presented without explanation:

```
\def\rotninety{\special{ps:currentpoint currentpoint translate 90
rotate neg exch neg exch translate}}\font\huge=cmbx10 at 14.4truept
\setbox0=\hbox to0pt{\huge A\hss}\vskip16truept\centerline{\copy0
\special{ps:gsave}\rotninety\copy0\rotninety\copy0\rotninety
\box0\special{ps:grestore}}\vskip16truept

\vbox to 2truein{\special{ps:gsave 0.3 setgray}\hrule height 2in
width\hsize\vskip-2in\special{ps:grestore}\font\big=cminch\big
\vss\special{ps:gsave 1 setgray}\vbox to 0pt{\vskip2pt
\line{\hss\hskip4pt NEAT\hss}\vss}\special{ps:0 setgray}%
\hbox{\raise2pt\line{\hss NEAT\hss}\special{ps:grestore}}\vss}
```

Some caveats are in order when using the above forms. Make sure that each `gsave' on a page is matched with a `grestore' on the same page. Do not use `save' or `restore'. Use of these macros can interact with the PostScript generated by dvips if care is not taken; try to understand what the above macros are doing before writing your own. The `\rotninety' macro especially has a useful trick that appears again and again.

Dynamic Creation of PostScript Graphics Files

PostScript is an excellent page description language--but it does tend to be rather verbose. Compressing PostScript graphics files can often reduce them by more than a factor of five. For this reason, if the filename parameter to one of the graphics inclusion techniques starts with a backtick (``'), the filename is instead interpreted as a command to execute that will send the actual file to standard output. Thus,

```
\special{psfile="`zcat foo.ps.Z"}
```

will include the uncompressed version of `foo.ps'. Since such a command is not accessible to TeX, if

you use this facility with the `EPSF' macros, you need to supply the bounding box parameter yourself,

```
\epsffile[72 72 540 720]{"`zcat screendump.ps.Z"}
```

to include `screendump.ps'. Of course, the commands to be executed can be anything, including using a file conversion utility such as `tek2ps' or whatever is appropriate.

This extension is not portable to other DVI-to-PostScript translators.

Using PostScript Fonts

Thanks to Donald E. Knuth, the dvips driver now supports PostScript fonts through the virtual font capability. PostScript fonts are (or should be) accompanied by a font metric file such as `Times-Roman.afm', which describes characteristics of a font called Times-Roman. To use such fonts with TeX, we need `tfm' files that contain similar information. These can be generated from `afm' files by running the program `afm2tfm', supplied with dvips. This program also creates virtual fonts with which you can use normal plain TeX conventions. See Donald E. Knuth, TUGboat v. 11, no. 1, Apr. 1990, pp. 13--23, "Virtual Fonts: More Fun for Grand Wizards", for a general introduction to virtual fonts.

Non-resident downloaded PostScript fonts tend to use a great deal of printer virtual memory. PostScript printers typically have between 130,000 and 400,000 bytes of available virtual memory; each downloaded font will require approximately 30,000 bytes of that. For many applications, bitmapped fonts work much better, even at typesetter resolutions (with the `-Z' option).

Even resident PostScript fonts can take a fair amount of memory, but less with this release of dvips than previously. Also, bitmapped fonts tend to image faster than PostScript fonts.

The Afm2tfm Program

In its simplest form, the `afm2tfm' program converts an Adobe `afm' file (e.g., `Times-Roman.afm') into a `raw' TeX TFM file (say, `rptmr.tfm' with the command

```
afm2tfm Times-Roman rptmr
```

This file is raw because it does no character remapping; it simply converts the character information on a one-to-one basis to TeX characters *with the same code*. (It would be better to have eschewed the use of the `r' prefix in favor of using a variant letter to specify the encoding; but we didn't think of that at the time, and it's too late to change existing fonts. See section 'Top' in Filenames for TeX fonts.)

In the following examples, we will use the font Times-Roman to illustrate the conversion process. For the standard 35 LaserWriter fonts, however, it is highly recommended that you use the supplied `tfm' and `vf' files that come with dvips, as these files contain some additional changes that make them work better with TeX than they otherwise would.

Standard PostScript fonts have a different encoding scheme from that of plain TeX. Although both schemes are based on ASCII, special characters such as ligatures and accents are handled quite differently.

Therefore we obtain best results by using a `virtual font' interface, which makes TeX see a standard Te encoding for the PostScript font. Such a virtual font can be obtained, for example, by the command

```
afm2tfm Times-Roman -v ptmr rptmr
```

This produces two files as output, namely the `virtual property list' file `ptmr.vpl', and the raw font metric file `rptmr.tfm'. (The upper case `-V' also produces a `vpl' file, but maps a set of small caps into the lower case alphabet.) To use the font in TeX, you should first run

```
vptovf ptmr.vpl ptmr.vf ptmr.tfm
```

and then install the virtual font file `ptmr.vf' where Dvips will see it, and install `ptmr.tfm' and `rptmr.tfm' where TeX and Dvips will see them.

You can also make more complex virtual fonts by editing `ptmr.vpl' before running `vptovf'; such editing might add the uppercase Greek characters in the standard TeX positions, for instance. (This has already been done for the font files that come with Dvips.) Once this has been done, you're all set. You can use code like this in TeX henceforth:

```
\font\myfont = ptmr at 12pt
\myfont Hello, I am being typeset in 12-point Times-Roman.
```

Thus we have two fonts, one actual (`rptmr', which is analogous to the font in the printer) and one virtual (`ptmr', which has been remapped to the standard TeX encoding (almost), and has typesetting know-how added. You could also say

```
\font\PTR = rptmr at 10pt
```

and typeset directly with that, but then you would have no ligatures or kerning, and you would have to use Adobe character positions for special letters like The virtual font called `ptmr' not only has ligatures and kerning, and most of the standard accent conventions of TeX, it also has a few additional features not present in the Computer Modern fonts. For example, it includes all the Adobe characters (such as the Polish ogonek and the French guillemots). The only things you lose from ordinary TeX text fonts are the dotless `j' (which can be hacked into the VPL file with literal PostScript specials if you have the patience) and uppercase Greek letters (which just don't exist unless you buy them separately).

The remapped `p*` fonts mostly follow plain TeX conventions for accents. The exceptions: the Hungarian umlaut (which is at position `0x7D' in `cmr10', but position `0xCD' in `ptmr'); the dot accent (at positions `0x5F' and `0xC7', respectively); and `\AA', which neds different tweaking. In order to use these accents with PostScript fonts or in math mode when `\textfont0' is a PostScript font, you will need to use the following definitions. These definitions will not work with the Computer Modern fonts for the relevant accents. They are already part of the distributed `psfonts.sty' for use with LaTeX.

```
\def\H#1{{\accent"CD #1}}
\def\.#1{{\accent"C7 #1}}
\def\dot{\mathaccent"70C7 }
\newdimen\aadimen
```


```
\def\AA{\leavevmode\setbox0\hbox{h}\aadimen\ht0
  \advance\aadimen-1ex\setbox0\hbox{A}\rlap{\raise.67\aadimen
  \hbox to \wd0{\hss\char'27\hss}}A}
```

These PostScript fonts can be scaled to any size. Go wild! Using PostScript fonts, however, does use up a great deal of the printer's memory and it does take time. You may find downloading bitmapped fonts to be faster than using the built-in PostScript fonts!

Changing a Font's Encoding

The `afm2tfm' program also allows you to specify a different encoding for a PostScript font. This can be done at two levels.

You can specify a different output encoding with `-t'. This only applies when you are building a virtual font, and it tells `afm2tfm' to attempt to remap the font to match the output encoding as closely as possible. In such an output encoding, you can also specify ligature pairs and kerning information that will be used in addition to the information in the `afm' file.

You can also specify a different PostScript encoding with `-p'. This changes both the raw `tfm' file created by Afm2tfm and the `vf' and `tfm' files created by VPtoVF. It also requires that the encoding file be downloaded as part of any document that uses this font. This is the only way to access characters in a PostScript font that are neither encoded in the `afm' file nor built from other characters (constructed characters). For instance, `Times-Roman' contains the extra characters `trademark' and `registered' (among others) that can only be accessed through such a PostScript reencoding. Any ligature or kern information specified in the PostScript encoding is ignored by `afm2tfm'.

To combine the effects of `-p' and `-t', use `-T'. If you make regular use of a private non-standard reencoding `-T' is usually the better idea, since you can get unexpected inconsistencies in mapping otherwise.

Afm2tfm's encoding files have precisely the same format as an encoding vector in any PostScript font. Specifically:

```
% Comments are ignored, unless the first word after the percent sign
% is LIGKERN -- see below.
/MyEncoding [ /Alpha /Beta /Gamma /Delta ...
     /A /B ... /Z % exactly 256 entries, each with a / at the front
     /wfooaccent /xfooaccent /yfooaccent /zfooaccent ] def
```

Comments, which start with a percent sign and continue until the end of the line, are ignored unless they start with `LIGKERN'. The first `word' of the file must start with a forward slash (a PostScript literal name) and defines the name of the encoding. The next word must be an left bracket `['. Following that must be precisely 256 character names; use `/.notdef' for any that you do not want defined. Finally, there must be a matching right bracket `]'. A final `def' token is optional. All names are case sensitive.

Any ligature or kern information is given in the comments. If the first word after the percent sign is `LIGKERN', then the entire rest of the line is parsed for ligature and kern information. This ligature and kern information is given in groups of words, each group of which must be terminated by a semicolon (with

a space before and after it, unless it occurs at the end of a line.)

In these `LIGKERN' statements, three types of information may be specified. These three types are ligature pairs, kerns to remove or ignore, and the character value of this font's boundary character. Which of the types the particular set of words corresponds to is automatically determined by the allowable syntax.

Throughout a `LIGKERN' statement, the boundary character is specified as `||'.

To set the boundary character value, a command such as `|| = 39 ;' must be used.

To indicate a kern to remove, give the names of the two characters (without the leading slash) separated by `{}', as in `one {} one ;'. This is similar to the way you might use `{}' in a TeX file to turn off ligatures or kerns at a particular location. Either or both of the character names can be given as `*', which is a wild card matching any character; thus, all kerns can be removed with `* {} * ;'.

To specify a ligature, specify the names of the pair of characters, followed by the ligature `operation' (as in METAFONT), followed by the replacing character name. Either (but not both) of the first two characters can be `||' to indicate a word boundary.

Normally the `operation' is `=:' meaning that both characters are removed and replaced by the third character, but by adding `|' characters on either side of the `=:', you can specify which of the two leading characters to retain. In addition, by suffixing the ligature operation with one or two `>' signs, you can indicate that the ligature scanning operation should skip that many characters before proceeding. This works just like in METAFONT. A typical ligature might be specified with `ff i =: ffi ;'. A more convoluted ligature is `one one |=:|>> exclam ;' which indicates that every pair of adjacent `1"s should be separated by an exclamation point, and then two of the resulting characters should be skipped over before continuing searching for ligatures and kerns. You cannot give more `>"s in an ligature operation as you did `|', so there are a total of eight possible ligature operations:

```
=: |=: |=:> =:| =:|> |=:| |=:|> |=:|>>
```

The default set of ligatures and kerns built in to `afm2tfm' can be specified with:

```
% LIGKERN question quoteleft =: questiondown ;
% LIGKERN exclam quoteleft =: exclamdown ;
% LIGKERN hyphen hyphen =: endash ; endash hyphen =: emdash ;
% LIGKERN quoteleft quoteleft =: quotedblleft ;
% LIGKERN quoteright quoteright =: quotedblright ;
% LIGKERN space {} * ; * {} space ; 0 {} * ; * {} 0 ;
% LIGKERN 1 {} * ; * {} 1 ; 2 {} * ; * {} 2 ; 3 {} * ; * {} 3 ;
% LIGKERN 4 {} * ; * {} 4 ; 5 {} * ; * {} 5 ; 6 {} * ; * {} 6 ;
% LIGKERN 7 {} * ; * {} 7 ; 8 {} * ; * {} 8 ; 9 {} * ; * {} 9 ;
```

Special Font Effects

Afm2tfm can do other manipulations as well. For example, to make an obliqued variant of Times Roman:

```
afm2tfm Times-Roman -s .167 -v ptmro rptmro
```

which creates `ptmro.vpl' and `rptmro.tfm'. To use this, put the line

```
rptmro Times-Roman ".167 SlantFont"
```

@flindex psfonts.map into `psfonts.map'. Then `rptmro' (our name for the obliqued Times) will act as if it were a resident font, although it is actually constructed from Times-Roman via the PostScript routine `SlantFont` (which slants everything 1/6 to the right, in this case).

Similarly, you can get an expanded font with

```
afm2tfm Times-Roman -e 1.2 -v ptmrre rptmrre
```

and by recording the pseudo-resident font

```
rptmrre Times-Roman "1.2 ExtendFont"
```

in `psfonts.map'.

You can also create a small caps font with a command such as

```
afm2tfm Times-Roman -V ptmrc rptmrc
```

This will generate a set of small caps mapped into the usual lower case positions and scaled down to 0.8 of the normal cap dimensions. You can also specify the scaling as something other than the default 0.8:

```
afm2tfm Times-Roman -c 0.7 -V ptmrc rptmrc
```

It is unfortunately not possible to increase the width of the small caps independently of the rest of the font. If you want a really professional looking set of small caps you will have to make up a custom pair of ``tfm'' and ``vpl'' files using the `-e' option, change the name and checksum of the D 1 font in `ptmrc.vpl' to match the customized raw `tfm', and replace the small caps at the lower case mappings with the small caps in the customized file. You can then throw away the customized `vpl' file, but not the customized `tfm'. That must be identified with an appropriate line in `psfonts.map'.

If you change the PostScript encoding of a font, you must specify the input file as a header file, as well as give a reencoding command. For instance, let us say we are using Times-Roman reencoded according to the encoding `MyEncoding' (stored in the file `myenc.enc') as `rptmrx'. In this case, our `psfonts.map' entry would look like

```
rptmrx Times-Roman "MyEncoding ReEncodeFont" <myenc.enc
```

The `afm2tfm' program prints out the precise line you need to add to `psfonts.map' to use that font, assuming the font is resident in the printer; if the font is not resident, you must add the header command to download the font yourself. Each identical line only needs to be specified once in the `psfonts.map' file, even though many different fonts (small caps variants, or ones with different output encodings) may be based on it.

Non-Resident PostScript Fonts

If you want to use a non-printer-resident PostScript font for which you have a `pfb' or `pfa' file (an Adobe Type 1 font program), you can make it act like a resident font by putting a `<' sign and the name of the `pfb' or `pfa' file just after the font name in the `psfonts.map' @flindex psfonts.map file entry. For example,

```
rpstrn StoneInformal <StoneInformal.pfb
```

will cause dvips to include `StoneInformal.pfb' in your document as if it were a header file, whenever the pseudo-resident font StoneInformal is used in a document. Similarly, you can generate transformed fonts and include lines like

```
rpstrc StoneInformal <StoneInformal.pfb ".8 ExtendFont"
```

in `psfonts.map', in which case `StoneInformal.pfb' will be loaded whenever StoneInformal-Condensed is used. Each header file is loaded at most once per document.

If you are using a `pfb' file that is also has a different PostScript encoding, you need to download multiple header files in `psfonts.map'. If, for instance, `Optima' was both non-resident and you wanted to reencode it in PostScript with `MyEncoding' stored in `myenc.enc', you need:

```
rpstrnx Optima "MyEncoding ReEncodeFont" <myenc.enc <Optima.pfb
```

When using PFB files, dvips is smart enough to unpack the binary PFB format into printable ASCII so there is no need to perform this conversion yourself. In addition, it will scan the font to determine its memory usage, as it would for any header file.

Here is a brief description of the contents of `psfonts.map'. If a line is empty or begins with a space, asterisk, semicolon, or hash mark, it is ignored. Otherwise, the line is separated into words, where words are separated by spaces or tabs, except that if a word begins with a double quote, it extends until the next double quote or the end of the line. If a word starts with a less than character, it is treated as a font header file (or a downloaded PostScript font). There can be more than one such header for a given font. If a word starts with a double quote, it is a special instruction for generating that font. Otherwise it is a name. The first such name is the TFM file that a virtual font file can refer to. If there is another name, it is used as the PostScript name; if there is only one name, it is used for both the TeX name and the PostScript name.

Invoking Afm2tfm

The command line switches to `afm2tfm' are:

`-c ratio'

See `-V'; overrides the default ratio of 0.8 for the scaling of small caps.

`-e ratio'

All characters are stretched horizontally by the stated ratio; if it is less than 1.0, you get a condensed font.

`-O'

This option forces all character codes in the output `vpl' file be given as octal values; this is useful for symbol or other special-purpose fonts where character names such as `A' have no meaning.

`-p file'

This specifies a file to use for the PostScript encoding of the font. This file must also be mentioned as a header file for the font in `psfonts.map', and that ligature and kern information in this file is ignored.

`-T file'

This option specifies that file is to be used for both the PostScript and target TeX encodings of the font.

`-t file'

This specifies a file to use for the target TeX encoding of the font. Ligature and kern information may also be specified in this file; the file need not be mentioned in `psfonts.map'.

`-s slant'

All characters are slanted to the right by slant. If slant is negative, the letters slope to the left (or they might be upright if you start with an italic font).

`-u'

This option indicates that `afm2tfm' should use only those characters that are required by the output encoding, and no others. Normally, `afm2tfm' tries to include both characters that fit the output encoding and any additional characters that might exist in the font. This option forbids those additional characters from being added.

`-v file'

Generate a virtual property list `vpl' file as well as a `tfm' file.

`-V file'

Same as `-v', but the virtual font generated is a small caps font obtained by scaling uppercase letters by 0.8 to typeset lowercase. This font handles accented letters and retains proper kerning.

Here are the `CODINGSCHEMES' that result from the various possible choices for reencoding.

`default encoding'

```
(CODINGSCHEME TeX text + AdobeStandardEncoding)
```

`-p DC.enc'

```
(CODINGSCHEME TeX text + DCEncoding)
```

`-t DC.enc'

```
(CODINGSCHEME DCEncoding + AdobeStandardEncoding)
```

`-T DC.enc'

```
(CODINGSCHEME DCEncoding + DCEncoding)
```

Command Line Options

The dvips driver has a plethora of command line options. Reading through this section will give a good idea of the capabilities of the driver.

Many of the parameterless options listed here can be turned off by immediately suffixing the option with a zero (0); for instance, to turn off page reversal if it is turned on by default, use `-r0'. The options that can be turned off in this way are `a', `f', `k', `i', `m', `q', `r', `s', `E', `F', `K', `M', `N', `U', and `Z'.

This is a handy summary of the options; it is printed out when you run dvips with no arguments.

```
This is dvipsk version Copyright 1986, 1993 Radical Eye Software
     Usage: dvips [options] filename[.dvi]
 a*  Conserve memory, not time      y # Multiply by dvi magnification
 b # Page copies, for posters e.g.  A   Print only odd (TeX) pages
 c # Uncollated copies              B   Print only even (TeX) pages
 d # Debugging                      C # Collated copies
 e # Maxdrift value                 D # Resolution
 f*  Run as filter                  E*  Try to create EPSF
 h f Add header file                F*  Send control-D at end
 i*  Separate file per section      K*  Pull comments from inclusions
 k*  Print crop marks               M*  Don't make fonts
 l # Last page                      N*  No structured comments
 m*  Manual feed                    O c Set/change paper offset
 n # Maximum number of pages        P s Load config.$s
 o f Output file                    R   Run securely
 p # First page                     S # Max section size in pages
 q*  Run quietly                    T c Specify desired page size
 r*  Reverse order of pages         U*  Disable string param trick
 s*  Enclose output in save/restore V*  Send downloadable PS fonts as PK
 t s Paper format                   X # Horizontal resolution
 x # Override dvi magnification     Y # Vertical resolution
                                    Z*  Compress bitmap fonts
 pp #-# First-page
     # = number   f = file   s = string  * = suffix, `0' to turn off
     c = comma-separated dimension pair (e.g., 3.2in,-32.1cm)
```

`-a'

Conserve memory by making three passes over the `dvi' file instead of two and only loading those characters actually used. Generally only useful on machines with a very limited amount of memory, like some PCs.

`-c num'

Generate num copies of every page. Default is 1. (For collated copies, see the `-C' option below.)

`-b num'

Generate num copies of each page, but duplicating the page body rather than using the `#numcopies'

option. This can be useful in conjunction with a header file setting `\bop-hook' to do color separations or other neat tricks.

`-d num'

Set the debug flags. This is intended only for emergencies or for unusual fact-finding expeditions; it will work only if dvips has been compiled with the `DEBUG' option. See section Debug Options, for the possible values of num. Use a value of `-1' for maximum output.

`-e num'

Make sure that each character is placed at most this many pixels from its `true' resolution-independent position on the page. The default value of this parameter is resolution dependent (it is the number of entries in the list [100, 200, 300, 400, 500, 600, 800, 1000, 1200, 1600, 2000, 2400, 2800, 3200, ...] that are less than or equal to the resolution in dots per inch). Allowing individual characters to `drift' from their correctly rounded positions by a few pixels, while regaining the true position at the beginning of each new word, improves the spacing of letters in words.

`-f'

Run as a filter. Read the `dvi' file from standard input and write the PostScript to standard output. The standard input must be seekable, so it cannot be a pipe. If you must use a pipe, write a shell script that copies the pipe output to a temporary file and then points dvips at this file. This option also disables the automatic reading of the `PRINTER' environment variable, and turns off the automatic sending of control D if it was turned on with the `-F' option or in the configuration file; use `-F' after this option if you want both.

`-h name'

Prepend file name as an additional header file. (However, if the name is simply `-', suppress all header files from the output.) This header file gets added to the PostScript `userdict'.

`-i'

Make each section be a separate file. Under certain circumstances, dvips will split the document up into `sections' to be processed independently; this is most often done for memory reasons. Using this option tells dvips to place each section into a separate file; the new file names are created replacing the suffix of the supplied output file name by a three-digit sequence number. This option is most often used in conjunction with the `-S' option which sets the maximum section length in pages. For instance, some phototypesetters cannot print more than ten or so consecutive pages before running out of steam; these options can be used to automatically split a book into ten-page sections, each to its own file.

`-k'

Print crop marks. This option increases the paper size (which should be specified, either with a paper size special or with the `-T' option) by a half inch in each dimension. It translates each page by a quarter inch and draws cross-style crop marks. It is mostly useful with typesetters that can set the page size automatically.

`-l num'

The last page printed will be the first one numbered num. Default is the last page in the document. If the num is prefixed by an equals sign, then it (and any argument to the `-p' option) is treated as a sequence number, rather than a value to compare with `\count0' values. Thus, using `-l =9' will end with the ninth page of the document, no matter what the pages are actually numbered.

`-m'

Specify manual feed for printer.

`-n num'

At most num pages will be printed. Default is 100000.

`-o name'

The output will be sent to file name. If no file name is given, the default name is `file.ps' where the `dvi' file was called `file.dvi'; if this option isn't given, any default in the configuration file is used. If the first character of the supplied output file name is an exclamation mark, then the remainder will be used as an argument to `popen'; thus, specifying `!lpr' as the output file will automatically queue the file for printing. This option also disables the automatic reading of the `PRINTER' environment variable, and turns off the automatic sending of control D if it was turned on with the `-F' option or in the configuration file; use `-F' after this option if you want both.

`-p num'

The first page printed will be the first one numbered num. Default is the first page in the document. If the num is prefixed by an equals sign, then it (and any argument to the `-l' option) is treated as a sequence number, rather than a value to compare with `\count0' values. Thus, using `-p =3' will start with the third page of the document, no matter what the pages are actually numbered.

`-pp first-last'

@opindex -pp range Print pages first through last; equivalent to `-p first -l last'. The `-' range separator can also be a `:'.

`-q'

Run in quiet mode. Don't chatter about pages converted, etc.; report nothing but errors to standard error.

`-r'

Stack pages in reverse order. Normally, page 1 will be printed first.

`-s'

Causes the entire global output to be enclosed in a save/restore pair. This causes the file to not be truly conformant, and is thus not recommended, but is useful if you are driving the printer directly and don't care too much about the portability of the output.

`-t papertype'

This sets the paper type to papertype. The papertype should be defined in one of the configuration files, along with the appropriate code to select it. See the documentation for `@'in the configuration file option descriptions. You can also specify `-t landscape', which rotates a document by 90 degrees. To rotate a document whose size is not letter, you can use the `-t' option twice, once for the page size, and once for `landscape'. The upper left corner of each page in the `dvi' file is placed one inch from the left and one inch from the top. Use of this option is highly dependent on the configuration file. Note that executing the `letter' or `a4' or other PostScript operators cause the document to be nonconforming and can cause it not to print on certain printers, so the default paper size should not execute such an operator if at all possible.

`-x num'

Set the magnification ratio to num/1000. Overrides the magnification specified in the `dvi' file. Must be between 10 and 100000. It is recommended that you use standard magstep values (1095, 1200,

1440, 1728, 2074, 2488, 2986, and so on) to help reduce the total number of PK files generated.

`-A'

This option prints only the odd pages. This option uses the \TeX\ page numbering rather than the sequence page numbers.

`-B'

This option prints only the even pages. This option uses the \TeX\ page numbering rather than the sequence page numbers.

`-C num'

Create num copies, but collated (by replicating the data in the PostScript file). Slower than the `-c' option, but easier on the hands, and faster than resubmitting the same PostScript file multiple times.

`-D num'

Set the resolution in dpi (dots per inch) to num. This affects the choice of bitmap fonts that are loaded and also the positioning of letters in resident PostScript fonts. Must be between 10 and 10000. This affects both the horizontal and vertical resolution. If a high resolution (something greater than 400 dpi, say) is selected, the `-Z' flag should probably also be used.

`-E'

Makes dvips attempt to generate an EPSF file with a tight bounding box. This only works on one-page files, and it only looks at marks made by characters and rules, not by any included graphics. In addition, it gets the glyph metrics from the `tfm' file, so characters that lie outside their enclosing `tfm' box may confuse it. In addition, the bounding box might be a bit too loose if the character glyph has significant left or right side bearings. Nonetheless, this option works well for creating small EPSF files for equations or tables or the like. (Of course, `dvips' output is resolution dependent and thus does not make very good EPSF files, especially if the images are to be scaled; use these EPSF files with a great deal of care.)

`-F'

Causes Control-D (ASCII code 4) to be appended as the very last character of the PostScript file. This is useful when dvips is driving the printer directly instead of working through a spooler, as is common on extremely small systems. Otherwise, it is not recommended.

`-K'

This option causes comments in included PostScript graphics, font files, and headers to be removed. This is sometimes necessary to get around bugs in spoolers or PostScript post-processing programs. Specifically, the `%%Page' comments, when left in, often cause difficulties. Use of this flag can cause some included graphics to fail, since the PostScript header macros from some software packages read portions of the input stream line by line, searching for a particular comment. This option has been turned off by default because PostScript previewers and spoolers have been getting better.

`-M'

Turns off the automatic font generation facility. If any fonts are missing, commands to generate the fonts are appended to the file `missfont.log' in the current directory; this file can then be executed and deleted to create the missing fonts.

`-N'

Turns off structured comments; this might be necessary on some systems that try to interpret

PostScript comments in weird ways, or on some PostScript printers. Old versions of TranScript i particular cannot handle modern Encapsulated PostScript.

`-O offset'

Move the origin by a certain amount. The offset is a comma-separated pair of dimensions, such as `.1in,-.3cm' (in the same syntax used in the `papersize' special). The origin of the page is shifted from the default position (of one inch down, one inch to the right from the upper left corner of the paper) by this amount.

`-P printername'

Sets up the output for the appropriate printer. This is implemented by reading in `config.printername', which can then set the output pipe (as in, `o !lpr -Pprintername') as well as the font paths and any other defaults for that printer only. It is recommended that all standard defaults go in the one master `config.ps' file and only things that vary printer to printer go in the `config.printername' files. Note that `config.ps' is read before `config.printername'. In addition, another file called `~/.dvipsrc' is searched for immediately after `config.ps'; this file is intended for user defaults. If no `-P' command is given, the environment variable `PRINTER' is checked. If that variable exists, and a corresponding configuration file exists, that configuration file is read in.

`-S num'

Set the maximum number of pages in each `section'. This option is most commonly used with the `-i' option; see that documentation above for more information.

`-T offset'

Set the paper size to the given pair of dimensions. This option takes its arguments in the same style as `-O'. It overrides any paper size special in the `dvi' file.

`-U'

Disable a PostScript virtual memory saving optimization that stores the character metric information in the same string that is used to store the bitmap information. This is only necessary when driving the Xerox 4045 PostScript interpreter. It is caused by a bug in that interpreter that results in `garbage' on the bottom of each character. Not recommended unless you must drive this printer.

`-V'

Download non-resident PostScript fonts as bitmaps. This requires use of `mtpk` or `pstopk` or some combination of the two in order to generate the required bitmap fonts; neither of these programs are supplied with Dvips.

`-X num'

Set the horizontal resolution in dots per inch to num.

`-Y num'

Set the vertical resolution in dots per inch to num.

`-Z'

Causes bitmapped fonts to be compressed before they are downloaded, thereby reducing the size of the PostScript font-downloading information. Especially useful at high resolutions or when very large fonts are used. Will slow down printing somewhat, especially on early 68000-based PostScript printers.

Configuration File Searching

@flindex config.ps

The dvips program has a system of loading configuration files such that parameters can be set globally across the system, on a per-printer basis, or by the user.

When dvips starts up, first the global `config.ps' file is searched for and loaded. This file is looked for along the path for configuration files, which is set in the `Makefile' at compilation. After this master configuration file is loaded, a file by the name of @flindex .dvipsrc `.dvipsrc' is loaded from the current user's home directory, if such a file exists. This file is loaded in exactly the same way as the global configuration file, and it can override any options set in the global file.

Then the command line is read and parsed. If the `-P' option is encountered, at that point in the command line a configuration file for that printer is read in. Thus, the printer configuration file can override anything in the global or user configuration file, and it can also override anything seen in the command line up to the point that the `-P' option was encountered.

After the command line has been completely scanned, if there was no `-P' option selected, and also the `-o' and `-f' command line options were not used, a `PRINTER' environment variable is searched for. If this variable exists, and a configuration file for the printer mentioned in it exists, this configuration file is loaded last of all.

Note that because the printer-specific configuration files are read after the user's configuration file, the user cannot override things in the printer configuration files. On the other hand, the configuration path usually includes the current directory, and can be set to include the user's home directory (or any other directory of the user), so the user can always provide his or her own printer-specific configuration files that will be found before the system global ones.

It is best to give a METAFONT mode as well as a resolution in the printer configuration file. Also make sure that METAFONT knows about the mode, by entering it into your local `mode_def' file. (For example, the file `modes.mf' @flindex modes.mf which is available from `ftp.cs.umb.edu' in `pub/tex/modes.mf'.)

The most common problem in generating fonts with METAFONT is that this file with the mode definitions is not included when creating the `plain.base' file.

Configuration File Options

Most of the configuration file options are similar to the command line options, but there are a few new ones.

Again, many may be turned off by suffixing the letter with a zero (0). These options are `a', `f', `q', `r', `K', `N', `U', and `Z'.

Within a configuration file, any empty line or line starting with a space, asterisk, equal sign, or a pound sign is ignored.

`@ name hsize vsize'

This option is used to set the paper size defaults and options for the particular printer this configuration file describes. There are three formats for this option. If the option is specified on a line by itself, with no parameters, it instructs dvips to discard all other paper size information (possibly from another configuration file) and start fresh. If three parameters are given, as above, with the first parameter being a name and the second and third being a dimension (as in 8.5in or 3.2cc, just like in the `papersize' special), then the option is interpreted as starting a new paper size description, where name is the name and hsize and vsize describe the horizontal and vertical size of the sheet of paper, respectively. If both hsize and vsize are equal to zero (although you must still specify units!) then any page size will match it. If the `@' character is immediately followed by a `+' sign, then the remainder of the line (after skipping any leading blanks) is treated as PostScript code to send to the printer to select that particular paper size. After all that, if the first character of the line is an exclamation point, then the line is put in the initial comments section of the final output file; else, it is put in the setup section of the output file. For instance, a subset of the paper size information supplied in the default `config.ps' looks like

```
@ letterSize 8.5in 11in
@ letter 8.5in 11in
@+ %%BeginPaperSize: Letter
@+ letter
@+ %%EndPaperSize
@ legal 8.5in 14in
@+ ! %%DocumentPaperSizes: Legal
@+ %%BeginPaperSize: Legal
@+ legal
@+ %%EndPaperSize
```

Note that you can even include structured comments in the configuration file! When dvips has a paper format name given on the command line, it looks for a match by the name; when it has a `papersize' special, it looks for a match by dimensions. The first match found (in the order the paper size information is found in the configuration file) is used. If nothing matches, a warning is printed and the first paper size given is used, so the first paper size should always be the default. The dimensions must match within a quarter of an inch. Landscape mode for all of the paper sizes are automatically supported. If your printer has a command to set a special paper size, then give dimensions of `0in 0in'; the PostScript code that sets the paper size can refer to the dimensions the user requested as `hsize' and `vsize'; these will be macros defined in the PostScript that return the requested size in default PostScript units. Note that virtually all of the PostScript commands you use here are device dependent and degrade the portability of the file; that is why the default first paper size entry should not send any PostScript commands down (although a structured comment or two would be okay). Also, some printers want `BeginPaperSize' comments and paper size setting commands; others (such as the NeXT) want `PaperSize' comments and they will handle setting the paper size. There is no solution I could find that works for both (except maybe specifying both.) See the supplied `config.ps' file for a more realistic example.

`a'

Conserve memory by making three passes over the `dvi' file instead of two and only loading those characters actually used. Generally only useful on machines with a very limited amount of memory, like some PCs.

`b num'

Generate num copies of each page, but duplicating the page body rather than using the `#numcopies' option. This can be useful in conjunction with a header file setting `\bop-hook' to do color separations or other neat tricks.

`e num'

Set the maximum drift parameter to num dots (pixels) as explained above.

`f'

Run as a filter by default.

`h name'

Add name as a PostScript header file to be downloaded at the beginning.

`i num'

Make each section be a separate file, and set the maximum number of pages in a given file to num. Under certain circumstances, dvips will split the document into `sections' to be processed independently; this is most often done for memory reasons. Using this option tells dvips to place each section into a separate file; the new file names are created by replacing the suffix of the supplied output file name with a three-digit sequence number. This is essentially a combination of the command line options `-i' and `-S'; see the documentation for these options for more information.

`m num'

The value num is the virtual memory available for fonts and strings in the printer. Default is 180000. This value must be accurate if memory conservation and document splitting is to work correctly. To determine this value, send the following file to the printer:

```
%! Hey, we're PostScript
/Times-Roman findfont 30 scalefont setfont 144 432 moveto
vmstatus exch sub 40 string cvs show pop showpage
```

Note that the number returned by this file is the total memory free; it is often a good idea to tell dvips that the printer has somewhat less memory. This is because many programs download permanent macros that can reduce the memory in the printer. In general, a memory size of about `300000' is plenty, since dvips can automatically split a document if required. It is unfortunate that PostScript printers with much less virtual memory still exist. Some systems or printers can dynamically increase the memory available to a PostScript interpreter, in which case this file might return a ridiculously low number; the NeXT computer is such a machine. For these systems, a value of one million works well.

`o name'

The default output file is set to name. As above, it can be a pipe. Useful in printer-specific configuration files to redirect the output to a particular printer queue.

`p name'

The file to examine for PostScript font aliases is name. It defaults to `psfonts.map'. This option allows different printers to use different resident fonts. If the name starts with a `+' character, then the rest of the name (after any leading spaces) is used as an additional map file; thus, it is possible to have local map files pointed to by local configuration files that append to the global map file.

`q'

Run in quiet mode by default.

`r'

Reverse the order of pages by default.

`s'

Enclose the entire document in a global save/restore pair by default. Not recommended, but useful in some environments; this breaks the conformance of the document to the Adobe PostScript structuring conventions.

`D num'

Set the vertical and horizontal resolution to num dots per inch. Useful in printer-specific configuration files.

`E command'

Executes the command listed; can be used to get the current date into a header file for inclusion, for instance. Possibly dangerous; in many installations this may be disabled, in which case a warning message will be printed if the option is used.

`H path'

The (colon-separated) path to search for PostScript header files is path. The environment variable `DVIPSHEADERS' overrides this.

`K'

Filter comments out of included PostScript files; see the description above for more information.

`M mode'

Set mode as the METAFONT mode to be used when generating fonts. This is passed along to `MakeTeXPK' and overrides mode derivation from the base resolution.

`N'

Disable PostScript comments by default.

`O offset'

Move the origin by a certain amount. The offset is a comma-separated pair of dimensions, such as `.1in,-.3cm' (in the same syntax as used in the `papersize' special). The origin of the page is shifted from the default position (of one inch down, one inch to the right from the upper left corner of the paper) by this amount.

`P path'

The (colon-separated) path to search for bitmap `pk' font files is path. The `PKFONTS', `TEXPKS', `GLYPHFONTS', and `TEXFONTS' environment variables override this. @xref{TeX environment variables, TeX environment variables, kpathsea, Kpathsearch library}.

`R num num ...'

Sets up a list of default resolutions to search for `pk' fonts, if the requested size is not available. The output will then scale the font found using PostScript scaling to the requested size. The resulting output may be ugly, and thus a warning is issued. To turn this off, use a line with just the `R' and no numbers.

`S path'

The path to search for special illustrations (Encapsulated PostScript files or psfiles) is path. The

`TEXPICTS' and then `TEXINPUTS' environment variables override this.

`T path'

The path to search for the `tfm' files is path. The `TEXFONTS' environment variable will override this. This path is used for resident fonts and fonts that can't otherwise be found. It's usually best to make it identical to the path used by TeX.

`U'

Turns off a memory-saving optimization; this is necessary for the Xerox 4045 printer, but not recommended otherwise. See the description above for more information.

`V path'

The path to search for virtual font `vf' files is path. This may be device-dependent if you use virtual fonts to simulate actual fonts on different devices.

`W string'

Sends string to stderr, if a parameter is given; otherwise it cancels another previous message. This is useful in the default configuration file if you want to require the user to specify a printer, for instance, or if you want to notify the user that the resultant output has special characteristics.

`X num'

Set the horizontal resolution to num dots per inch.

`Y num'

Set the vertical resolution to num dots per inch.

`Z'

Compress all downloaded fonts by default, as above.

Automatic Font Generation

One major problem with TeX and the Computer Modern fonts is the huge amount of disk space a full set of high resolution fonts can take. Dvips solves this problem by creating fonts on demand, so only those fonts that are actually used are stored on disk. At a typical site, less than one-fifth of the full set of Computer Modern fonts are used over a long period, so this saves a great deal of disk space.

Furthermore, the addition of dynamic font generation allows fonts to be used at any size, including typesetter resolutions and extremely huge banner sizes. Nothing special needs to be done; the fonts will be automatically created and installed as needed.

The downside is that it does take a certain amount of time to create a new font if it has never been used before. But once a font is created, it will exist on disk, and the next time that document is printed it will print very quickly.

It is the `MakeTeXPK' shell script that is responsible for making these fonts. It *must* echo the filename of the new font (and nothing else) to standard output. Use standard error for commentary. `MakeTeXPK' is passed various arguments, the last of which is the font it's supposed to make. You can override the other argument conventions with environment variables or at compilation time; see the Kpathsea documentation.

The `MakeTeXPK' script supplied invokes Metafont (using the Sauter scripts first, if necessary and if they

are installed) to create the font and then copies the resultant `pk' file to a world-writable font cache area.

`MakeTeXPK' can be customized to do other things to get the font. For instance, if you are installing dvips to replace (or run alongside) an existing PostScript driver, and that driver demands `gf' fonts, you can easily modify `MakeTeXPK' to invoke `gftopk' to convert the `gf' files to `pk' files for dvips. This provides the same space savings listed above.

Because dvips (and thus `MakeTeXPK') is run by a wide variety of users, there must be a system-wide place to put the cached font files. In order for everyone to be able to supply fonts, the directory must be world writable. If your system administrator considers this a security hole, `MakeTeXPK' can write to `/tmp/pk' or some such directory, and periodically the cached fonts can be moved to a more general system area. The cache directory must exist on the `pk' file search path in order for `MakeTeXPK' to work.

Environment Variables

Dvips reads a certain set of environment variables to configure its operation. The path variables are read as needed, after all configuration files are read, so they override values in the configuration files. (Except for the `TEXCONFIG' variable.)

See section 'Path specifications' in Kpathsea library, for details of interpretation of environment variable values, and the common environment variables. Only the environment specific to `dvips' are mentioned here.

`DVIPSFONTS'

Default path to search for all fonts. Overrides all the font-related config file options and other environment variables.

`DVIPSHEADERS'

Default path to search for PostScript header files. Overrides the `H' config file option.

`DVIPSMAKEPK'

Overrides `MakeTeXPK' as the name of the program to invoke to create missing PK fonts. You can change the arguments passed to the `MakeTeXPK' program with the `MAKETEXPK' environment variables; see the Kpathsea documentation.

`DVIPSSIZES'

Last-resort sizes for scaling for unfound fonts. Overrides the `R' definition in config files.

`PRINTER'

This environment variable is read to determine which default printer configuration file to read in. It is the responsibility of the configuration file to send output to the proper print queue, if such functionality is desired.

`TEXCONFIG'

This environment variable sets the directories to search for configuration files, including the system-wide one. Using this single environment variable and the appropriate configuration files, it is possible to set up the program for any environment. (The other path environment variables are thus redundant.)

`TEXPICTS'

Path to search for special illustrations. Overrides the `S' config file option. If not set, `TEXINPU is used.

Other Bells And Whistles

For special effects, if any of the macros `bop-hook', `eop-hook', `start-hook', or `end-hook' are defined in the PostScript `userdict', they will be executed at the beginning of a page, end of a page, start of the document, and end of a document, respectively.

When these macros are executed, the default PostScript coordinate system and origin is in effect. Such macros can be defined in headers added by the `-h' option or the `header=' special, and might be useful for writing, for instance, DRAFT across the entire page, or, with the aid of a shell script, dating the document. These macros are executed outside of the save/restore context of the individual pages, so it is possible for them to accumulate information, but if a document must be divided into sections because of memory constraints, such added information will be lost across section breaks.

The single argument to `bop-hook' is the sequence number of the page in the file; the first page gets zero, the second one, etc. The procedure must leave this number on the stack. None of the other hooks are (currently) given parameters, although this may change in the future.

As an example of what can be done, the following special will write a light DRAFT across each page in the document:

```
\special{!userdict begin /bop-hook{gsave 200 30 translate
65 rotate /Times-Roman findfont 216 scalefont setfont
0 0 moveto 0.7 setgray (DRAFT) show grestore}def end}
```

Note that using `bop-hook' or `eop-hook' in any way that preserves information across pages will break compliance with the Adobe document structuring conventions, so if you use any such tricks, it is recommended that you also use the `-N' option to turn off structured comments.

Several of the above tricks can be used nicely together, and it is not necessary that a `printer configuration file' be used only to set printer defaults. For instance, a `-P' file can be set up to print the date on each page; the particular configuration file will execute a command to put the date into a header file, which is then included with a `h' line in the configuration file. Multiple `-P' options can be used.

MS-DOS

If you have any experience with `dvipsk' under DOS, `karl@cs.umb.edu` would like to hear about it.

Installation

To compile and install Dvipsk:

- Edit the top-level `Makefile.in' if you want to make changes that will have effect across different runs of `configure'; for example, changing the installation directories. Alternatively, override the make variables on the command line when you run make.
- Edit `kpathsea/paths.h.in' to define your local default paths. See section 'System dependencies' in Kpathsea, for more details on changing the paths.
- Run `sh configure' in the top-level directory. This tries to figure out system dependencies and the installation prefix. See section 'The `configure' script' in Kpathsearch library, for options and other information about `configure'.
- @flindex Makefile, editing @flindex c-auto.h, editing If necessary, edit the paths or other definitions in the files `Makefile' and in `dvipsk/c-auto.h' (which `configure' created.)
- Run `make'; e.g., just type `make' in the top-level directory. Barring configuration and compiler bugs, this will compile all the programs. @flindex MACHINES See the `./MACHINES' for possibly helpful system-dependent information.
- @flindex MakeTeXPK, editing Check the paths in `MakeTeXPK', unless you do not want automatic font generation (in which case you need not install `MakeTeXPK' at all). See section Automatic Font Generation. The `MakeTeXPK' in the distribution will overwrite the installed file only if the latter contains the string `original MakeTeXPK --'. Dvipsk, unlike the original dvips, requires `MakeTeXPK' to echo the generated filename (and nothing else) to standard output (standard error can be used for commentary). For more details, or in general if your `MakeTeXPK' fails, see section Unable to Generate Fonts.
- @flindex config.ps, editing Update the device parameters (available memory, resolution, etc.) in `config.ps'. This file is installed as the system-wide configuration file. See section Configuration File Options. The `config.ps' in the distribution will overwrite the installed file only if the latter contains the string `original config.ps --'.
- Install the programs and supporting macros, fonts, and data files with `make install'. If you want to install only the executables, do `make install-exec'; for only the data files, `make install-data'. And if you don't want to install the fonts (perhaps because your directory structure is different from the default), set the Make variable `install_fonts=false'.
- If you want to use more than one device, create configuration files for each and install them in the directory named by the Make variable `configdir'.

See section 'Reporting bugs' in Kpathsea, for the bug reporting address and information.

Diagnosing problems

You've gone through all the trouble of installing dvips, carefully read all the instructions in this manual, and still can't get something to work. This is all too common, and is usually caused by some broken PostScript application out there. The following sections provide some helpful hints if you find yourself in such a situation.

In all cases, you should attempt to find the smallest file that causes the problem. This will not only make debugging easier, it will also reduce the number of possible interactions among different parts of the system.

Debug Options

The `-d' flag to dvips is very useful for helping to track down certain errors. The parameter to this flag is an integer that tells what errors are currently being tracked. To track a certain class of debug messages, simply provide the appropriate number given below; if you wish to track multiple classes, sum the numbers of the classes you wish to track. The classes are:

1
: specials

2
: paths

4
: fonts

8
: pages

16
: headers

32
: font compression

64
: files

128
: memory

No Output At All

If you are not getting any output at all, even from the simplest one-character file (for instance, `\ \bye'), then something is very wrong. Practically any file sent to a PostScript laser printer should generate some output, at the very least a page detailing what error occurred, if any. Talk to your system administrator about downloading a PostScript error handler. (Adobe distributes a good one called `ehandler.ps'.) @flindex

ehandler.ps

It is possible, especially if you are using non-Adobe PostScript, that your PostScript interpreter is broken. Even then it should generate an error message. I've tried to work around as many bugs as possible in common non-Adobe PostScript interpreters, but I'm sure I've missed a few.

If dvips gives any strange error messages, or compilation on your machine generated a lot of warnings, perhaps the dvips program itself is broken. Carefully check the types in `dvips.h' and the declarations in the `Makefile', and try using the debug options to determine where the error occurred.

It is possible your spooler is broken and is misinterpreting the structured comments. Try the `-N' flag to turn off structured comments and see what happens.

Output Too Small or Inverted

If some documents come out inverted or too small, your spooler is not supplying an end of job indicator at the end of each file. (This happens a lot on small machines that don't have spoolers.) You can force dvips to do this with the `-F' flag, but note that this generates files with a binary character (control-D) in them. You can also try using the `-s' flag to enclose the entire job in a save/restore pair.

Error Messages From Printer

If your printer returns error messages, the error message gives very good information on what might be going wrong. One of the most common error messages is `bop undefined'. This is caused by old versions of Transcript and other spoolers that do not properly parse the setup section of the PostScript. To fix this, turn off structured comments with the `-N' option, but make sure you get your spooling software updated.

Another error message is `VM exhausted'. (Some printers indicate this error by locking up; others quietly reset.) This is caused by telling dvips that the printer has more memory than it actually does, and then printing a complicated document. To fix this, try lowering the parameter to `m' in the configuration file; use the debug option to make sure you adjust the correct file.

Other errors may indicate that the graphics you are trying to include don't nest properly in other PostScript documents, or any of a number of other possibilities. Try the output on a QMS PS-810 or other Adobe PostScript printer; it might be a problem with the printer itself.

400 DPI Is Used Instead Of 300 DPI

This common error is caused by not editing the `config.ps' file to reflect the correct resolution for your site. You can use the debug flags (`-d64') to see what files are actually being read.

Long Documents Fail To Print

This is usually caused by incorrectly specifying the amount of memory the printer has in `config.ps'; see the description above.

Including Graphics Fails

The reasons why graphics inclusions fail are too numerous to mention. The most common problem is an incorrect bounding box; read the section on bounding boxes and check your PostScript file. Complain very loudly to whoever wrote the software that generated the file if the bounding box is indeed incorrect.

Another possible problem is that the figure you are trying to include does not nest properly; there are certain rules PostScript applications should follow when generating files to be included. The dvips program includes work-arounds for such errors in Adobe Illustrator and other programs, but there are certainly applications that haven't been tested.

One possible thing to try is the `-K' flag, to strip the comments from an included figure. This might be necessary if the PostScript spooling software does not read the structuring comments correctly. Use of this flag will break graphics from some applications, though, since some applications read the PostScript file from the input stream looking for a particular comment.

Any application which generates graphics output containing raw binary (not ASCII hex) will probably fail with dvips.

Unable to Find Font Files

If dvips complains that it cannot find certain font files, it is possible that the paths haven't been set up correctly for your system. Use the debug flags to determine precisely what fonts are being looked for and make sure these match where the fonts are located on your system.

Unable to Generate Fonts

This happens a lot if either `MakeTeXPK' hasn't been properly edited and installed, or if the local installation of METAFONT isn't correct.

`MakeTeXPK' must echo the generated filename (and nothing else) to standard output. See section Automatic Font Generation.

If METAFONT isn't found when `MakeTeXPK' is running, then you need to install it. Retrieve it from, e.g., `ftp.cs.umb.edu' in `pub/tex/web2c.tar.gz' and `pub/tex/web.tar.gz'.

If METAFONT runs but generates fonts that are too large (and prints out the name of each character as well as just a character number), then your METAFONT base file probably hasn't been made properly. To make a proper `plain.base', assuming the local mode definitions are contained in `modes.mf' type the following command (assuming UNIX):

```
inimf "plain; input modes; dump"
```

Then copy the `plain.base' file from the current directory to where the base files are stored on your system.

By the way, the macros defined in `cmbase' will break fonts that do not use `cmbase'; such fonts include the LaTeX fonts. Loading the `cmbase' macros when they are needed is done automatically and takes less than

a second--an insignificant fraction of the total run time of METAFONT for a font, especially when the possibility of generating incorrect fonts is taken into account. If you create the LaTeX font `circle10', for instance, with the `cmbase' macros loaded, the characters will have incorrect widths.

Using Color with dvips

This new feature of `dvips' is somewhat experimental so your experiences and comments are welcome. Initially added by Jim Hafner, IBM Research, `hafner@almaden.ibm.com', the color support has gone through many changes by Tomas Rokicki. Besides the changes to the source code itself, there are additional TeX macro files: `colordvi.tex' and `blackdvi.tex'. There are also `.sty' versions of these files that can be used with LaTeX and other similar macro packages. This feature adds one-pass multi-color printing of TeX documents on any color PostScript device.

In this section we describe the use of color from the document preparer's point of view and then add some simple instructions on installation for the system administrator.

Macro Files

All the color macro commands are defined in `colordvi.tex' (or `colordvi.sty'). To access these macros simply add to the top of your TeX file the command

```
\input colordvi
```

or, if your document uses style files like LaTeX, add the `colordvi' style option as in

```
\documentstyle[12pt,colordvi]{article}
```

There are basically two kinds of color macros, ones for local color changes to, say, a few words or even one symbol and one for global color changes. Note that all the color names use a mixed case scheme. There are 68 predefined colors, with names taken primarily from the Crayola crayon box of 64 colors, and one pair of macros for the user to set his own color pattern. More on this extra feature later. You can browse the file `colordvi.tex' for a list of the predefined colors. The comments in this file also show a rough correspondence between the crayon names and PANTONEs.

A local color command is in the form

```
\ColorName{this will print in color}
```

Here `ColorName' is the name of a predefined color. As this example shows, this type of command takes one argument which is the text that is to print in the selected color. This can be used for nested color changes since it restores the original color state when it completes. For example, suppose you were writing in green and want to switch temporarily to red, then blue, back to red and restore green. Here is one way that you can do this:

```
This text is green but here we are \Red{switching to red,
```


```
\Blue{nesting blue}, recovering the red} and back to
original green.
```

In principle there is no limit to the nesting level, but it is not advisable to nest too deep lest you loose track of the color history. The global color command has the form

```
\textColorName
```

This macro takes no arguments and immediately changes the default color from that point on to the specified color. This of course can be overriden globally by another such command or locally by local color commands. For example, expanding on the example above, we might have

```
\textGreen
This text is green but here we are \Red{switching to red,
\Blue{nesting blue}, recovering the red} and back to
original green.
\textCyan
The text from here on will be cyan unless
\Yellow{locally changed to yellow}. Now we are back to cyan.
```

The color commands will even work in math mode and across math mode boundaries. This means that if you have a color before going into math mode, the mathematics will be set in that color as well. More importantly however, in alignment environments like `\halign', `tabular' or `eqnarray', local color commands cannot extend beyond the alignment characters. Because local color commands respect only some environment and deliminator changes besides their own, care must be taken in setting their scope. It is best not to have then stretch too far. At the present time there are no macros for color environments in LaTeX which might have a larger range. This is primarily to keep the TeX and LaTeX use compatible.

User Definable Colors

There are two ways for the user to specify colors not already defined. For local changes, there is the command `\Color' which takes two arguments. The first argument is a quadruple of numbers between zero and one and specifies the intensity of cyan, magenta, yellow and black (CMYK) in that order. The second argument is the text that should appear in the given color. For example, suppose you want the words "this color is pretty" to appear in a color which is 50% cyan, 85% magenta, 40% yellow and 20% black. You would use the command

```
\Color{.5 .85 .4 .2}{this color is pretty}
```

For global color changes, there is a command `\textColor' which takes one argument, the CMYK quadruple of relative color intensities. For example, if you want the default color to be as above, then the command

```
\textColor{.5 .85 .4 .2}
The text from now on will be this pretty color
```

will do the trick. Making a global color change in the midst of a nested local colors is highly discouraged. Consequently, dvips will give you warning message and do its best to recover by discarding the current

color history.

Subtleties in Using Color

These color macros are defined by use of specialized `\special' keywords. As such, they are put in the `.dvi' file only as explicit message strings to the driver. The (unpleasant) result is that certain unprotected regions of the text can have unwanted color side effects. For example, if a color region is split by TeX across a page boundary, then the footers of the current page (e.g., the page number) and the headers of the next page can inherit that color. To avoid this effect globally, users should make sure that these special regions of the text are defined with their own local color commands. For example in TeX, to protect the header and footer, use

```
\headline{\Black{My Header}}
\footline{\Black{\hss\tenrm\folio\hss}}
```

This warning also applies to figures and other insertions, so be careful!

Of course, in LaTeX, this is much more difficult to do because of the complexity of the macros that control these regions. This is unfortunate, but is somehow inevitable because TeX and LaTeX were not written with color in mind.

Even when writing your own macros, much care must be taken. The color macros that `colorize' a portion of the text work by prefixing the text with a special command to turn the color on and postfixing it with a different special command to restore the original color. It is often useful to ensure that TeX is in horizontal mode before the first special command is issued; this can be done by prefixing the color command with `\leavevmode'.

Printing in Black/White, after Colorizing

If you have a TeX or LaTeX document written with color macros and you want to print it in black and white there are two options. On all (good) PostScript devices, printing a color file will print in corresponding grey-levels. This is useful since in this way you can get a rough idea of where the colors are changing without using expensive color printing devices. The second option is to replace the call to input `colordvi.tex' with `blackdvi.tex' (and similarly for the `.sty' files). So in the above example, replacing the word `colordvi' with `blackdvi' suffices. This file defines the color macros as no-ops, and so will produce normal black/white printing. By this simple mechanism, the user can switch to all black/white printing without having to ferret out the color commands. Also, some device drivers, particularly non-PostScript ones like screen previewers, will simply ignore the color commands and so print in normal black/white. Hopefully, in the future screen previewers for color displays will be compatible with some form of color support.

Configuring dvips for Color Devices

To configure dvips for a particular color device you need to fine tune the color parameters to match your devices color rendition. To do this, you will need a PANTONE chart for your device. The header file `color.lpro' shows a (rough) correspondence between the Crayola crayon names and the PANTONE numbers and also defines default CMYK values for each of the colors. Note that these colors must be defined in CMYK terms and not RGB as dvips outputs PostScript color commands in CMYK. This header file also defines (if they are not known to the interpreter) the PostScript commands `setcmykcolor' and `currentcmykcolor' in terms of a RGB equivalent so if your device only understands RGB, there should be no problem.

The parameters set in this file were determined by comparing the PANTONE chart of a Tektronics PHASER printer with the actual Crayola Crayons. Because these were defined for a particular device, the actual color rendition on your device may be very different. There are two ways to adjust this. One is to use the PANTONE chart for your device to rewrite `color.lpro' prior to compilation and installation. A better alternative, which supports multiple devices, is to add a header file option in the configuration file for each device that defines, in `userdict', the color parameters for those colors that need redefining.

For example, if you need to change the parameters defining `Goldenrod' (approximately PANTONE 109) for your device `mycolordev', do the following. In the PANTONE chart for your device, find the CMYK values for PANTONE 109. Let's say they are `{\ 0 0.10 0.75 0.03 }'. Then create a header file named `mycolordev.pro' with the commands

```
userdict begin
/Goldenrod { 0 0.10 0.75 0.03 setcmykcolor} bind def
```

Finally, in `config.mycolordev' add the line

```
h mycolordev.pro
```

This will then define `Goldenrod' in your device's CMYK values in `userdict' which is checked before defining it in `TeXdict' by `color.pro'.

This mechanism, together with additions to `colordvi.tex' and `blackdvi.tex' (and the `.sty' files), can also be used to predefine other colors for your users.

Color Support Details

To support color, dvips recognizes a certain set of specials. These specials all start with the keyword `color' or the keyword `background'.

We will describe `background' first, since it is the simplest. The `background' keyword must be followed by a color specification. That color specification is used as a fill color for the background. The last `background' special on a page is the one that gets issued, and it gets issued at the very beginning of the page, before any text or specials are sent. (This is possible because the prescan phase of dvips notices all of the color specials so that the appropriate information can be written out during the second phase.)

Ahh, but what is a color specification? It is one of three things. First, it might be a PostScript procedure defined in a PostScript header file. The `color.pro' file defines 64 of these, including `Maroon'. This PostScript procedure must set the current color to be some value; in this case, `Maroon' is defined as `0.87 0.68 0.32 setcmykcolor'.

The second possibility is the name of a color model (initially, one of `rgb', `hsb', `cmyk', or `gray') followed by the appropriate number of parameters. When dvips encounters such a macro, it sends out the parameters first, followed by the string created by prefixing `TeXcolor' to the color model. Thus, the color specification `rgb 0.3 0.4 0.5' would generate the PostScript code `0.3 0.4 0.5 TeXrgbcolor'. Note that the case of zero arguments is disallowed, as that is handled by the single keyword case above (where no changes to the name are made before it is sent to the PostScript file.)

The third and final type of color specification is a double quote followed by any sequence of PostScript. The double quote is stripped from the output. For instance, the color specification `"AggiePattern setpattern' will set the `color' to the Aggie logo pattern (assuming such exists.)

So those are the three types of color specifications. The same type of specifications are used by both the `background' special and the `color' special. The `color' special itself has three forms. The first is just `color' followed by a color specification. In this case, the current global color is set to that color; the color stack must be empty when such a command is executed.

The second form is `color push' followed by a color specification. This saves the current color on the color stack and sets the color to be that given by the color specification. This is the most common way to set a color.

The final version of the `color' special is just `color pop', with no color specification; this says to pop the color last pushed on the color stack from the color stack and set the current color to be that color.

The `dvips' program correctly handles these color specials across pages, even when the pages are repeated or reversed.

These color specials can be used for things such as patterns or screens as well as simple colors. However, note that in the PostScript, only one `color specification' can be active at a time. For instance, at the beginning of a page, only the bottommost entry on the color stack is sent; also, when a color is `popped', all that is done is that the color specification from the previous stack entry is sent. No `gsave' or `grestore' is used. This means that you cannot easily mix usage of the `color' specials for screens and colors, just one or the other. This may be addressed in the future by adding support for different `categories' of color-like state.

Index

\

a

b

c

d

e

f

g

h

i

k

l

m

n

o

p

q

r

s

t

u

v

GNU Emacs Lisp Reference Manual

GNU Emacs Version 19

for Unix Users

Edition 2.0, May 1993

by Bil Lewis, Dan LaLiberte, Richard Stallman and the GNU Manual Group

Go to the next section.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free

use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that wor not derived from the Program, and can be reasonably considered independent and separate wo in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
    1. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
    2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
    3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

    The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

    If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so

long as such parties remain in full compliance.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number

of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the

"copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type `show c'
for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program `Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License

instead of this License.

Go to the next section.

Go to the previous, next section.

Introduction

Most of the GNU Emacs text editor is written in the programming language called Emacs Lisp. You can write new code in Emacs Lisp and install it as an extension to the editor. However, Emacs Lisp is more than a mere "extension language"; it is a full computer programming language in its own right. You can use it as you would any other programming language.

Because Emacs Lisp is designed for use in an editor, it has special features for scanning and parsing text as well as features for handling files, buffers, displays, subprocesses, and so on. Emacs Lisp is closely integrated with the editing facilities; thus, editing commands are functions that can also conveniently be called from Lisp programs, and parameters for customization are ordinary Lisp variables.

This manual describes Emacs Lisp, presuming considerable familiarity with the use of Emacs for editing. (See The GNU Emacs Manual, for this basic information.) Generally speaking, the earlier chapters describe features of Emacs Lisp that have counterparts in many programming languages, and later chapters describe features that are peculiar to Emacs Lisp or relate specifically to editing.

This is edition 2.0.

Caveats

This manual has gone through numerous drafts. It is nearly complete but not flawless. There are a few sections which are not included, either because we consider them secondary (such as most of the individual modes) or because they are yet to be written.

Because we are not able to deal with them completely, we have left out several parts intentionally. This includes most references to VMS and all information relating Sunview. (The Free Software Foundation expends no effort on support for Sunview, since we believe users should use the free X window system rather than proprietary window systems.)

The manual should be fully correct in what it does cover, and it is therefore open to criticism on anything it says--from specific examples and descriptive text, to the ordering of chapters and sections. If something is confusing, or you find that you have to look at the sources or experiment to learn something not covered in the manual, then perhaps the manual should be fixed. Please let us know.

As you use the manual, we ask that you mark pages with corrections so you can later look them up and send them in. If you think of a simple, real life example for a function or group of functions, please make an effort to write it up and send it in. Please reference any comments to the chapter name, section name, and function name, as appropriate, since page numbers and chapter and section numbers will change. Also state the number of the edition which you are criticizing.

Please mail comments and corrections to

```
bug-lisp-manual@prep.ai.mit.edu
```


```
--Bil Lewis, Dan LaLiberte, Richard Stallman
```

Lisp History

Lisp (LISt Processing language) was first developed in the late 1950s at the Massachusetts Institute of Technology for research in artificial intelligence. The great power of the Lisp language makes it superior for other purposes as well, such as writing editing commands.

Dozens of Lisp implementations have been built over the years, each with its own idiosyncrasies. Many of them were inspired by Maclisp, which was written in the 1960's at MIT's Project MAC. Eventually the implementors of the descendents of Maclisp came together and developed a standard for Lisp systems, called Common Lisp.

GNU Emacs Lisp is largely inspired by Maclisp, and a little by Common Lisp. If you know Common Lisp, you will notice many similarities. However, many of the features of Common Lisp have been omitted or simplified in order to reduce the memory requirements of GNU Emacs. Sometimes the simplifications are so drastic that a Common Lisp user might be very confused. We will occasionally point out how GNU Emacs Lisp differs from Common Lisp. If you don't know Common Lisp, don't worry about it; this manual is self-contained.

Conventions

This section explains the notational conventions that are used in this manual. You may want to skip this section and refer back to it later.

Some Terms

Throughout this manual, the phrases "the Lisp reader" and "the Lisp printer" are used to refer to those routines in Lisp that convert textual representations of Lisp objects into actual objects, and vice versa. See section Printed Representation and Read Syntax, for more details. You, the person reading this manual, are thought of as "the programmer" and are addressed as "you". "The user" is the person who uses Lisp programs including those you write.

Examples of Lisp code appear in this font or form: `(list 1 2 3)`. Names that represent arguments or metasyntactic variables appear in this font or form: first-number.

`nil` and `t`

In Lisp, the symbol `nil` is overloaded with three meanings: it is a symbol with the name `nil'; it is the logical truth value false; and it is the empty list--the list of zero elements. When used as a variable, `nil` always has the value `nil`.

As far as the Lisp reader is concerned, `()' and `nil' are identical: they stand for the same object, the symbol `nil`. The different ways of writing the symbol are intended entirely for human readers. After the

Lisp reader has read either `()' or `nil', there is no way to determine which representation was actually written by the programmer.

In this manual, we use `()` when we wish to emphasize that it means the empty list, and we use `nil` when we wish to emphasize that it means the truth value false. That is a good convention to use in Lisp programs also.

```
(cons 'foo ())                  ; Emphasize the empty list
(not nil)                       ; Emphasize the truth value false
```

In contexts where a truth value is expected, any non-`nil` value is considered to be true. However, `t` is the preferred way to represent the truth value true. When you need to choose a value which represents true, and there is no other basis for choosing, use `t`. The symbol `t` always has value `t`.

In Emacs Lisp, `nil` and `t` are special symbols that always evaluate to themselves. This is so that you do not need to quote them to use them as constants in a program. An attempt to change their values results in a `setting-constant` error. See section Accessing Variable Values.

Evaluation Notation

A Lisp expression that you can evaluate is called a form. Evaluating a form always produces a result, which is a Lisp object. In the examples in this manual, this is indicated with `=>':

```
(car '(1 2))
     => 1
```

You can read this as "`(car '(1 2))` evaluates to 1".

When a form is a macro call, it expands into a new form for Lisp to evaluate. We show the result of the expansion with `==>'. We may or may not show the actual result of the evaluation of the expanded form.

```
(third '(a b c))
     ==> (car (cdr (cdr '(a b c))))
     => c
```

Sometimes to help describe one form we show another form which produces identical results. The exact equivalence of two forms is indicated with `=='.

```
(make-sparse-keymap) == (list 'keymap)
```

Printing Notation

Many of the examples in this manual print text when they are evaluated. If you execute the code from an example in a Lisp Interaction buffer (such as the buffer `*scratch*'), the printed text is inserted into the buffer. If the example is executed by other means (such as by evaluating the function `eval-region`), the text printed is usually displayed in the echo area. You should be aware that text displayed in the echo

area is truncated to a single line.

In examples that print text, the printed text is indicated with `-|', irrespective of how the form is executed. The value returned by evaluating the form (here `bar') follows on a separate line.

```
(progn (print 'foo) (print 'bar))
     -| foo
     -| bar
     => bar
```

Error Messages

Some examples cause errors to be signaled. In them, the error message (which always appears in the echo area) is shown on a line starting with `error-->'. Note that `error-->' itself does not appear in the echo area.

```
(+ 23 'x)
error--> Wrong type argument: integer-or-marker-p, x
```

Buffer Text Notation

Some examples show modifications to text in a buffer, with "before" and "after" versions of the text. In such cases, the entire contents of the buffer in question are included between two lines of dashes containing the buffer name. In addition, the location of point is shown as `-!-'. (The symbol for point, of course, is not part of the text in the buffer; it indicates the place *between* two characters where point is located.)

```
---------- Buffer: foo ----------
This is the -!-contents of foo.
---------- Buffer: foo ----------

(insert "changed ")
     => nil
---------- Buffer: foo ----------
This is the changed -!-contents of foo.
---------- Buffer: foo ----------
```

Format of Descriptions

Functions, variables, macros, commands, user options, and special forms are described in this manual in a uniform format. The first line of a description contains the name of the item followed by its arguments, if any. The category--function, variable, or whatever--is printed next to the right margin. The description follows on succeeding lines, sometimes with examples.

A Sample Function Description

In a function description, the name of the function being described appears first. It is followed on the same line by a list of parameters. The names used for the parameters are also used in the body of the description.

The appearance of the keyword `&optional` in the parameter list indicates that the arguments for subsequent parameters may be omitted (omitted parameters default to `nil`). Do not write `&optional` when you call the function.

The keyword `&rest` (which will always be followed by a single parameter) indicates that any number of arguments can follow. The value of the single following parameter will be a list of all these arguments. Do not write `&rest` when you call the function.

Here is a description of an imaginary function `foo`:

Function: **foo** *integer1 &optional integer2 &rest integers*

The function `foo` subtracts integer1 from integer2, then adds all the rest of the arguments to the result. If integer2 is not supplied, then the number 19 is used by default.

```
(foo 1 5 3 9)
     => 16
(foo 5)
     => 14
```

More generally,

```
(foo w x y...)
==
(+ (- x w) y...)
```

Any parameter whose name contains the name of a type (e.g., integer, integer1 or buffer) is expected to be of that type. A plural of a type (such as buffers) often means a list of objects of that type. Parameters named object may be of any type. (See section Lisp Data Types, for a list of Emacs object types.) Parameters with other sorts of names (e.g., new-file) are discussed specifically in the description of the function. In some sections, features common to parameters of several functions are described at the beginning.

See section Lambda Expressions, for a more complete description of optional and rest arguments.

Command, macro, and special form descriptions have the same format, but the word `Function' is replaced by `Command', `Macro', or `Special Form', respectively. Commands are simply functions that may be called interactively; macros process their arguments differently from functions (the arguments are not evaluated), but are presented the same way.

Special form descriptions use a more complex notation to specify optional and repeated parameters because they can break the argument list down into separate arguments in more complicated ways. `[optional-arg]' means that optional-arg is optional and `repeated-args...' stands for zero or more arguments. Parentheses are used when several arguments are grouped into additional levels of list

structure. Here is an example:

Special Form: **count-loop** *(var [from to [inc]]) body...*

This imaginary special form implements a loop that executes the body forms and then increments the variable var on each iteration. On the first iteration, the variable has the value from; on subsequent iterations, it is incremented by 1 (or by inc if that is given). The loop exits before executing body if var equals to. Here is an example:

```
(count-loop (i 0 10)
  (prin1 i) (princ " ")
  (prin1 (aref vector i)) (terpri))
```

If from and to are omitted, then var is bound to `nil` before the loop begins, and the loop exits if var is non-`nil` at the beginning of an iteration. Here is an example:

```
(count-loop (done)
  (if (pending)
      (fixit)
    (setq done t)))
```

In this special form, the arguments from and to are optional, but must both be present or both absent. If they are present, inc may optionally be specified as well. These arguments are grouped with the argument var into a list, to distinguish them from body, which includes all remaining elements of the form.

A Sample Variable Description

A variable is a name that can hold a value. Although any variable can be set by the user, certain variables that exist specifically so that users can change them are called user options. Ordinary variables and user options are described using a format like that for functions except that there are no arguments.

Here is a description of the imaginary `electric-future-map` variable.

Variable: **electric-future-map**

The value of this variable is a full keymap used by electric command future mode. The functions in this map will allow you to edit commands you have not yet thought about executing.

User option descriptions have the same format, but `Variable' is replaced by `User Option'.

Acknowledgements

This manual was written by Robert Krawitz, Bil Lewis, Dan LaLiberte, Richard M. Stallman and Chris Welty, the volunteers of the GNU manual group, in an effort extending over several years. Robert J. Chassell helped to review and edit the manual, with the support of the Defense Advanced Research Projects Agency, ARPA Order 6082, arranged by Warren A. Hunt, Jr. of Computational Logic, Inc.

Corrections were supplied by Karl Berry, Jim Blandy, Bard Bloom, David Boyes, Alan Carroll, David A.

Duff, Beverly Erlebacher, David Eckelkamp, Eirik Fuller, Eric Hanchrow, George Hartzell, Nathan Hess, Dan Jacobson, Jak Kirman, Bob Knighten, Frederick M. Korz, Joe Lammens, K. Richard Magg Brian Marick, Roland McGrath, Skip Montanaro, John Gardiner Myers, Arnold D. Robbins, Raul Rockwell, Shinichirou Sugou, Kimmo Suominen, Edward Tharp, Bill Trost, Jean White, Matthew Wilding, Carl Witty, Dale Worley, Rusty Wright, and David D. Zuhn.

Go to the previous, next section.

Go to the previous, next section.

Lisp Data Types

A Lisp object is a piece of data used and manipulated by Lisp programs. For our purposes, a type or data type is a set of possible objects.

Every object belongs to at least one type. Objects of the same type have similar structures and may usually be used in the same contexts. Types can overlap, and objects can belong to two or more types. Consequently, we can ask whether an object belongs to a particular type, but not for "the" type of an object.

A few fundamental object types are built into Emacs. These, from which all other types are constructed, are called primitive types. Each object belongs to one and only one primitive type. These types include integer, float, cons, symbol, string, vector, subr, byte-code function, and several special types, such as buffer, that are related to editing. (See section Editing Types.)

Each primitive type has a corresponding Lisp function that checks whether an object is a member of that type.

Note that Lisp is unlike many other languages in that Lisp objects are self-typing: the primitive type of the object is implicit in the object itself. For example, if an object is a vector, it cannot be treated as a number because Lisp knows it is a vector, not a number.

In most languages, the programmer must declare the data type of each variable, and the type is known by the compiler but not represented in the data. Such type declarations do not exist in Emacs Lisp. A Lisp variable can have any type of value, and remembers the type of any value you store in it.

This chapter describes the purpose, printed representation, and read syntax of each of the standard types in GNU Emacs Lisp. Details on how to use these types can be found in later chapters.

Printed Representation and Read Syntax

The printed representation of an object is the format of the output generated by the Lisp printer (the function `print`) for that object. The read syntax of an object is the format of the input accepted by the Lisp reader (the function `read`) for that object. Most objects have more than one possible read syntax. Some types of object have no read syntax; except for these cases, the printed representation of an object is also a read syntax for it.

In other languages, an expression is text; it has no other form. In Lisp, an expression is primarily a Lisp object and only secondarily the text that is the object's read syntax. Often there is no need to emphasize this distinction, but you must keep it in the back of your mind, or you will occasionally be very confused.

Every type has a printed representation. Some types have no read syntax, since it may not make sense to enter objects of these types directly in a Lisp program. For example, the buffer type does not have a read syntax. Objects of these types are printed in hash notation: the characters `#<' followed by a descriptive

string (typically the type name followed by the name of the object), and closed with a matching `>'. I notation cannot be read at all, so the Lisp reader signals the error `invalid-read-syntax` whene a `#<' is encountered.

```
(current-buffer)
     => #<buffer objects.texi>
```

When you evaluate an expression interactively, the Lisp interpreter first reads the textual representation of it, producing a Lisp object, and then evaluates that object (see section Evaluation). However, evaluation and reading are separate activities. Reading returns the Lisp object represented by the text that is read; the object may or may not be evaluated later. See section Input Functions, for a description of `read`, the basic function for reading objects.

Comments

A comment is text that is written in a program only for the sake of humans that read the program, and that has no effect on the meaning of the program. In Lisp, a comment starts with a semicolon (`;') if it is not within a string or character constant, and continues to the end of line. Comments are discarded by the Lisp reader, and do not become part of the Lisp objects which represent the program within the Lisp system.

See section Tips on Writing Comments, for conventions for formatting comments.

Programming Types

There are two general categories of types in Emacs Lisp: those having to do with Lisp programming, and those having to do with editing. The former are provided in many Lisp implementations, in one form or another. The latter are unique to Emacs Lisp.

Integer Type

Integers are the only kind of number in GNU Emacs Lisp, version 18. The range of values for integers is -8388608 to 8388607 (24 bits; i.e., to on most machines, but is 25 or 26 bits on some systems. It is important to note that the Emacs Lisp arithmetic functions do not check for overflow. Thus `(1+ 8388607)` is -8388608 on 24-bit implementations.

The read syntax for numbers is a sequence of (base ten) digits with an optional sign. The printed representation produced by the Lisp interpreter never has a leading `+'.

```
-1              ; The integer -1.
1               ; The integer 1.
+1              ; Also the integer 1.
16777217        ; Also the integer 1!
                ;   (on a 24-bit or 25-bit implementation)
```

See section Numbers, for more information.

Floating Point Type

Emacs version 19 supports floating point numbers, if compiled with the macro `LISP_FLOAT_TYPE` defined. The precise range of floating point numbers is machine-specific.

The printed representation for floating point numbers requires either a decimal point (with at least one digit following), an exponent, or both. For example, `1500.0', `15e2', `15.0e2', `1.5e3', and `.15e4' are five ways of writing a floating point number whose value is 1500. They are all equivalent.

See section Numbers, for more information.

Character Type

A character in Emacs Lisp is nothing more than an integer. In other words, characters are represented by their character codes. For example, the character A is represented as the integer 65.

Individual characters are not often used in programs. It is far more common to work with *strings*, which are sequences composed of characters. See section String Type.

Characters in strings, buffers, and files are currently limited to the range of 0 to 255. If an arbitrary integer is used as a character for those purposes, only the lower eight bits are significant. Characters that represent keyboard input have a much wider range.

Since characters are really integers, the printed representation of a character is a decimal number. This is also a possible read syntax for a character, but writing characters that way in Lisp programs is a very bad idea. You should *always* use the special read syntax formats that Emacs Lisp provides for characters. These syntax formats start with a question mark.

The usual read syntax for alphanumeric characters is a question mark followed by the character; thus, `?A' for the character A, `?B' for the character B, and `?a' for the character a.

For example:

```
?Q => 81

?q => 113
```

You can use the same syntax for punctuation characters, but it is often a good idea to add a `\' to prevent Lisp mode from getting confused. For example, `?\ ' is the way to write the space character. If the character is `\', you *must* use a second `\' to quote it: `?\\'.

You can express the characters control-g, backspace, tab, newline, vertical tab, formfeed, return, and escape as `?\a', `?\b', `?\t', `?\n', `?\v', `?\f', `?\r', `?\e', respectively. Those values are 7, 8, 9, 10, 11, 12, 13, and 27 in decimal. Thus,

```
?\a => 7                 ; C-g
```


```
?\b => 8                ; backspace, BS, C-h
?\t => 9                ; tab, TAB, C-i
?\n => 10               ; newline, LFD, C-j
?\v => 11               ; vertical tab, C-k
?\f => 12               ; formfeed character, C-l
?\r => 13               ; carriage return, RET, C-m
?\e => 27               ; escape character, ESC, C-[
?\\ => 92               ; backslash character, \
```

These sequences which start with backslash are also known as escape sequences, because backslash plays the role of an escape character, but they have nothing to do with the character ESC.

Control characters may be represented using yet another read syntax. This consists of a question mark followed by a backslash, caret, and the corresponding non-control character, in either upper or lower case. For example, either `?\^I' or `?\^i' may be used as the read syntax for the character C-i, the character whose value is 9.

Instead of the `^', you can use `C-'; thus, `?\C-i' is equivalent to `?\^I' and to `?\^i':

```
?\^I => 9

?\C-I => 9
```

For use in strings and buffers, you are limited to the control characters that exist in ASCII, but for keyboard input purposes, you can turn any character into a control character with `C-'. The character codes for these characters include the 2**22 bit as well as the code for the non-control character. Ordinary terminals have no way of generating non-ASCII control characters, but you can generate them straightforwardly using an X terminal.

The DEL key can be considered and written as Control-?:

```
?\^? => 127

?\C-? => 127
```

When you represent control characters to be found in files or strings, we recommend the `^' syntax; but when you refer to keyboard input, we prefer the `C-' syntax. This does not affect the meaning of the program, but may guide the understanding of people who read it.

A meta character is a character typed with the META key. The integer that represents such a character has the 2**23 bit set (which on most machines makes it a negative number). We use high bits for this and other modifiers to make possible a wide range of basic character codes.

In a string, the 2**7 bit indicates a meta character, so the meta characters that can fit in a string have codes in the range from 128 to 255, and are the meta versions of the ordinary ASCII characters. (In Emacs versions 18 and older, this convention was used for characters outside of strings as well.)

The read syntax for meta characters uses `\M-'. For example, `?\M-A' stands for M-A. You can use `\M-'

together with octal codes, `\C-', or any other syntax for a character. Thus, you can write M-A as `?\M-A', or as `?\M-\101'. Likewise, you can write C-M-b as `?\M-\C-b', `?\C-\M-b', or `?\M-\002'.

The shift modifier is used in indicating the case of a character in special circumstances. The case of an ordinary letter is indicated by its character code as part of ASCII, but ASCII has no way to represent whether a control character is upper case or lower case. Emacs uses the 2**21 bit to indicate that the shift key was used for typing a control character. This distinction is possible only when you use X terminals or other special terminals; ordinary terminals do not indicate the distinction to the computer in any way.

The X Window system defines three other modifier bits that can be set in a character: hyper, super and alt. The syntaxes for these bits are `\H-', `\s-' and `\A-'. Thus, `?\H-\M-\A-x' represents Alt-Hyper-Meta-x. Numerically, the bit values are 2**18 for alt, 2**19 for super and 2**20 for hyper.

Finally, the most general read syntax consists of a question mark followed by a backslash and the character code in octal (up to three octal digits); thus, `?\101' for the character A, `?\001' for the character C-a, and `?\002` for the character C-b. Although this syntax can represent any ASCII character, it is preferred only when the precise octal value is more important than the ASCII representation.

```
?\012 => 10          ?\n => 10          ?\C-j => 10

?\101 => 65          ?A => 65
```

A backslash is allowed, and harmless, preceding any character without a special escape meaning; thus, `?\A' is equivalent to `?A'. There is no reason to use a backslash before most such characters. However, any of the characters `()\|;'`"#.,' should be preceded by a backslash to avoid confusing the Emacs commands for editing Lisp code. Whitespace characters such as space, tab, newline and formfeed should also be preceded by a backslash. However, it is cleaner to use one of the easily readable escape sequences, such as `\t', instead of an actual control character such as a tab.

Sequence Types

A sequence is a Lisp object that represents an ordered set of elements. There are two kinds of sequence in Emacs Lisp, lists and arrays. Thus, an object of type list or of type array is also considered a sequence.

Arrays are further subdivided into strings and vectors. Vectors can hold elements of any type, but string elements must be characters in the range from 0 to 255. However, the characters in a string can have text properties; vectors do not support text properties even when their elements happen to be characters.

Lists, strings and vectors are different, but they have important similarities. For example, all have a length l, and all have elements which can be indexed from zero to l minus one. Also, several functions, called sequence functions, accept any kind of sequence. For example, the function `elt` can be used to extract an element of a sequence, given its index. See section Sequences, Arrays, and Vectors.

It is impossible to read the same sequence twice, in the sense of `eq` (see section Equality Predicates), since sequences are always created anew upon reading. There is one exception: the empty list `()` always stands for the same object, `nil`.

List Type

A list is a series of cons cells, linked together. A cons cell is an object comprising two pointers named the CAR and the CDR. Each of them can point to any Lisp object, but when the cons cell is part of a list, the CDR points either to another cons cell or to the empty list. See section Lists, for functions that work on lists.

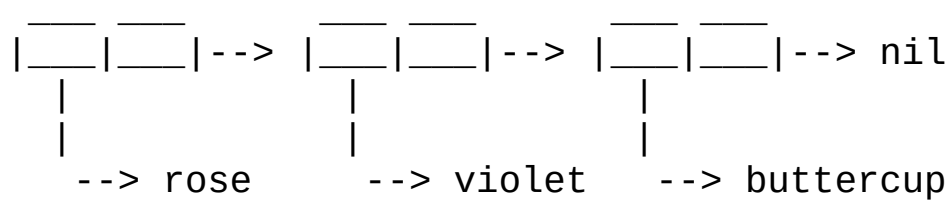
The names CAR and CDR have only historical meaning now. The original Lisp implementation ran on an IBM 704 computer which divided words into two parts, called the "address" part and the "decrement"; CAR was an instruction to extract the contents of the address part of a register, and CDR an instruction to extract the contents of the decrement. By contrast, "cons cells" are named for the function `cons` that creates them, which in turn is named for its purpose, the construction of cells.

Because cons cells are so central to Lisp, we also have a word for "an object which is not a cons cell". These objects are called atoms.

The read syntax and printed representation for lists are identical, and consist of a left parenthesis, an arbitrary number of elements, and a right parenthesis.

Upon reading, any object inside the parentheses is made into an element of the list. That is, a cons cell is made for each element. The CAR of the cons cell points to the element, and its CDR points to the next cons cell which holds the next element in the list. The CDR of the last cons cell is set to point to `nil`.

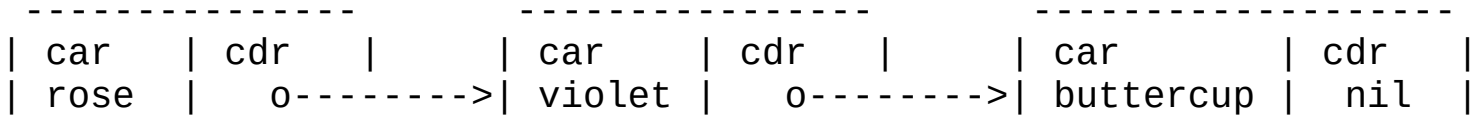
A list can be illustrated by a diagram in which the cons cells are shown as pairs of boxes. (The Lisp reader cannot read such an illustration; unlike the textual notation, which can be understood both humans and computers, the box illustrations can only be understood by humans.) The following represents the three-element list `(rose violet buttercup)`:

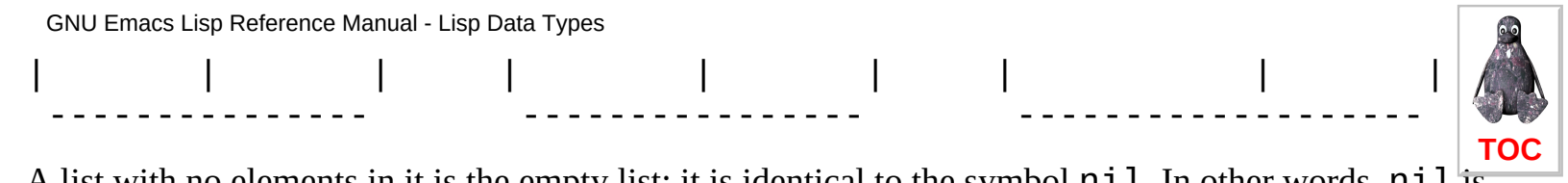


In the diagram, each box represents a slot that can refer to any Lisp object. Each pair of boxes represents a cons cell. Each arrow is a reference to a Lisp object, either an atom or another cons cell.

In this example, the first box, the CAR of the first cons cell, refers to or "contains" `rose` (a symbol). The second box, the CDR of the first cons cell, refers to the next pair of boxes, the second cons cell. The CAR of the second cons cell refers to `violet` and the CDR refers to the third cons cell. The CDR of the third (and last) cons cell refers to `nil`.

Here is another diagram of the same list, `(rose violet buttercup)`, sketched in a different manner:



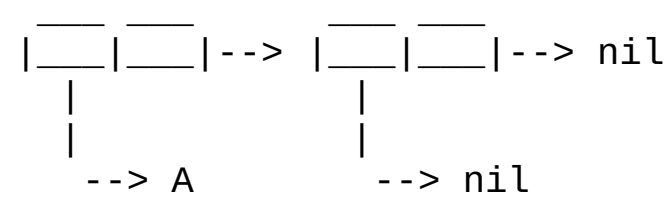


A list with no elements in it is the empty list; it is identical to the symbol `nil`. In other words, `nil` is both a symbol and a list.

Here are examples of lists written in Lisp syntax:

```
(A 2 "A")            ; A list of three elements.
()                   ; A list of no elements (the empty list).
nil                  ; A list of no elements (the empty list).
("A ()")             ; A list of one element: the string "A ()".
(A ())               ; A list of two elements: A and the empty list.
(A nil)              ; Equivalent to the previous.
((A B C))            ; A list of one element
                     ;   (which is a list of three elements).
```

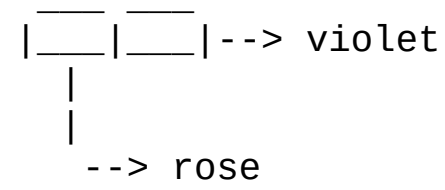
Here is the list `(A ())`, or equivalently `(A nil)`, depicted with boxes and arrows:



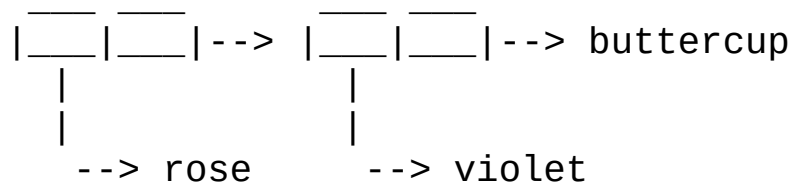
Dotted Pair Notation

Dotted pair notation is an alternative syntax for cons cells that represents the CAR and CDR explicitly. In this syntax, `(a . b)` stands for a cons cell whose CAR is the object a, and whose CDR is the object b. Dotted pair notation is therefore more general than list syntax. In the dotted pair notation, the list `(1 2 3)' is written as `(1 . (2 . (3 . nil)))'. For `nil`-terminated lists, the two notations produce the same result, but list notation is usually clearer and more convenient when it is applicable. When printing a list, the dotted pair notation is only used if the CDR of a cell is not a list.

Box notation can also be used to illustrate what dotted pairs look like. For example, `(rose . violet)` is diagrammed as follows:

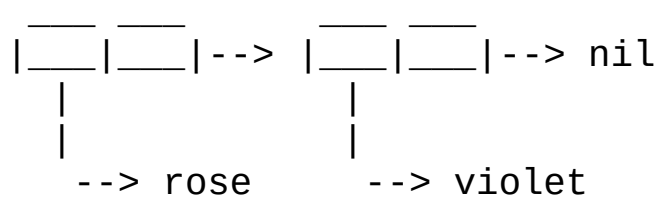


Dotted pair notation can be combined with list notation to represent a chain of cons cells with a non-`nil` final CDR. For example, `(rose violet . buttercup)` is equivalent to `(rose . (violet . buttercup))`. The object looks like this:



These diagrams make it evident that `(rose . violet . buttercup)` must have an invalid syntax since it would require that a cons cell have three parts rather than two.

The list `(rose violet)` is equivalent to `(rose . (violet))` and looks like this:



Similarly, the three-element list `(rose violet buttercup)` is equivalent to `(rose . (violet . (buttercup)))`.

Association List Type

An association list or alist is a specially-constructed list whose elements are cons cells. In each element, the CAR is considered a key, and the CDR is considered an associated value. (In some cases, the associated value is stored in the CAR of the CDR.) Association lists are often used to implement stacks, since new associations may easily be added to or removed from the front of the list.

For example,

```
(setq alist-of-colors
      '((rose . red) (lily . white)  (buttercup . yellow)))
```

sets the variable `alist-of-colors` to an alist of three elements. In the first element, `rose` is the key and `red` is the value.

See section Association Lists, for a further explanation of alists and for functions that work on alists.

Array Type

An array is composed of an arbitrary number of other Lisp objects, arranged in a contiguous block of memory. Any element of an array may be accessed in constant time. In contrast, accessing an element of a list requires time proportional to the position of the element in the list. (Elements at the end of a list take longer to access than elements at the beginning of a list.)

Emacs defines two types of array, strings and vectors. A string is an array of characters and a vector is an array of arbitrary objects. Both are one-dimensional. (Most other programming languages support

multidimensional arrays, but we don't think they are essential in Emacs Lisp.) Each type of array has its own read syntax; see section String Type, and section Vector Type.

An array may have any length up to the largest integer; but once created, it has a fixed size. The first element of an array has index zero, the second element has index 1, and so on. This is called zero-origin indexing. For example, an array of four elements has indices 0, 1, 2, and 3.

The array type is contained in the sequence type and contains both strings and vectors.

String Type

A string is an array of characters. Strings are used for many purposes in Emacs, as can be expected in a text editor; for example, as the names of Lisp symbols, as messages for the user, and to represent text extracted from buffers. Strings in Lisp are constants; evaluation of a string returns the same string.

The read syntax for strings is a double-quote, an arbitrary number of characters, and another double-quote, `"like this"`. The Lisp reader accepts the same formats for reading the characters of a string as it does for reading single characters (without the question mark that begins a character literal). You can enter a nonprinting character such as tab, C-a or M-C-A using the convenient escape sequences, like this: `"\t, \C-a, \M-\C-a"`. You can include a double-quote in a string by preceding it with a backslash; thus, `"\""` is a string containing just a single double-quote character. (See section Character Type, for a description of the read syntax for characters.)

If you use the `\M-' syntax to indicate a meta character in a string constant, this sets the 2**7 bit of the character in the string. This is not the same representation that the meta modifier has in a character regarded as a simple integer. See section Character Type.

Strings cannot hold characters that have the hyper, super or alt modifiers; they can hold ASCII control characters, but no others. They do not distinguish case in ASCII control characters.

In contrast with the C programming language, Emacs Lisp allows newlines in string literals. But an escaped newline--one that is preceded by `\'---does not become part of the string; i.e., the Lisp reader ignores an escaped newline in a string literal.

```
"It is useful to include newlines
in documentation strings,
but the newline is \
ignored if escaped."
     => "It is useful to include newlines
in documentation strings,
but the newline is ignored if escaped."
```

The printed representation of a string consists of a double-quote, the characters it contains, and another double-quote. However, any backslash or double-quote characters in the string are preceded with a backslash like this: `"this \" is an embedded quote"`.

A string can hold properties of the text it contains, in addition to the characters themselves. This enables programs that copy text between strings and buffers to preserve the properties with no special effort. See

section Text Properties. Strings with text properties have a special read and print syntax:

```
#("characters" property-data...)
```

where property-data is zero or more elements in groups of three as follows:

```
beg end plist
```

The elements beg and end are integers, and together specify a portion of the string; plist is the property list for that portion.

See section Strings and Characters, for functions that work on strings.

Vector Type

A vector is a one-dimensional array of elements of any type. It takes a constant amount of time to access any element of a vector. (In a list, the access time of an element is proportional to the distance of the element from the beginning of the list.)

The printed representation of a vector consists of a left square bracket, the elements, and a right square bracket. This is also the read syntax. Like numbers and strings, vectors are considered constants for evaluation.

```
[1 "two" (three)]      ; A vector of three elements.
     => [1 "two" (three)]
```

See section Vectors, for functions that work with vectors.

Symbol Type

A symbol in GNU Emacs Lisp is an object with a name. The symbol name serves as the printed representation of the symbol. In ordinary use, the name is unique--no two symbols have the same name.

A symbol may be used in programs as a variable, as a function name, or to hold a list of properties. Or it may serve only to be distinct from all other Lisp objects, so that its presence in a data structure may be recognized reliably. In a given context, usually only one of these uses is intended.

A symbol name can contain any characters whatever. Most symbol names are written with letters, digits, and the punctuation characters `-+=*/'. Such names require no special punctuation; the characters of the name suffice as long as the name does not look like a number. (If it does, write a `\' at the beginning of the name to force interpretation as a symbol.) The characters `_~!@$%^&:<>{}' are less often used but also require no special punctuation. Any other characters may be included in a symbol's name by escaping them with a backslash. In contrast to its use in strings, however, a backslash in the name of a symbol quotes the single character that follows the backslash, without conversion. For example, in a string, `\t' represents a tab character; in the name of a symbol, however, `\t' merely quotes the letter t. To have a symbol with a tab character in its name, you must actually type an tab (preceded with a backslash). But you would hardly ever do such a thing.

Common Lisp note: in Common Lisp, lower case letters are always "folded" to upper case, unless they are explicitly escaped. This is in contrast to Emacs Lisp, in which upper case and lower case letters are distinct.

Here are several examples of symbol names. Note that the `+' in the fifth example is escaped to prevent it from being read as a number. This is not necessary in the last example because the rest of the name makes it invalid as a number.

```
foo                 ; A symbol named `foo'.
FOO                 ; A symbol named `FOO', different from `foo'.
char-to-string      ; A symbol named `char-to-string'.
1+                  ; A symbol named `1+'
                    ;   (not `+1', which is an integer).
\+1                 ; A symbol named `+1'
                    ;   (not a very readable name).
\(*\ 1\ 2\)         ; A symbol named `(* 1 2)' (a worse name).
+-*/_~!@$%^&=:<>{}  ; A symbol named `+-*/_~!@$%^&=:<>{}'.
                    ;   These characters need not be escaped.
```

Lisp Function Type

Just as functions in other programming languages are executable, Lisp function objects are pieces of executable code. However, functions in Lisp are primarily Lisp objects, and only secondarily the text which represents them. These Lisp objects are lambda expressions: lists whose first element is the symbol `lambda` (see section Lambda Expressions).

In most programming languages, it is impossible to have a function without a name. In Lisp, a function has no intrinsic name. A lambda expression is also called an anonymous function (see section Anonymous Functions). A named function in Lisp is actually a symbol with a valid function in its function cell (see section Defining Named Functions).

Most of the time, functions are called when their names are written in Lisp expressions in Lisp programs. However, a function object found or constructed at run time can be called and passed arguments with the primitive functions `funcall` and `apply`. See section Calling Functions.

Lisp Macro Type

A Lisp macro is a user-defined construct that extends the Lisp language. It is represented as an object much like a function, but with different parameter-passing semantics. A Lisp macro has the form of a list whose first element is the symbol `macro` and whose CDR is a Lisp function object, including the `lambda` symbol.

Lisp macro objects are usually defined with the built-in `defmacro` function, but any list that begins with `macro` is a macro as far as Emacs is concerned. See section Macros, for an explanation of how to write a macro.

Primitive Function Type

A primitive function is a function callable from Lisp but written in the C programming language. Primitive functions are also called subrs or built-in functions. (The word "subr" is derived from "subroutine".) Most primitive functions evaluate all their arguments when they are called. A primitive function that does not evaluate all its arguments is called a special form (see section Special Forms).

It does not matter to the caller of a function whether the function is primitive. However, this does matter if you are trying to substitute a function written in Lisp for a primitive of the same name. The reason is that the primitive function may be called directly from C code. When the redefined function is called from Lisp, the new definition will be used; but calls from C code may still use the old definition.

The term function is used to refer to all Emacs functions, whether written in Lisp or C. See section Lisp Function Type, for information about the functions written in Lisp.

Primitive functions have no read syntax and print in hash notation with the name of the subroutine.

```
(symbol-function 'car)          ; Access the function cell
                                ;   of the symbol.
     => #<subr car>
(subrp (symbol-function 'car))  ; Is this a primitive function?
     => t                       ; Yes.
```

Byte-Code Function Type

The byte compiler produces byte-code function objects. Internally, a byte-code function object is much like a vector; however, the evaluator handles this data type specially when it appears as a function to be called. See section Byte Compilation, for information about the byte compiler.

The printed representation for a byte-code function object is like that for a vector, with an additional `#' before the opening `['.

Autoload Type

An autoload object is a list whose first element is the symbol `autoload`. It is stored as the function definition of a symbol to say that a file of Lisp code should be loaded when necessary to find the true definition of that symbol. The autoload object contains the name of the file, plus some other information about the real definition.

After the file has been loaded, the symbol should have a new function definition that is not an autoload object. The new definition is then called as if it had been there to begin with. From the user's point of view, the function call works as expected, using the function definition in the loaded file.

An autoload object is usually created with the function `autoload`, which stores the object in the function cell of a symbol. See section Autoload, for more details.

Editing Types

The types in the previous section are common to many Lisp-like languages. But Emacs Lisp provides several additional data types for purposes connected with editing.

Buffer Type

A buffer is an object that holds text that can be edited (see section Buffers). Most buffers hold the contents of a disk file (see section Files) so they can be edited, but some are used for other purposes. Most buffers are also meant to be seen by the user, and therefore displayed, at some time, in a window (see section Windows). But a buffer need not be displayed in a window.

The contents of a buffer are much like a string, but buffers are not used like strings in Emacs Lisp, and the available operations are different. For example, text can be inserted into a buffer very quickly, while "inserting" text into a string is accomplished by concatenation and the result is an entirely new string object.

Each buffer has a designated position called point (see section Positions). And one buffer is the current buffer. Most editing commands act on the contents of the current buffer in the neighborhood of point. Many other functions manipulate or test the characters in the current buffer and much of this manual is devoted to describing these functions (see section Text).

Several other data structures are associated with each buffer:

- a local syntax table (see section Syntax Tables);
- a local keymap (see section Keymaps); and,
- a local variable binding list (see section Buffer-Local Variables).

The local keymap and variable list contain entries which individually override global bindings or values. These are used to customize the behavior of programs in different buffers, without actually changing the programs.

Buffers have no read syntax. They print in hash notation with the buffer name.

```
(current-buffer)
     => #<buffer objects.texi>
```

Window Type

A window describes the portion of the terminal screen that Emacs uses to display a buffer. Every window has one associated buffer, whose contents appear in the window. By contrast, a given buffer may appear in one window, no window, or several windows.

Though many windows may exist simultaneously, one window is designated the selected window. This is the window where the cursor is (usually) displayed when Emacs is ready for a command. The selected window usually displays the current buffer, but this is not necessarily the case.

Windows are grouped on the screen into frames; each window belongs to one and only one frame. See section Frame Type.

Windows have no read syntax. They print in hash notation, giving the window number and the name of the buffer being displayed. The window numbers exist to identify windows uniquely, since the buffer displayed in any given window can change frequently.

```
(selected-window)
     => #<window 1 on objects.texi>
```

See section Windows, for a description of the functions that work on windows.

Frame Type

A frame is a rectangle on the screen that contains one or more Emacs windows. A frame initially contains a single main window (plus perhaps a minibuffer window) which you can subdivide vertically or horizontally into smaller windows.

Frames have no read syntax. They print in hash notation, giving the frame's title, plus its address in core (useful to identify the frame uniquely).

```
(selected-frame)
     => #<frame xemacs@mole.gnu.ai.mit.edu 0xdac80>
```

See section Frames, for a description of the functions that work on frames.

Window Configuration Type

A window configuration stores information about the positions and sizes of windows at the time the window configuration is created, so that the screen layout may be recreated later.

Window configurations have no read syntax. They print as `#<window-configuration>'. See section Window Configurations, for a description of several functions related to window configurations.

Marker Type

A marker denotes a position in a specific buffer. Markers therefore have two components: one for the buffer, and one for the position. The position value is changed automatically as necessary as text is inserted into or deleted from the buffer. This is to ensure that the marker always points between the same two characters in the buffer.

Markers have no read syntax. They print in hash notation, giving the current character position and the name of the buffer.

```
(point-marker)
     => #<marker at 10779 in objects.texi>
```

See section Markers, for information on how to test, create, copy, and move markers.

Process Type

The word process means a running program. Emacs itself runs in a process of this sort. However, in Emacs Lisp, a process is a Lisp object that designates a subprocess created by Emacs process. External subprocesses, such as shells, GDB, ftp, and compilers, may be used to extend the processing capability of Emacs.

A process takes input from Emacs and returns output to Emacs for further manipulation. Both text and signals can be communicated between Emacs and a subprocess.

Processes have no read syntax. They print in hash notation, giving the name of the process:

```
(process-list)
     => (#<process shell>)
```

See section Processes, for information about functions that create, delete, return information about, send input or signals to, and receive output from processes.

Stream Type

A stream is an object that can be used as a source or sink for characters--either to supply characters for input or to accept them as output. Many different types can be used this way: markers, buffers, strings, and functions. Most often, input streams (character sources) obtain characters from the keyboard, a buffer, or a file, and output streams (character sinks) send characters to a buffer, such as a `` `*Help*' `` buffer, or to the echo area.

The object `nil`, in addition to its other meanings, may be used as a stream. It stands for the value of the variable `standard-input` or `standard-output`. Also, the object `t` as a stream specifies input using the minibuffer (see section Minibuffers) or output in the echo area (see section The Echo Area).

Streams have no special printed representation or read syntax, and print as whatever primitive type they are.

See section Reading and Printing Lisp Objects, for a description of various functions related to streams, including various parsing and printing functions.

Keymap Type

A keymap maps keys typed by the user to functions. This mapping controls how the user's command input is executed. A keymap is actually a list whose CAR is the symbol `keymap`.

See section Keymaps, for information about creating keymaps, handling prefix keys, local as well as global keymaps, and changing key bindings.

Syntax Table Type

A syntax table is a vector of 256 integers. Each element of the vector defines how one character is interpreted when it appears in a buffer. For example, in C mode (see section Major Modes), the `+' character is punctuation, but in Lisp mode it is a valid character in a symbol. These different interpretations are effected by changing the syntax table entry for `+', i.e., at index 43.

Syntax tables are only used for scanning text in buffers, not for reading Lisp expressions. The table the Lisp interpreter uses to read expressions is built into the Emacs source code and cannot be changed; thus, to change the list delimiters to be `{' and `}' instead of `(' and `)' would be impossible.

See section Syntax Tables, for details about syntax classes and how to make and modify syntax tables.

Display Table Type

A display table specifies how to display each character code. Each buffer and each window can have its own display table. A display table is actually a vector of length 261. See section Display Tables.

Overlay Type

An overlay specifies temporary alteration of the display appearance of a part of a buffer. It contains markers delimiting a range of the buffer, plus a property list (a list whose elements are alternating property names and values). Overlays are used to present parts of the buffer temporarily in a different display style.

See section Overlays, for how to create and use overlays.

Type Predicates

The Emacs Lisp interpreter itself does not perform type checking on the actual arguments passed to functions when they are called. It could not do otherwise, since variables in Lisp are not declared to be of a certain type, as they are in other programming languages. It is therefore up to the individual function to test whether each actual argument belongs to a type that can be used by the function.

All built-in functions do check the types of their actual arguments when appropriate and signal a `wrong-type-argument` error if an argument is of the wrong type. For example, here is what happens if you pass an argument to `+` which it cannot handle:

```
(+ 2 'a)
     error--> Wrong type argument: integer-or-marker-p, a
```

Many functions, called type predicates, are provided to test whether an object is a member of a given type. (Following a convention of long standing, the names of most Emacs Lisp predicates end in `p'.)

Here is a table of predefined type predicates, in alphabetical order, with references to further information.

`atom`
: see section Predicates on Lists

`arrayp`
: see section Functions that Operate on Arrays

`bufferp`
: see section Buffer Basics

`byte-code-function-p`
: see section Byte-Code Function Type

`case-table-p`
: see section The Case Table

`char-or-string-p`
: see section The Predicates for Strings

`commandp`
: see section Interactive Call

`consp`
: see section Predicates on Lists

`floatp`
: see section Type Predicates for Numbers

`frame-live-p`
: see section Deleting Frames

`framep`
: see section Frames

`integer-or-marker-p`
: see section Predicates on Markers

`integerp`
: see section Type Predicates for Numbers

`keymapp`
: see section Creating Keymaps

`listp`
: see section Predicates on Lists

`markerp`
: see section Predicates on Markers

`natnump`
: see section Type Predicates for Numbers

`nlistp`

see section Predicates on Lists

`numberp`

see section Type Predicates for Numbers

`number-or-marker-p`

see section Predicates on Markers

`overlayp`

see section Overlays

`processp`

see section Processes

`sequencep`

see section Sequences

`stringp`

see section The Predicates for Strings

`subrp`

see section Accessing Function Cell Contents

`symbolp`

see section Symbols

`syntax-table-p`

see section Syntax Tables

`user-variable-p`

see section Defining Global Variables

`vectorp`

see section Vectors

`window-configuration-p`

see section Window Configurations

`window-live-p`

see section Deleting Windows

`windowp`

see section Basic Concepts of Emacs Windows

Equality Predicates

Here we describe two functions that test for equality between any two objects. Other functions test equality between objects of specific types, e.g., strings. See the appropriate chapter describing the data type for these predicates.

Function: **eq** *object1 object2*

This function returns `t` if object1 and object2 are the same object, `nil` otherwise. The "same object" means that a change in one will be reflected by the same change in the other.

`eq` returns `t` if object1 and object2 are integers with the same value. Also, since symbol names are normally unique, if the arguments are symbols with the same name, they are `eq`. For other types (e.g., lists, vectors, strings), two arguments with the same contents or elements are not necessarily `eq` to each other: they are `eq` only if they are the same object.

(The `make-symbol` function returns an uninterned symbol that is not interned in the standard `obarray`. When uninterned symbols are in use, symbol names are no longer unique. Distinct symbols with the same name are not `eq`. See section Creating and Interning Symbols.)

```
(eq 'foo 'foo)
     => t

(eq 456 456)
     => t

(eq "asdf" "asdf")
     => nil

(eq '(1 (2 (3))) '(1 (2 (3))))
     => nil

(eq [(1 2) 3] [(1 2) 3])
     => nil

(eq (point-marker) (point-marker))
     => nil
```

Function: **equal** *object1 object2*

This function returns `t` if object1 and object2 have equal components, `nil` otherwise. Whereas `eq` tests if its arguments are the same object, `equal` looks inside nonidentical arguments to see if their elements are the same. So, if two objects are `eq`, they are `equal`, but the converse is not always true.

```
(equal 'foo 'foo)
     => t

(equal 456 456)
     => t

(equal "asdf" "asdf")
     => t
(eq "asdf" "asdf")
```


```
     => nil

(equal '(1 (2 (3))) '(1 (2 (3))))
     => t
(eq '(1 (2 (3))) '(1 (2 (3))))
     => nil

(equal [(1 2) 3] [(1 2) 3])
     => t
(eq [(1 2) 3] [(1 2) 3])
     => nil

(equal (point-marker) (point-marker))
     => t

(eq (point-marker) (point-marker))
     => nil
```

Comparison of strings is case-sensitive.

```
(equal "asdf" "ASDF")
     => nil
```

The test for equality is implemented recursively, and circular lists may therefore cause infinite recursion (leading to an error).

Go to the previous, next section.

Go to the previous, next section.

Numbers

GNU Emacs supports two numeric data types: integers and floating point numbers. Integers are whole numbers such as -3, 0, 7, 13, and 511. Their values are exact. Floating point numbers are numbers with fractional parts, such as -4.5, 0.0, or 2.71828. They can also be expressed in an exponential notation as well: thus, 1.5e2 equals 150; in this example, `e2' stands for ten to the second power, and is multiplied by 1.5. Floating point values are not exact; they have a fixed, limited amount of precision.

Support for floating point numbers is a new feature in Emacs 19, and it is controlled by a separate compilation option, so you may encounter a site where Emacs does not support them.

Integer Basics

The range of values for an integer depends on the machine. The range is -8388608 to 8388607 (24 bits; i.e., to) on most machines, but on others it is -16777216 to 16777215 (25 bits), or -33554432 to 33554431 (26 bits). All of the examples shown below assume an integer has 24 bits.

The Lisp reader reads numbers as a sequence of digits with an optional sign.

```
 1               ; The integer 1.
+1               ; Also the integer 1.
-1               ; The integer -1.
 16777217        ; Also the integer 1, due to overflow.
 0               ; The number 0.
-0               ; The number 0.
 1.              ; The integer 1.
```

To understand how various functions work on integers, especially the bitwise operators (see section Bitwise Operations on Integers), it is often helpful to view the numbers in their binary form.

In 24 bit binary, the decimal integer 5 looks like this:

```
0000 0000  0000 0000  0000 0101
```

(We have inserted spaces between groups of 4 bits, and two spaces between groups of 8 bits, to make the binary integer easier to read.)

The integer -1 looks like this:

```
1111 1111  1111 1111  1111 1111
```

-1 is represented as 24 ones. (This is called two's complement notation.)

The negative integer, -5, is creating by subtracting 4 from -1. In binary, the decimal integer 4 is 100. Consequently, -5 looks like this:

```
1111 1111  1111 1111  1111 1011
```

In this implementation, the largest 24 bit binary integer is the decimal integer 8,388,607. In binary, this number looks like this:

```
0111 1111  1111 1111  1111 1111
```

Since the arithmetic functions do not check whether integers go outside their range, when you add 1 to 8,388,607, the value is negative integer -8,388,608:

```
(+ 1 8388607)
     => -8388608
     => 1000 0000  0000 0000  0000 0000
```

Many of the following functions accept markers for arguments as well as integers. (See section Markers.) More precisely, the actual parameters to such functions may be either integers or markers, which is why we often give these parameters the name int-or-marker. When the actual parameter is a marker, the position value of the marker is used and the buffer of the marker is ignored.

Floating Point Basics

Emacs version 19 supports floating point numbers, if compiled with the macro `LISP_FLOAT_TYPE` defined. The precise range of floating point numbers is machine-specific; it is the same as the range of the C data type `double` on the machine in question.

The printed representation for floating point numbers requires either a decimal point (with at least one digit following), an exponent, or both. For example, `1500.0', `15e2', `15.0e2', `1.5e3', and `.15e4' are five ways of writing a floating point number whose value is 1500. They are all equivalent. You can also use a minus sign to write negative floating point numbers, as in `-1.0'.

You can use `logb` to extract the binary exponent of a floating point number (or estimate the logarithm of an integer):

Function: **logb** *number*

This function returns the binary exponent of number. More precisely, the value is the logarithm of number base 2, rounded down to an integer.

Type Predicates for Numbers

The functions in this section test whether the argument is a number or whether it is a certain sort of number. The functions `integerp` and `floatp` can take any type of Lisp object as argument (the predicates would not be of much use otherwise); but the `zerop` predicate requires a number as its

argument. See also `integer-or-marker-p` and `number-or-marker-p`, in section Predicates on Markers.

Function: **floatp** *object*

This predicate tests whether its argument is a floating point number and returns `t` if so, `nil` otherwise.

`floatp` does not exist in Emacs versions 18 and earlier.

Function: **integerp** *object*

This predicate tests whether its argument is an integer, and returns `t` if so, `nil` otherwise.

Function: **numberp** *object*

This predicate tests whether its argument is a number (either integer or floating point), and returns `t` if so, `nil` otherwise.

Function: **natnump** *object*

The `natnump` predicate (whose name comes from the phrase "natural-number-p") tests to see whether its argument is a nonnegative integer, and returns `t` if so, `nil` otherwise. 0 is considered non-negative.

Markers are not converted to integers, hence `natnump` of a marker is always `nil`.

People have pointed out that this function is misnamed, because the term "natural number" is usually understood as excluding zero. We are open to suggestions for a better name to use in a future version.

Function: **zerop** *number*

This predicate tests whether its argument is zero, and returns `t` if so, `nil` otherwise. The argument must be a number.

These two forms are equivalent: `(zerop x) == (= x 0)`.

Comparison of Numbers

Floating point numbers in Emacs Lisp actually take up storage, and there can be many distinct floating point number objects with the same numeric value. If you use `eq` to compare them, then you test whether two values are the same *object*. If you want to compare just the numeric values, use `=`.

If you use `eq` to compare two integers, it always returns `t` if they have the same value. This is sometimes useful, because `eq` accepts arguments of any type and never causes an error, whereas `=` signals an error if the arguments are not numbers or markers. However, it is a good idea to use `=` if you can, even for comparing integers, just in case we change the representation of integers in a future Emacs version.

There is another wrinkle: because floating point arithmetic is not exact, it is often a bad idea to check for equality of two floating point values. Usually it is better to test for approximate equality. Here's a function to do this:


```
(defvar fuzz-factor 1.0e-6)

(defun approx-equal (x y)
  (< (/ (abs (- x y))
        (max (abs x) (abs y)))
     fuzz-factor))
```

Common Lisp note: because of the way numbers are implemented in Common Lisp, you generally need to use `=' to test for equality between numbers of any kind.

Function: = *number-or-marker1 number-or-marker2*

This function tests whether its arguments are the same number, and returns `t` if so, `nil` otherwise.

Function: /= *number-or-marker1 number-or-marker2*

This function tests whether its arguments are not the same number, and returns `t` if so, `nil` otherwise.

Function: < *number-or-marker1 number-or-marker2*

This function tests whether its first argument is strictly less than its second argument. It returns `t` if so, `nil` otherwise.

Function: <= *number-or-marker1 number-or-marker2*

This function tests whether its first argument is less than or equal to its second argument. It returns `t` if so, `nil` otherwise.

Function: > *number-or-marker1 number-or-marker2*

This function tests whether its first argument is strictly greater than its second argument. It returns `t` if so, `nil` otherwise.

Function: >= *number-or-marker1 number-or-marker2*

This function tests whether its first argument is greater than or equal to its second argument. It returns `t` if so, `nil` otherwise.

Function: **max** *number-or-marker &rest numbers-or-markers*

This function returns the largest of its arguments.

```
(max 20)
     => 20
(max 1 2)
     => 2
(max 1 3 2)
     => 3
```

Function: **min** *number-or-marker &rest numbers-or-markers*

This function returns the smallest of its arguments.

Numeric Conversions

To convert an integer to floating point, use the function `float`.

Function: **float** *number*

This returns number converted to floating point. If number is already a floating point number, `float` returns it unchanged.

There are four functions to convert floating point numbers to integers; they differ in how they round. You can call these functions with an integer argument also; if you do, they return it without change.

Function: **truncate** *number*

This returns number, converted to an integer by rounding towards zero.

Function: **floor** *number*

This returns number, converted to an integer by rounding downward (towards negative infinity).

Function: **ceiling** *number*

This returns number, converted to an integer by rounding upward (towards positive infinity).

Function: **round** *number*

This returns number, converted to an integer by rounding towards the nearest integer.

Arithmetic Operations

Emacs Lisp provides the traditional four arithmetic operations: addition, subtraction, multiplication, and division. A remainder function supplements the (integer) division function. The functions to add or subtract 1 are provided because they are traditional in Lisp and commonly used.

All of these functions except % return a floating point value if any argument is floating.

It is important to note that in GNU Emacs Lisp, arithmetic functions do not check for overflow. Thus `(1+ 8388607)` may equal -8388608, depending on your hardware.

Function: **1+** *number-or-marker*

This function returns number-or-marker plus 1. For example,

```
(setq foo 4)
     => 4
(1+ foo)
     => 5
```

This function is not analogous to the C operator ++---it does not increment a variable. It just computes a

sum. Thus,

```
foo
     => 4
```

If you want to increment the variable, you must use `setq`, like this:

```
(setq foo (1+ foo))
     => 5
```

Function: **1-** *number-or-marker*

This function returns number-or-marker minus 1.

Function: **abs** *number*

This returns the absolute value of number.

Function: **+** *&rest numbers-or-markers*

This function adds its arguments together. When given no arguments, `+` returns 0. It does not check for overflow.

```
(+)
     => 0
(+ 1)
     => 1
(+ 1 2 3 4)
     => 10
```

Function: **-** *&optional number-or-marker &rest other-numbers-or-markers*

The `-` function serves two purposes: negation and subtraction. When `-` has a single argument, the value is the negative of the argument. When there are multiple arguments, each of the other-numbers-or-markers is subtracted from number-or-marker, cumulatively. If there are no arguments, the result is 0. This function does not check for overflow.

```
(- 10 1 2 3 4)
     => 0
(- 10)
     => -10
(-)
     => 0
```

Function: ***** *&rest numbers-or-markers*

This function multiplies its arguments together, and returns the product. When given no arguments, `*` returns 1. It does not check for overflow.


```
(*)
     => 1
(* 1)
     => 1
(* 1 2 3 4)
     => 24
```

Function: / *dividend divisor &rest divisors*

This function divides dividend by divisors and returns the quotient. If there are additional arguments divisors, then dividend is divided by each divisor in turn. Each argument may be a number or a marker.

If all the arguments are integers, then the result is an integer too. This means the result has to be rounded. On most machines, the result is rounded towards zero after each division, but some machines may round differently with negative arguments. This is because the Lisp function `/` is implemented using the C division operator, which has the same possibility for machine-dependent rounding. As a practical matter, all known machines round in the standard fashion.

If you divide by 0, an `arith-error` error is signaled. (See section Errors.)

```
(/ 6 2)
     => 3
(/ 5 2)
     => 2
(/ 25 3 2)
     => 4
(/ -17 6)
     => -2
```

Since the division operator in Emacs Lisp is implemented using the division operator in C, the result of dividing negative numbers may in principle vary from machine to machine, depending on how they round the result. Thus, the result of `(/ -17 6)` could be -3 on some machines. In practice, nearly all machines round the quotient towards 0.

Function: **%** *dividend divisor*

This function returns the value of dividend modulo divisor; in other words, the integer remainder after division of dividend by divisor. The sign of the result is the sign of dividend. The sign of divisor is ignored. The arguments must be integers.

For negative arguments, the value is in principle machine-dependent since the quotient is; but in practice, all known machines behave alike.

An `arith-error` results if divisor is 0.

```
(% 9 4)
     => 1
```


```
(% -9 4)
     => -1
(% 9 -4)
     => 1
(% -9 -4)
     => -1
```

For any two numbers dividend and divisor,

```
(+ (% dividend divisor)
   (* (/ dividend divisor) divisor))
```

always equals dividend.

Bitwise Operations on Integers

In a computer, an integer is represented as a binary number, a sequence of bits (digits which are either zero or one). A bitwise operation acts on the individual bits of such a sequence. For example, shifting moves the whole sequence left or right one or more places, reproducing the same pattern "moved over".

The bitwise operations in Emacs Lisp apply only to integers.

Function: **lsh** *integer1 count*

`lsh`, which is an abbreviation for logical shift, shifts the bits in integer1 to the left count places, or to the right if count is negative. If count is negative, `lsh` shifts zeros into the most-significant bit, producing a positive result even if integer1 is negative. Contrast this with `ash`, below.

Thus, the decimal number 5 is the binary number 00000101. Shifted once to the left, with a zero put in the one's place, the number becomes 00001010, decimal 10.

Here are two examples of shifting the pattern of bits one place to the left. Since the contents of the rightmost place has been moved one place to the left, a value has to be inserted into the rightmost place. With `lsh`, a zero is placed into the rightmost place. (These examples show only the low-order eight bits of the binary pattern; the rest are all zero.)

```
(lsh 5 1)
     => 10

;; Decimal 5 becomes decimal 10.
00000101 => 00001010

(lsh 7 1)
     => 14

;; Decimal 7 becomes decimal 14.
00000111 => 00001110
```

As the examples illustrate, shifting the pattern of bits one place to the left produces a number that is t the value of the previous number.

Note, however that functions do not check for overflow, and a returned value may be negative (and in any case, no more than a 24 bit value) when an integer is sufficiently left shifted.

For example, left shifting 8,388,607 produces -2:

```
(lsh 8388607 1)          ; left shift
     => -2
```

In binary, in the 24 bit implementation, the numbers looks like this:

```
;; Decimal 8,388,607
0111 1111  1111 1111  1111 1111
```

which becomes the following when left shifted:

```
;; Decimal -2
1111 1111  1111 1111  1111 1110
```

Shifting the pattern of bits two places to the left produces results like this (with 8-bit binary numbers):

```
(lsh 3 2)
     => 12

;; Decimal 3 becomes decimal 12.
00000011 => 00001100
```

On the other hand, shifting the pattern of bits one place to the right looks like this:

```
(lsh 6 -1)
     => 3

;; Decimal 6 becomes decimal 3.
00000110 => 00000011

(lsh 5 -1)
     => 2

;; Decimal 5 becomes decimal 2.
00000101 => 00000010
```

As the example illustrates, shifting the pattern of bits one place to the right divides the value of the binary number by two, rounding downward.

Function: **ash** *integer1 count*

`ash` (arithmetic shift) shifts the bits in integer1 to the left count places, or to the right if count is negative.

`ash` gives the same results as `lsh` except when integer1 and count are both negative. In that case, `ash` puts a one in the leftmost position, while `lsh` puts a zero in the leftmost position.

Thus, with `ash`, shifting the pattern of bits one place to the right looks like this:

```
(ash -6 -1)
     => -3

;; Decimal -6
;; becomes decimal -3.

1111 1111  1111 1111  1111 1010
     =>
1111 1111  1111 1111  1111 1101
```

In contrast, shifting the pattern of bits one place to the right with `lsh` looks like this:

```
(lsh -6 -1)
     => 8388605

;; Decimal -6
;; becomes decimal 8,388,605.

1111 1111  1111 1111  1111 1010
     =>
0111 1111  1111 1111  1111 1101
```

In this case, the 1 in the leftmost position is shifted one place to the right, and a zero is shifted into the leftmost position.

Here are other examples:

```
                  ;               24-bit binary values

(lsh 5 2)         ;   5  =  0000 0000  0000 0000  0000 0101
     => 20        ;  20  =  0000 0000  0000 0000  0001 0100
(ash 5 2)
     => 20
(lsh -5 2)        ;  -5  =  1111 1111  1111 1111  1111 1011
     => -20       ; -20  =  1111 1111  1111 1111  1110 1100
(ash -5 2)
     => -20
(lsh 5 -2)        ;   5  =  0000 0000  0000 0000  0000 0101
```


```
     => 1           ;   1  =  0000 0000  0000 0000  0000 0001
(ash 5 -2)
     => 1
(lsh -5 -2)         ;  -5  =  1111 1111  1111 1111  1111 1011
     => 4194302     ;         0011 1111  1111 1111  1111 1110
(ash -5 -2)         ;  -5  =  1111 1111  1111 1111  1111 1011
     => -2          ;  -2  =  1111 1111  1111 1111  1111 1110
```

Function: **logand** *&rest ints-or-markers*

This function returns the "logical and" of the arguments: the nth bit is set in the result if, and only if, the nth bit is set in all the arguments. ("Set" means that the value of the bit is 1 rather than 0.)

For example, using 4-bit binary numbers, the "logical and" of 13 and 12 is 12: 1101 combined with 1100 produces 1100.

In both the binary numbers, the leftmost two bits are set (i.e., they are 1's), so the leftmost two bits of the returned value are set. However, for the rightmost two bits, each is zero in at least one of the arguments, so the rightmost two bits of the returned value are 0's.

Therefore,

```
(logand 13 12)
     => 12
```

If `logand` is not passed any argument, it returns a value of -1. This number is an identity element for `logand` because its binary representation consists entirely of ones. If `logand` is passed just one argument, it returns that argument.

```
                    ;                  24-bit binary values

(logand 14 13)      ; 14  =  0000 0000  0000 0000  0000 1110
                    ; 13  =  0000 0000  0000 0000  0000 1101
     => 12          ; 12  =  0000 0000  0000 0000  0000 1100

(logand 14 13 4)    ; 14  =  0000 0000  0000 0000  0000 1110
                    ; 13  =  0000 0000  0000 0000  0000 1101
                    ;  4  =  0000 0000  0000 0000  0000 0100
     => 4           ;  4  =  0000 0000  0000 0000  0000 0100

(logand)
     => -1          ; -1  =  1111 1111  1111 1111  1111 1111
```

Function: **logior** *&rest ints-or-markers*

This function returns the "inclusive or" of its arguments: the nth bit is set in the result if, and only if, the nth bit is set in at least one of the arguments. If there are no arguments, the result is zero, which is an identity element for this operation. If `logior` is passed just one argument, it returns that argument.


```
                    ;                  24-bit binary values

(logior 12 5)       ; 12  =  0000 0000  0000 0000  0000 1100
                    ;  5  =  0000 0000  0000 0000  0000 0101
     => 13          ; 13  =  0000 0000  0000 0000  0000 1101

(logior 12 5 7)     ; 12  =  0000 0000  0000 0000  0000 1100
                    ;  5  =  0000 0000  0000 0000  0000 0101
                    ;  7  =  0000 0000  0000 0000  0000 0111
     => 15          ; 15  =  0000 0000  0000 0000  0000 1111
```

Function: **logxor** *&rest ints-or-markers*

This function returns the "exclusive or" of its arguments: the nth bit is set in the result if, and only if, the nth bit is set in an odd number of the arguments. If there are no arguments, the result is 0. If `logxor` is passed just one argument, it returns that argument.

```
                    ;                  24-bit binary values

(logxor 12 5)       ; 12  =  0000 0000  0000 0000  0000 1100
                    ;  5  =  0000 0000  0000 0000  0000 0101
     => 9           ;  9  =  0000 0000  0000 0000  0000 1001

(logxor 12 5 7)     ; 12  =  0000 0000  0000 0000  0000 1100
                    ;  5  =  0000 0000  0000 0000  0000 0101
                    ;  7  =  0000 0000  0000 0000  0000 0111
     => 14          ; 14  =  0000 0000  0000 0000  0000 1110
```

Function: **lognot** *integer*

This function returns the logical complement of its argument: the nth bit is one in the result if, and only if, the nth bit is zero in integer, and vice-versa.

```
;;  5  =  0000 0000  0000 0000  0000 0101
;; becomes
;; -6  =  1111 1111  1111 1111  1111 1010

(lognot 5)
     => -6
```

Transcendental Functions

These mathematical functions are available if floating point is supported. They allow integers as well as floating point numbers as arguments.

Function: **sin** *arg*

Function: **cos** *arg*

Function: **tan** *arg*

These are the ordinary trigonometric functions, with argument measured in radians.

Function: **asin** *arg*

The value of `(asin arg)` is a number between - pi / 2 and pi / 2 (inclusive) whose sine is arg; if, however, arg is out of range (outside [-1, 1]), then the result is a NaN.

Function: **acos** *arg*

The value of `(acos arg)` is a number between 0 and pi (inclusive) whose cosine is arg; if, however, arg is out of range (outside [-1, 1]), then the result is a NaN.

Function: **atan** *arg*

The value of `(atan arg)` is a number between - pi / 2 and pi / 2 (exclusive) whose tangent is arg.

Function: **exp** *arg*

This is the exponential function; it returns *e* to the power arg.

Function: **log** *arg &optional base*

This function returns the logarithm of arg, with base base. If you don't specify base, the base e is used. If arg is negative, the result is a NaN.

Function: **log10** *arg*

This function returns the logarithm of arg, with base 10. If arg is negative, the result is a NaN.

Function: **expt** *x y*

This function returns x raised to power y.

Function: **sqrt** *arg*

This returns the square root of arg.

Random Numbers

In a computer, a series of pseudo-random numbers is generated in a deterministic fashion. The numbers are not truly random, but they have certain properties that mimic a random series. For example, all possible values occur equally often in a pseudo-random series.

In Emacs, pseudo-random numbers are generated from a "seed" number. Starting from any given seed, the `random` function always generates the same sequence of numbers. Emacs always starts with the same seed value, so the sequence of values of `random` is actually the same in each Emacs run! For

example, in one operating system, the first call to `(random)` after you start Emacs always returns -1457731, and the second one always returns -7692030. This is helpful for debugging.

If you want truly unpredictable random numbers, execute `(random t)`. This chooses a new seed based on the current time of day and on Emacs' process ID number.

Function: **random** *&optional limit*

This function returns a pseudo-random integer. When called more than once, it returns a series of pseudo-random integers.

If limit is `nil`, then the value may in principle be any integer. If limit is a positive integer, the value is chosen to be nonnegative and less than limit (only in Emacs 19).

If limit is `t`, it means to choose a new seed based on the current time of day and on Emacs's process ID number.

On some machines, any integer representable in Lisp may be the result of `random`. On other machines, the result can never be larger than a certain maximum or less than a certain (negative) minimum.

Go to the previous, next section.

Go to the previous, next section.

Strings and Characters

A string in Emacs Lisp is an array that contains an ordered sequence of characters. Strings are used as names of symbols, buffers, and files, to send messages to users, to hold text being copied between buffers, and for many other purposes. Because strings are so important, many functions are provided expressly for manipulating them. Emacs Lisp programs use strings more often than individual characters.

See section Putting Keyboard Events in Strings, for special considerations when using strings of keyboard character events.

Introduction to Strings and Characters

Strings in Emacs Lisp are arrays that contain an ordered sequence of characters. Characters are represented in Emacs Lisp as integers; whether an integer was intended as a character or not is determined only by how it is used. Thus, strings really contain integers.

The length of a string (like any array) is fixed and independent of the string contents, and cannot be altered. Strings in Lisp are *not* terminated by a distinguished character code. (By contrast, strings in C are terminated by a character with ASCII code 0.) This means that any character, including the null character (ASCII code 0), is a valid element of a string.

Since strings are considered arrays, you can operate on them with the general array functions. (See section Sequences, Arrays, and Vectors.) For example, you can access or change individual characters in a string using the functions `aref` and `aset` (see section Functions that Operate on Arrays).

Each character in a string is stored in a single byte. Therefore, numbers not in the range 0 to 255 are truncated when stored into a string. This means that a string takes up much less memory than a vector of the same length.

Sometimes key sequences are represented as strings. When a string is a key sequence, string elements in the range 128 to 255 represent meta characters (which are extremely large integers) rather than keyboard events in the range 128 to 255.

Strings cannot hold characters that have the hyper, super or alt modifiers; they can hold ASCII control characters, but no others. They do not distinguish case in ASCII control characters. See section Character Type, for more information about representation of meta and other modifiers for keyboard input characters.

Like a buffer, a string can contain text properties for the characters in it, as well as the characters themselves. See section Text Properties.

See section Text, for information about functions that display strings or copy them into buffers. See section Character Type, and section String Type, for information about the syntax of characters and

strings.

The Predicates for Strings

For more information about general sequence and array predicates, see section Sequences, Arrays, and Vectors, and section Arrays.

Function: **stringp** *object*

This function returns `t` if object is a string, `nil` otherwise.

Function: **char-or-string-p** *object*

This function returns `t` if object is a string or a character (i.e., an integer), `nil` otherwise.

Creating Strings

The following functions create strings, either from scratch, or by putting strings together, or by taking them apart.

Function: **make-string** *count character*

This function returns a string made up of count repetitions of character. If count is negative, an error is signaled.

```
(make-string 5 ?x)
     => "xxxxx"
(make-string 0 ?x)
     => ""
```

Other functions to compare with this one include `char-to-string` (see section Conversion of Characters and Strings), `make-vector` (see section Vectors), and `make-list` (see section Building Cons Cells and Lists).

Function: **substring** *string start &optional end*

This function returns a new string which consists of those characters from string in the range from (and including) the character at the index start up to (but excluding) the character at the index end. The first character is at index zero.

```
(substring "abcdefg" 0 3)
     => "abc"
```

Here the index for `a' is 0, the index for `b' is 1, and the index for `c' is 2. Thus, three letters, `abc', are copied from the full string. The index 3 marks the character position up to which the substring is copied. The character whose index is 3 is actually the fourth character in the string.

A negative number counts from the end of the string, so that -1 signifies the index of the last character of the string. For example:

```
(substring "abcdefg" -3 -1)
     => "ef"
```

In this example, the index for `e' is -3, the index for `f' is -2, and the index for `g' is -1. Therefore, `e' and `f' are included, and `g' is excluded.

When `nil` is used as an index, it falls after the last character in the string. Thus:

```
(substring "abcdefg" -3 nil)
     => "efg"
```

Omitting the argument end is equivalent to specifying `nil`. It follows that `(substring string 0)` returns a copy of all of string.

```
(substring "abcdefg" 0)
     => "abcdefg"
```

But we recommend `copy-sequence` for this purpose (see section Sequences).

A `wrong-type-argument` error is signaled if either start or end are non-integers. An `args-out-of-range` error is signaled if start indicates a character following end, or if either integer is out of range for string.

Contrast this function with `buffer-substring` (see section Examining Buffer Contents), which returns a string containing a portion of the text in the current buffer. The beginning of a string is at index 0, but the beginning of a buffer is at index 1.

Function: **concat** *&rest sequences*

This function returns a new string consisting of the characters in the arguments passed to it. The arguments may be strings, lists of numbers, or vectors of numbers; they are not themselves changed. If no arguments are passed to `concat`, it returns an empty string.

```
(concat "abc" "-def")
     => "abc-def"
(concat "abc" (list 120 (+ 256 121)) [122])
     => "abcxyz"
(concat "The " "quick brown " "fox.")
     => "The quick brown fox."
(concat)
     => ""
```

The second example above shows how characters stored in strings are taken modulo 256. In other words, each character in the string is stored in one byte.

The `concat` function always constructs a new string that is not `eq` to any existing string.

When an argument is an integer (not a sequence of integers), it is converted to a string of digits making up the decimal printed representation of the integer. This special case exists for compatibility with Mocklisp, and we don't recommend you take advantage of it. If you want to convert an integer in this way, use `format` (see section Formatting Strings) or `int-to-string` (see section Conversion of Characters and Strings).

```
(concat 137)
     => "137"
(concat 54 321)
     => "54321"
```

For information about other concatenation functions, see the description of `mapconcat` in section Mapping Functions, `vconcat` in section Vectors, and `append` in section Building Cons Cells and Lists.

Comparison of Characters and Strings

Function: **char-equal** *character1 character2*

This function returns `t` if the arguments represent the same character, `nil` otherwise. This function ignores differences in case if `case-fold-search` is non-`nil`.

```
(char-equal ?x ?x)
     => t
(char-to-string (+ 256 ?x))
     => "x"
(char-equal ?x  (+ 256 ?x))
     => t
```

Function: **string=** *string1 string2*

This function returns `t` if the characters of the two strings match exactly; case is significant.

```
(string= "abc" "abc")
     => t
(string= "abc" "ABC")
     => nil
(string= "ab" "ABC")
     => nil
```

Function: **string-equal** *string1 string2*

`string-equal` is another name for `string=`.

Function: **string<** *string1 string2*

This function compares two strings a character at a time. First it scans both the strings at once to find the first pair of corresponding characters that do not match. If the lesser character of those two is the character from string1, then string1 is less, and this function returns `t`. If the lesser character is the one from string2, then string1 is greater, and this function returns `nil`. If the two strings match entirely, the value is `nil`.

Pairs of characters are compared by their ASCII codes. Keep in mind that lower case letters have higher numeric values in the ASCII character set than their upper case counterparts; numbers and many punctuation characters have a lower numeric value than upper case letters.

```
(string< "abc" "abd")
     => t
(string< "abd" "abc")
     => nil
(string< "123" "abc")
     => t
```

When the strings have different lengths, and they match up to the length of string1, then the result is `t`. If they match up to the length of string2, the result is `nil`. A string without any characters in it is the smallest possible string.

```
(string< "" "abc")
     => t
(string< "ab" "abc")
     => t
(string< "abc" "")
     => nil
(string< "abc" "ab")
     => nil
(string< "" "")
     => nil
```

Function: **string-lessp** *string1 string2*

`string-lessp` is another name for `string<`.

See `compare-buffer-substrings` in section Comparing Text, for a way to compare text in buffers.

Conversion of Characters and Strings

Characters and strings may be converted into each other and into integers. `format` and `prin1-to-string` (see section Output Functions) may also be used to convert Lisp objects into strings. `read-from-string` (see section Input Functions) may be used to "convert" a string

representation of a Lisp object into an object.

See section Documentation, for a description of functions which return a string representing the Emacs standard notation of the argument character (`single-key-description` and `text-char-description`). These functions are used primarily for printing help messages.

Function: **char-to-string** *character*

This function returns a new string with a length of one character. The value of character, modulo 256, is used to initialize the element of the string.

This function is similar to `make-string` with an integer argument of 1. (See section Creating Strings.) This conversion can also be done with `format` using the `%c' format specification. (See section Formatting Strings.)

```
(char-to-string ?x)
     => "x"
(char-to-string (+ 256 ?x))
     => "x"
(make-string 1 ?x)
     => "x"
```

Function: **string-to-char** *string*

This function returns the first character in string. If the string is empty, the function returns 0. The value is also 0 when the first character of string is the null character, ASCII code 0.

```
(string-to-char "ABC")
     => 65
(string-to-char "xyz")
     => 120
(string-to-char "")
     => 0
(string-to-char "\000")
     => 0
```

This function may be eliminated in the future if it does not seem useful enough to retain.

Function: **number-to-string** *number*

Function: **int-to-string** *number*

This function returns a string consisting of the printed representation of number, which may be an integer or a floating point number. The value starts with a sign if the argument is negative.

```
(int-to-string 256)
     => "256"
(int-to-string -23)
```


```
     => "-23"
(int-to-string -23.5)
     => "-23.5"
```

See also the function `format` in section Formatting Strings.

Function: **string-to-number** *string*

Function: **string-to-int** *string*

This function returns the integer value of the characters in string, read as a number in base ten. It skips spaces at the beginning of string, then reads as much of string as it can interpret as a number. (On some systems it ignores other whitespace at the beginning, not just spaces.) If the first character after the ignored whitespace is not a digit or a minus sign, this function returns 0.

```
(string-to-number "256")
     => 256
(string-to-number "25 is a perfect square.")
     => 25
(string-to-number "X256")
     => 0
(string-to-number "-4.5")
     => -4.5
```

Formatting Strings

Formatting means constructing a string by substitution of computed values at various places in a constant string. This string controls how the other values are printed as well as where they appear; it is called a format string.

Formatting is often useful for computing messages to be displayed. In fact, the functions `message` and `error` provide the same formatting feature described here; they differ from `format` only in how they use the result of formatting.

Function: **format** *string &rest objects*

This function returns a new string that is made by copying string and then replacing any format specification in the copy with encodings of the corresponding objects. The arguments objects are the computed values to be formatted.

A format specification is a sequence of characters beginning with a `%'. Thus, if there is a `%d' in string, the `format` function replaces it with the printed representation of one of the values to be formatted (one of the arguments objects). For example:

```
(format "The value of fill-column is %d." fill-column)
     => "The value of fill-column is 72."
```

If string contains more than one format specification, the format specifications are matched with successive values from objects. Thus, the first format specification in string is matched with the first value, the second format specification is matched with the second such value, and so on. Any extra format specifications (those for which there are no corresponding values) cause unpredictable behavior. Any extra values to be formatted will be ignored.

Certain format specifications require values of particular types. However, no error is signaled if the value actually supplied fails to have the expected type. Instead, the output is likely to be meaningless.

Here is a table of the characters that can follow `%' to make up a format specification:

`s'

Replace the specification with the printed representation of the object, made without quoting. Thus, strings are represented by their contents alone, with no `"' characters, and symbols appear without `\' characters.

If there is no corresponding object, the empty string is used.

`S'

Replace the specification with the printed representation of the object, made with quoting. Thus, strings are enclosed in `"' characters, and `\' characters appear where necessary before special characters.

If there is no corresponding object, the empty string is used.

`o'

Replace the specification with the base-eight representation of an integer.

`d'

Replace the specification with the base-ten representation of an integer.

`x'

Replace the specification with the base-sixteen representation of an integer.

`c'

Replace the specification with the character which is the value given.

`e'

Replace the specification with the exponential notation for a floating point number.

`f'

Replace the specification with the decimal-point notation for a floating point number.

`g'

Replace the specification with notation for a floating point number, using either exponential notation or decimal-point notation whichever is shorter.

`%'

A single `%' is placed in the string. This format specification is unusual in that it does not use a value. For example, `(format "%% %d" 30)` returns `"% 30"`.

Any other format character results in an `Invalid format operation' error.

Here are several examples:

```
(format "The name of this buffer is %s." (buffer-name))
     => "The name of this buffer is strings.texi."

(format "The buffer object prints as %s." (current-buffer))
     => "The buffer object prints as #<buffer strings.texi>."

(format "The octal value of 18 is %o,
         and the hex value is %x." 18 18)
     => "The octal value of 18 is 22,
         and the hex value is 12."
```

All the specification characters allow an optional numeric prefix between the `%' and the character. The optional numeric prefix defines the minimum width for the object. If the printed representation of the object contains fewer characters than this, then it is padded. The padding is on the left if the prefix is positive (or starts with zero) and on the right if the prefix is negative. The padding character is normally a space, but if the numeric prefix starts with a zero, zeros are used for padding.

```
(format "%06d will be padded on the left with zeros" 123)
     => "000123 will be padded on the left with zeros"

(format "%-6d will be padded on the right" 123)
     => "123    will be padded on the right"
```

`format` never truncates an object's printed representation, no matter what width you specify. Thus, you can use a numeric prefix to specify a minimum spacing between columns with no risk of losing information.

In the following three examples, `%7s' specifies a minimum width of 7. In the first case, the string inserted in place of `%7s' has only 3 letters, so 4 blank spaces are inserted for padding. In the second case, the string `"specification"` is 13 letters wide but is not truncated. In the third case, the padding is on the right.

```
(format "The word `%7s' actually has %d letters in it." "foo"
        (length "foo"))
     => "The word `    foo' actually has 3 letters in it."

(format "The word `%7s' actually has %d letters in it."
        "specification"
        (length "specification"))
     => "The word `specification' actually has 13 letters in it."

(format "The word `%-7s' actually has %d letters in it." "foo"
        (length "foo"))
     => "The word `foo    ' actually has 3 letters in it."
```

Character Case

The character case functions change the case of single characters or of the contents of strings. The functions convert only alphabetic characters (the letters `A' through `Z' and `a' through `z'); other characters are not altered. The functions do not modify the strings that are passed to them as arguments.

The examples below use the characters `X' and `x' which have ASCII codes 88 and 120 respectively.

Function: **downcase** *string-or-char*

This function converts a character or a string to lower case.

When the argument to `downcase` is a string, the function creates and returns a new string in which each letter in the argument that is upper case is converted to lower case. When the argument to `downcase` is a character, `downcase` returns the corresponding lower case character. This value is an integer. If the original character is lower case, or is not a letter, then the value equals the original character.

```
(downcase "The cat in the hat")
     => "the cat in the hat"

(downcase ?X)
     => 120
```

Function: **upcase** *string-or-char*

This function converts a character or a string to upper case.

When the argument to `upcase` is a string, the function creates and returns a new string in which each letter in the argument that is lower case is converted to upper case.

When the argument to `upcase` is a character, `upcase` returns the corresponding upper case character. This value is an integer. If the original character is upper case, or is not a letter, then the value equals the original character.

```
(upcase "The cat in the hat")
     => "THE CAT IN THE HAT"

(upcase ?x)
     => 88
```

Function: **capitalize** *string-or-char*

This function capitalizes strings or characters. If string-or-char is a string, the function creates and returns a new string, whose contents are a copy of string-or-char in which each word has been capitalized. This means that the first character of each word is converted to upper case, and the rest are converted to lower case.

The definition of a word is any sequence of consecutive characters that are assigned to the word

TOC

constituent category in the current syntax table (See section Table of Syntax Classes).

When the argument to `capitalize` is a character, `capitalize` has the same result as `upcase`.

```
(capitalize "The cat in the hat")
     => "The Cat In The Hat"

(capitalize "THE 77TH-HATTED CAT")
     => "The 77th-Hatted Cat"

(capitalize ?x)
     => 88
```

The Case Table

You can customize case conversion by installing a special case table. A case table specifies the mapping between upper case and lower case letters. It affects both the string and character case conversion functions (see the previous section) and those that apply to text in the buffer (see section Case Changes). Use case table if you are using a language which has letters that are not the standard ASCII letters.

A case table is a list of this form:

```
(downcase upcase canonicalize equivalences)
```

where each element is either `nil` or a string of length 256. The element downcase says how to map each character to its lower-case equivalent. The element upcase maps each character to its upper-case equivalent. If lower and upper case characters are in one-to-one correspondence, use `nil` for upcase; then Emacs deduces the upcase table from downcase.

For some languages, upper and lower case letters are not in one-to-one correspondence. There may be two different lower case letters with the same upper case equivalent. In these cases, you need to specify the maps for both directions.

The element canonicalize maps each character to a canonical equivalent; any two characters that are related by case-conversion have the same canonical equivalent character.

The element equivalences is a map that cyclicly permutes each equivalence class (of characters with the same canonical equivalent). (For ordinary ASCII, this would map `a' into `A' and `A' into `a', and likewise for each set of equivalent characters.)

You can provide `nil` for both canonicalize and equivalences, in which case both are deduced from downcase and upcase. Normally, that's what you should do, when you construct a case table. But when you look at the case table that's in use, you will find non-`nil` values for those components.

Each buffer has a case table. Emacs also has a standard case table which is copied into each buffer when you create the buffer. (Changing the standard case table doesn't affect any existing buffers.)

Here are the functions for working with case tables:

Function: **case-table-p** *object*

This predicate returns non-`nil` if object is a valid case table.

Function: **set-standard-case-table** *table*

This function makes table the standard case table, so that it will apply to any buffers created subsequently.

Function: **standard-case-table**

This returns the standard case table.

Function: **current-case-table**

This function returns the current buffer's case table.

Function: **set-case-table** *table*

This sets the current buffer's case table to table.

The following three functions are convenient subroutines for packages that define non-ASCII character sets. They modify a string downcase-table provided as an argument; this should be a string to be used as the downcase part of a case table. They also modify two syntax tables, the standard syntax table and the Text mode syntax table. (See section Syntax Tables.)

Function: **set-case-syntax-pair** *uc lc downcase-table*

This function specifies a pair of corresponding letters, one upper case and one lower case.

Function: **set-case-syntax-delims** *l r downcase-table*

This function makes characters l and r a matching pair of case-invariant delimiters.

Function: **set-case-syntax** *char syntax downcase-table*

This function makes char case-invariant, with syntax syntax.

Command: **describe-buffer-case-table**

This command displays a description of the contents of the current buffer's case table.

You can load the library `` `iso-syntax' `` to set up the syntax and case table for the 256 bit ISO Latin 1 character set.

Go to the previous, next section.

Go to the previous, next section.

Lists

A list represents a sequence of zero or more elements (which may be any Lisp objects). The important difference between lists and vectors is that two or more lists can share part of their structure; in addition, you can insert or delete elements in a list without copying the whole list.

Lists and Cons Cells

Lists in Lisp are not a primitive data type; they are built up from cons cells. A cons cell is a data object which represents an ordered pair. It records two Lisp objects, one labeled as the CAR, and the other labeled as the CDR. (These names are traditional.)

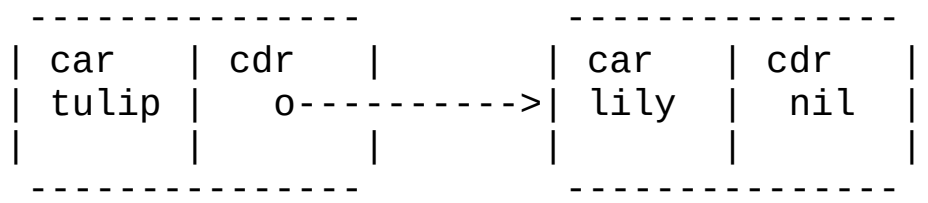
A list is made by chaining cons cells together, one cons cell per element. By convention, the CARs of the cons cells are the elements of the list, and the CDRs are used to chain the list: the CDR of each cons cell is the following cons cell. The CDR of the last cons cell is `nil`. This asymmetry between the CAR and the CDR is entirely a matter of convention; at the level of cons cells, the CAR and CDR slots have the same characteristics.

The symbol `nil` is considered a list as well as a symbol; it is the list with no elements. For convenience, the symbol `nil` is considered to have `nil` as its CDR (and also as its CAR).

The CDR of any nonempty list l is a list containing all the elements of l except the first.

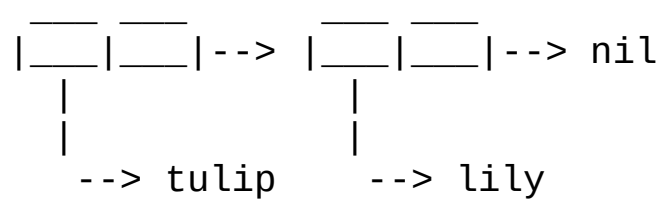
Lists as Linked Pairs of Boxes

A cons cell can be illustrated as a pair of boxes. The first box represents the CAR and the second box represents the CDR. Here is an illustration of the two-element list, `(tulip lily)`, made from two cons cells:

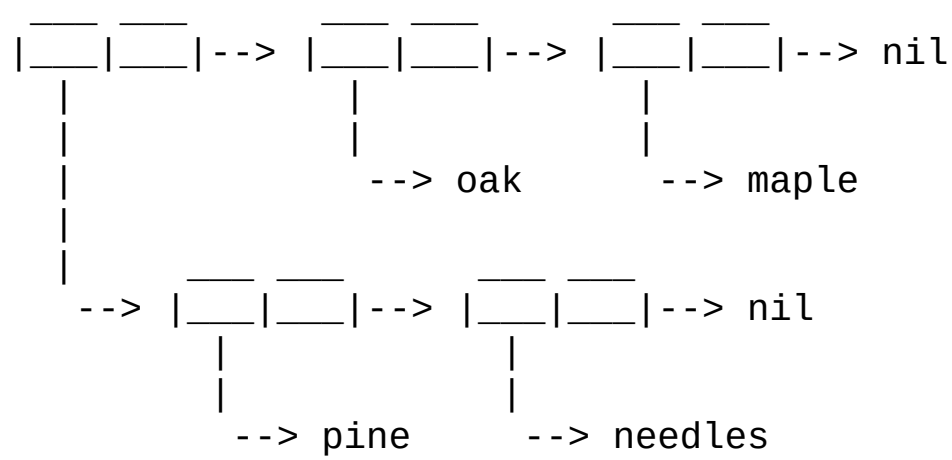


Each pair of boxes represents a cons cell. Each box "refers to", "points to" or "contains" a Lisp object. (These terms are synonymous.) The first box, which is the CAR of the first cons cell, contains the symbol `tulip`. The arrow from the CDR of the first cons cell to the second cons cell indicates that the CDR of the first cons cell points to the second cons cell.

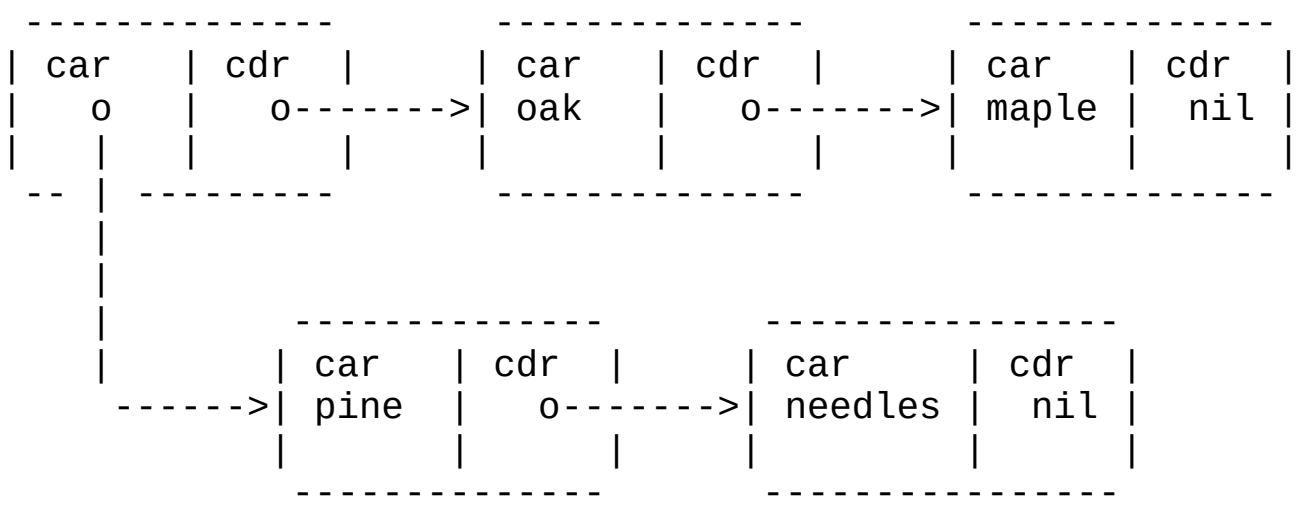
The same list can be illustrated in a different sort of box notation like this:



Here is a more complex illustration, this time of the three-element list, `((pine needles) oak maple)`, the first element of which is a two-element list:



The same list is represented in the first box notation like this:



See section List Type, for the read and print syntax of lists, and for more "box and arrow" illustrations of lists.

Predicates on Lists

The following predicates test whether a Lisp object is an atom, is a cons cell or is a list, or whether it is the distinguished object `nil`. (Many of these tests can be defined in terms of the others, but they are used so often that it is worth having all of them.)

Function: **consp** *object*

This function returns `t` if object is a cons cell, `nil` otherwise. `nil` is not a cons cell, although it *is* a

Function: **atom** *object*

This function returns `t` if object is an atom, `nil` otherwise. All objects except cons cells are atoms. The symbol `nil` is an atom and is also a list; it is the only Lisp object which both.

```
(atom object) == (not (consp object))
```

Function: **listp** *object*

This function returns `t` if object is a cons cell or `nil`. Otherwise, it returns `nil`.

```
(listp '(1))
     => t
(listp '())
     => t
```

Function: **nlistp** *object*

This function is the opposite of `listp`: it returns `t` if object is not a list. Otherwise, it returns `nil`.

```
(listp object) == (not (nlistp object))
```

Function: **null** *object*

This function returns `t` if object is `nil`, and returns `nil` otherwise. This function is identical to `not`, but as a matter of clarity we use `null` when object is considered a list and `not` when it is considered a truth value (see `not` in section Constructs for Combining Conditions).

```
(null '(1))
     => nil
(null '())
     => t
```

Accessing Elements of Lists

Function: **car** *cons-cell*

This function returns the value pointed to by the first pointer of the cons cell cons-cell. Expressed another way, this function returns the CAR of cons-cell.

As a special case, if cons-cell is `nil`, then `car` is defined to return `nil`; therefore, any list is a valid argument for `car`. An error is signaled if the argument is not a cons cell or `nil`.

```
(car '(a b c))
```


```
     => a
(car '())
     => nil
```

Function: **cdr** *cons-cell*

This function returns the value pointed to by the second pointer of the cons cell cons-cell. Expressed another way, this function returns the CDR of cons-cell.

As a special case, if cons-cell is `nil`, then `cdr` is defined to return `nil`; therefore, any list is a valid argument for `cdr`. An error is signaled if the argument is not a cons cell or `nil`.

```
(cdr '(a b c))
     => (b c)
(cdr '())
     => nil
```

Function: **car-safe** *object*

This function lets you take the CAR of a cons cell while avoiding errors for other data types. It returns the CAR of object if object is a cons cell, `nil` otherwise. This is in contrast to `car`, which signals an error if object is not a list.

```
(car-safe object)
==
(let ((x object))
  (if (consp x)
      (car x)
    nil))
```

Function: **cdr-safe** *object*

This function lets you take the CDR of a cons cell while avoiding errors for other data types. It returns the CDR of object if object is a cons cell, `nil` otherwise. This is in contrast to `cdr`, which signals an error if object is not a list.

```
(cdr-safe object)
==
(let ((x object))
  (if (consp x)
      (cdr x)
    nil))
```

Function: **nth** *n list*

This function returns the nth element of list. Elements are numbered starting with zero, so the CAR of list is element number zero. If the length of list is n or less, the value is `nil`.

If n is less than zero, then the first element is returned.

```
(nth 2 '(1 2 3 4))
     => 3
(nth 10 '(1 2 3 4))
     => nil
(nth -3 '(1 2 3 4))
     => 1

(nth n x) == (car (nthcdr n x))
```

Function: **nthcdr** *n list*

This function returns the nth cdr of list. In other words, it removes the first n links of list and returns what follows.

If n is less than or equal to zero, then all of list is returned. If the length of list is n or less, the value is `nil`.

```
(nthcdr 1 '(1 2 3 4))
     => (2 3 4)
(nthcdr 10 '(1 2 3 4))
     => nil
(nthcdr -3 '(1 2 3 4))
     => (1 2 3 4)
```

Building Cons Cells and Lists

Many functions build lists, as lists reside at the very heart of Lisp. `cons` is the fundamental list-building function; however, it is interesting to note that `list` is used more times in the source code for Emacs than `cons`.

Function: **cons** *object1 object2*

This function is the fundamental function used to build new list structure. It creates a new cons cell, making object1 the CAR, and object2 the CDR. It then returns the new cons cell. The arguments object1 and object2 may be any Lisp objects, but most often object2 is a list.

```
(cons 1 '(2))
     => (1 2)
(cons 1 '())
     => (1)
(cons 1 2)
     => (1 . 2)
```

`cons` is often used to add a single element to the front of a list. This is called consing the element onto

the list. For example:

```
(setq list (cons newelt list))
```

Note that there is no conflict between the variable named `list` used in this example and the function named `list` described below; any symbol can serve both functions.

Function: **list** *&rest objects*

This function creates a list with objects as its elements. The resulting list is always `nil`-terminated. If no objects are given, the empty list is returned.

```
(list 1 2 3 4 5)
     => (1 2 3 4 5)
(list 1 2 '(3 4 5) 'foo)
     => (1 2 (3 4 5) foo)
(list)
     => nil
```

Function: **make-list** *length object*

This function creates a list of length length, in which all the elements have the identical value object. Compare `make-list` with `make-string` (see section Creating Strings).

```
(make-list 3 'pigs)
     => (pigs pigs pigs)
(make-list 0 'pigs)
     => nil
```

Function: **append** *&rest sequences*

This function returns a list containing all the elements of sequences. The sequences may be lists, vectors, strings, or integers. All arguments except the last one are copied, so none of them are altered.

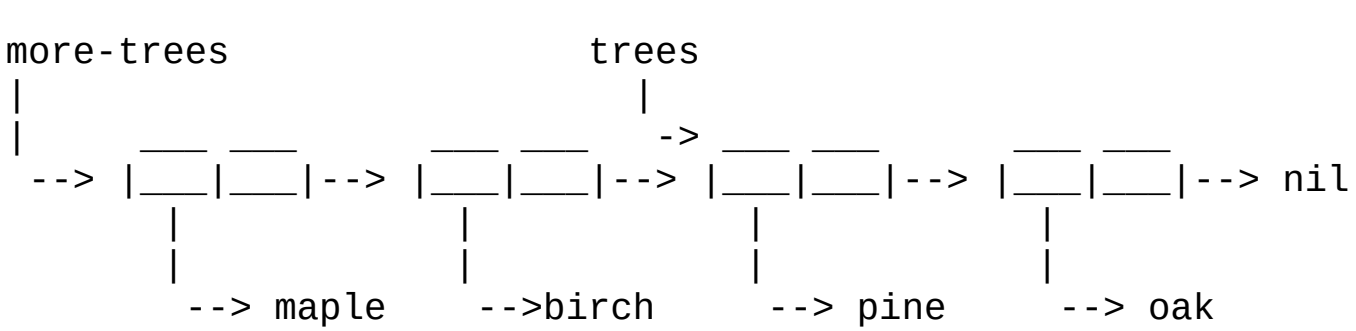
The final argument to `append` may be any object but it is typically a list. The final argument is not copied or converted; it becomes part of the structure of the new list.

Here is an example:

```
(setq trees '(pine oak))
     => (pine oak)
(setq more-trees (append '(maple birch) trees))
     => (maple birch pine oak)

trees
     => (pine oak)
more-trees
     => (maple birch pine oak)
```


```
(eq trees (cdr (cdr more-trees)))
     => t
```

You can see what happens by looking at a box diagram. The variable `trees` is set to the list `(pine oak)` and then the variable `more-trees` is set to the list `(maple birch pine oak)`. However, the variable `trees` continues to refer to the original list:



An empty sequence contributes nothing to the value returned by `append`. As a consequence of this, a final `nil` argument forces a copy of the previous argument.

```
trees
     => (pine oak)
(setq wood (append trees ()))
     => (pine oak)
wood
     => (pine oak)
(eq wood trees)
     => nil
```

This once was the standard way to copy a list, before the function `copy-sequence` was invented. See section Sequences, Arrays, and Vectors.

With the help of `apply`, we can append all the lists in a list of lists:

```
(apply 'append '((a b c) nil (x y z) nil))
     => (a b c x y z)
```

If no sequences are given, `nil` is returned:

```
(append)
     => nil
```

In the special case where one of the sequences is an integer (not a sequence of integers), it is first converted to a string of digits making up the decimal print representation of the integer. This special case exists for compatibility with Mocklisp, and we don't recommend you take advantage of it. If you want to convert an integer in this way, use `format` (see section Formatting Strings) or `number-to-string` (see section Conversion of Characters and Strings).


```
(setq trees '(pine oak))
     => (pine oak)
(char-to-string 54)
     => "6"
(setq longer-list (append trees 6 '(spruce)))
     => (pine oak 54 spruce)
(setq x-list (append trees 6 6))
     => (pine oak 54 . 6)
```

See `nconc` in section Functions that Rearrange Lists, for another way to join lists without copying.

Function: **reverse** *list*

This function creates a new list whose elements are the elements of list, but in reverse order. The original argument list is *not* altered.

```
(setq x '(1 2 3 4))
     => (1 2 3 4)
(reverse x)
     => (4 3 2 1)
x
     => (1 2 3 4)
```

Modifying Existing List Structure

You can modify the CAR and CDR contents of a cons cell with the primitives `setcar` and `setcdr`.

> **Common Lisp note:** Common Lisp uses functions `rplaca` and `rplacd` to alter list structure; they change structure the same way as `setcar` and `setcdr`, but the Common Lisp functions return the cons cell while `setcar` and `setcdr` return the new CAR or CDR.

Altering List Elements with `setcar`

Changing the CAR of a cons cell is done with `setcar` and replaces one element of a list with a different element.

Function: **setcar** *cons object*

This function stores object as the new CAR of cons, replacing its previous CAR. It returns the value object. For example:

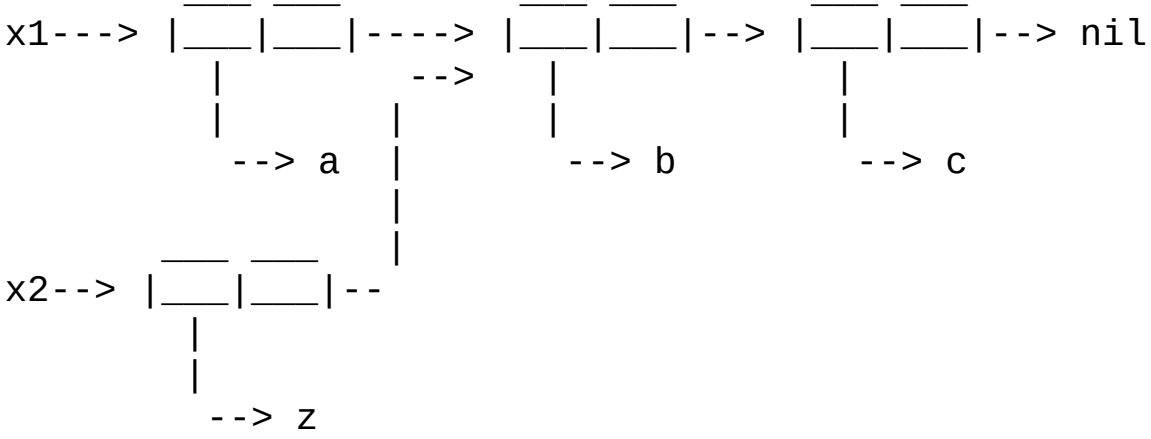
```
(setq x '(1 2))
     => (1 2)
(setcar x '4)
     => 4
x
```

```
     => (4 2)
```

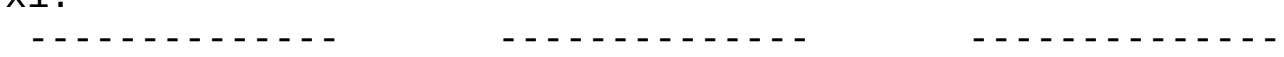
When a cons cell is part of the shared structure of several lists, storing a new CAR into the cons changes one element of each of these lists. Here is an example:

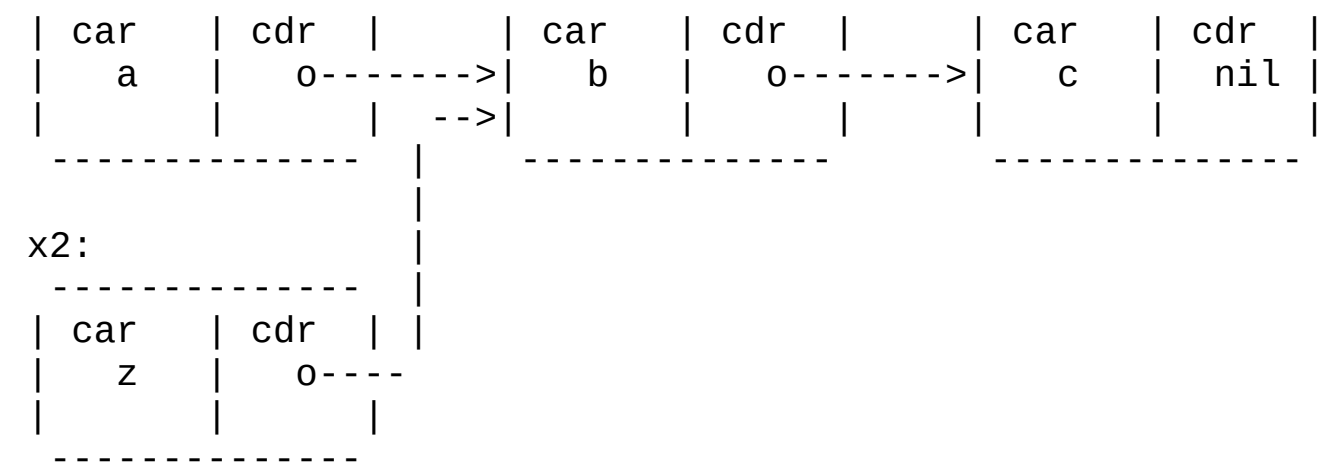
```
;; Create two lists that are partly shared.
(setq x1 '(a b c))
     => (a b c)
(setq x2 (cons 'z (cdr x1)))
     => (z b c)

;; Replace the CAR of a shared link.
(setcar (cdr x1) 'foo)
     => foo
x1                              ; Both lists are changed.
     => (a foo c)
x2
     => (z foo c)

;; Replace the CAR of a link that is not shared.
(setcar x1 'baz)
     => baz
x1                              ; Only one list is changed.
     => (baz foo c)
x2
     => (z foo c)
```

Here is a graphical depiction of the shared structure of the two lists x1 and x2, showing why replacing `b` changes them both:



Here is an alternative form of box diagram, showing the same relationship:





Altering the CDR of a List

The lowest-level primitive for modifying a CDR is `setcdr`:

Function: **setcdr** *cons object*

This function stores object into the cdr of cons. The value returned is object, not cons.

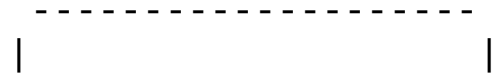
Here is an example of replacing the CDR of a list with a different list. All but the first element of the list are removed in favor of a different sequence of elements. The first element is unchanged, because it resides in the CAR of the list, and is not reached via the CDR.

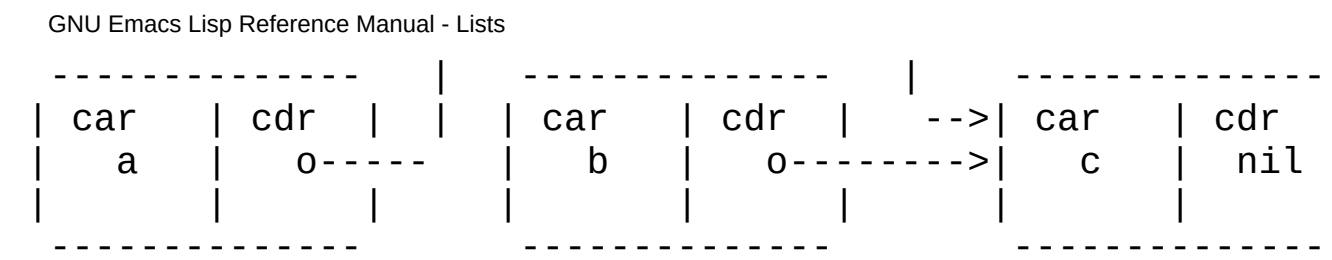
```
(setq x '(1 2 3))
     => (1 2 3)
(setcdr x '(4))
     => (4)
x
     => (1 4)
```

You can delete elements from the middle of a list by altering the CDRs of the cons cells in the list. For example, here we delete the second element, `b`, from the list `(a b c)`, by changing the CDR of the first cell:

```
(setq x1 '(a b c))
     => (a b c)
(setcdr x1 (cdr (cdr x1)))
     => (c)
x1
     => (a c)
```

Here is the result in box notation:



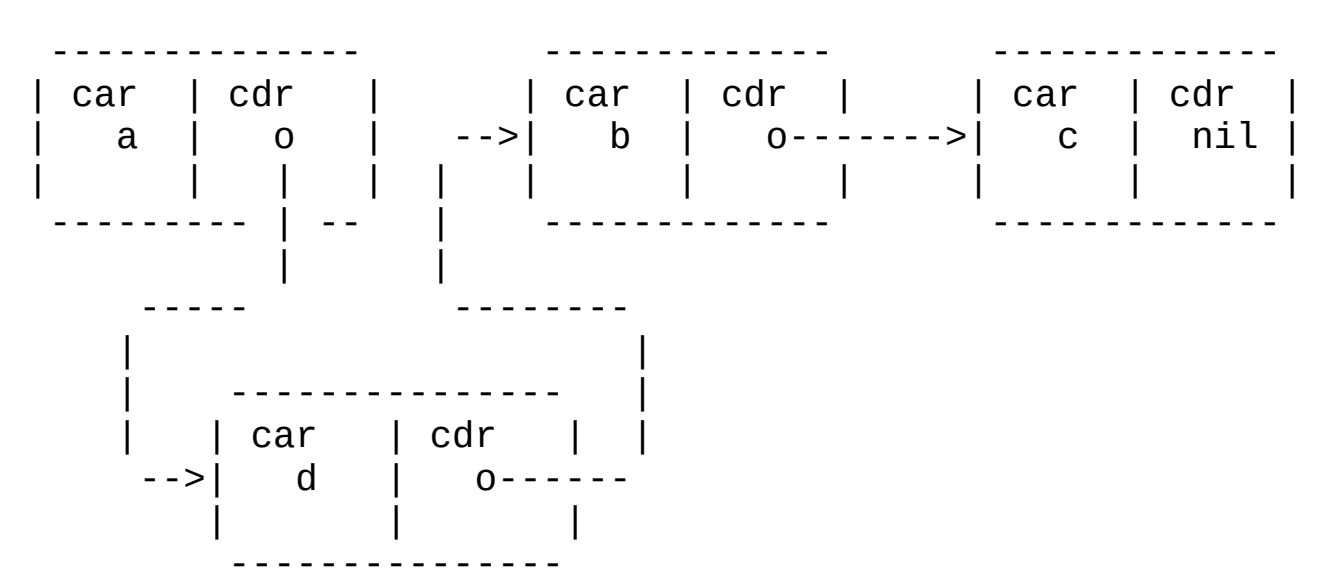


The second cons cell, which previously held the element `b`, still exists and its CAR is still `b`, but it no longer forms part of this list.

It is equally easy to insert a new element by changing CDRs:

```
(setq x1 '(a b c))
     => (a b c)
(setcdr x1 (cons 'd (cdr x1)))
     => (d b c)
x1
     => (a d b c)
```

Here is this result in box notation:



Functions that Rearrange Lists

Here are some functions that rearrange lists "destructively" by modifying the CDRs of their component cons cells. We call these functions "destructive" because the original lists passed as arguments to them are chewed up to produce a new list that is subsequently returned.

Function: **nconc** *&rest lists*

This function returns a list containing all the elements of lists. Unlike `append` (see section Building Cons Cells and Lists), the lists are *not* copied. Instead, the last CDR of each of the lists is changed to refer to the following list. The last of the lists is not altered. For example:


```
(setq x '(1 2 3))
     => (1 2 3)
(nconc x '(4 5))
     => (1 2 3 4 5)
x
     => (1 2 3 4 5)
```

Since the last argument of `nconc` is not itself modified, it is reasonable to use a constant list, such as `'(4 5)`, as is done in the above example. For the same reason, the last argument need not be a list:

```
(setq x '(1 2 3))
     => (1 2 3)
(nconc x 'z)
     => (1 2 3 . z)
x
     => (1 2 3 . z)
```

A common pitfall is to use a quoted constant list as a non-last argument to `nconc`. If you do this, your program will change each time you run it! Here is what happens:

```
(defun add-foo (x)              ; This function should add
  (nconc '(foo) x))             ;   foo to the front of its arg.

(symbol-function 'add-foo)
     => (lambda (x) (nconc (quote (foo)) x))

(setq xx (add-foo '(1 2)))    ; It seems to work.
     => (foo 1 2)
(setq xy (add-foo '(3 4)))    ; What happened?
     => (foo 1 2 3 4)
(eq xx xy)
     => t

(symbol-function 'add-foo)
     => (lambda (x) (nconc (quote (foo 1 2 3 4) x)))
```

Function: **nreverse** *list*

This function reverses the order of the elements of list. Unlike `reverse`, `nreverse` alters its argument destructively by reversing the CDRs in the cons cells forming the list. The cons cell which used to be the last one in list becomes the first cell of the value.

For example:

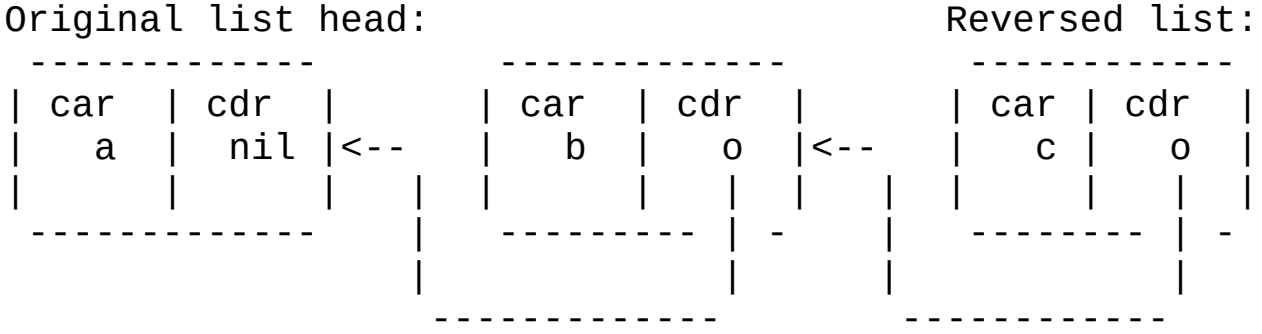
```
(setq x '(1 2 3 4))
     => (1 2 3 4)
x
```


```
     => (1 2 3 4)
(nreverse x)
     => (4 3 2 1)
;; The cell that was first is now last.
x
     => (1)
```

To avoid confusion, we usually store the result of `nreverse` back in the same variable which held the original list:

```
(setq x (nreverse x))
```

Here is the `nreverse` of our favorite example, `(a b c)`, presented graphically:



Function: **sort** *list predicate*

This function sorts list stably, though destructively, and returns the sorted list. It compares elements using predicate. A stable sort is one in which elements with equal sort keys maintain their relative order before and after the sort. Stability is important when successive sorts are used to order elements according to different criteria.

The argument predicate must be a function that accepts two arguments. It is called with two elements of list. To get an increasing order sort, the predicate should return `t` if the first element is "less than" the second, or `nil` if not.

The destructive aspect of `sort` is that it rearranges the cons cells forming list by changing CDRs. A nondestructive sort function would create new cons cells to store the elements in their sorted order. If you wish to sort a list without destroying the original, copy it first with `copy-sequence`.

The CARs of the cons cells are not changed; the cons cell that originally contained the element `a` in list still has `a` in its CAR after sorting, but it now appears in a different position in the list due to the change of CDRs. For example:

```
(setq nums '(1 3 2 6 5 4 0))
     => (1 3 2 6 5 4 0)
(sort nums '<)
     => (0 1 2 3 4 5 6)
nums
```


```
     => (1 2 3 4 5 6)
```

Note that the list in `nums` no longer contains 0; this is the same cons cell that it was before, but it is no longer the first one in the list. Don't assume a variable that formerly held the argument now holds the entire sorted list! Instead, save the result of `sort` and use that. Most often we store the result back into the variable that held the original list:

```
(setq nums (sort nums '<))
```

See section Sorting Text, for more functions that perform sorting. See `documentation` in section Access to Documentation Strings, for a useful example of `sort`.

The function `delq` in the following section is another example of destructive list manipulation.

Using Lists as Sets

A list can represent an unordered mathematical set--simply consider a value an element of a set if it appears in the list, and ignore the order of the list. To form the union of two sets, use `append` (as long as you don't mind having duplicate elements). Other useful functions for sets include `memq` and `delq`, and their `equal` versions, `member` and `delete`.

> **Common Lisp note:** Common Lisp has functions `union` (which avoids duplicate elements) and `intersection` for set operations, but GNU Emacs Lisp does not have them. You can write them in Lisp if you wish.

Function: **memq** *object list*

This function tests to see whether object is a member of list. If it is, `memq` returns a list starting with the first occurrence of object. Otherwise, it returns `nil`. The letter `q' in `memq` says that it uses `eq` to compare object against the elements of the list. For example:

```
(memq 2 '(1 2 3 2 1))
     => (2 3 2 1)
(memq '(2) '((1) (2)))    ; (2) and (2) are not eq.
     => nil
```

Function: **delq** *object list*

This function removes all elements `eq` to object from list. The letter `q' in `delq` says that it uses `eq` to compare object against the elements of the list, like `memq`.

When `delq` deletes elements from the front of the list, it does so simply by advancing down the list and returning a sublist that starts after those elements:

```
(delq 'a '(a b c))
==
(cdr '(a b c))
```

When an element to be deleted appears in the middle of the list, removing it involves changing the C
(see section Altering the CDR of a List).

```
(setq sample-list '(1 2 3 (4)))
     => (1 2 3 (4))
(delq 1 sample-list)
     => (2 3 (4))
sample-list
     => (1 2 3 (4))
(delq 2 sample-list)
     => (1 3 (4))
sample-list
     => (1 3 (4))
```

Note that `(delq 2 sample-list)` modifies `sample-list` to splice out the second element, but `(delq 1 sample-list)` does not splice anything--it just returns a shorter list. Don't assume that a variable which formerly held the argument list now has fewer elements, or that it still holds the original list! Instead, save the result of `delq` and use that. Most often we store the result back into the variable that held the original list:

```
(setq flowers (delq 'rose flowers))
```

In the following example, the `(4)` that `delq` attempts to match and the `(4)` in the `sample-list` are not `eq`:

```
(delq '(4) sample-list)
     => (1 3 (4))
```

The following two functions are like `memq` and `delq` but use `equal` rather than `eq` to compare elements. They are new in Emacs 19.

Function: **member** *object list*

The function `member` tests to see whether object is a member of list, comparing members with object using `equal`. If object is a member, `memq` returns a list starting with its first occurrence in list. Otherwise, it returns `nil`.

Compare this with `memq`:

```
(member '(2) '((1) (2)))  ; (2) and (2) are equal.
     => ((2))
(memq '(2) '((1) (2)))    ; (2) and (2) are not eq.
     => nil
;; Two strings with the same contents are equal.
(member "foo" '("foo" "bar"))
     => ("foo" "bar")
```

Function: **delete** *object list*

This function removes all elements `equal` to object from list. It is to `delq` as `member` is to `memq`: it uses `equal` to compare elements with object, like `member`; when it finds an element that matches, it removes the element just as `delq` would. For example:

```
(delete '(2) '((2) (1) (2)))
     => '((1))
```

Common Lisp note: The functions `member` and `delete` in GNU Emacs Lisp are derived from Maclisp, not Common Lisp. The Common Lisp versions do not use `equal` to compare elements.

Association Lists

An association list, or alist for short, records a mapping from keys to values. It is a list of cons cells called associations: the CAR of each cell is the key, and the CDR is the associated value. (This usage of "key" is not related to the term "key sequence"; it means any object which can be looked up in a table.)

Here is an example of an alist. The key `pine` is associated with the value `cones`; the key `oak` is associated with `acorns`; and the key `maple` is associated with `seeds`.

```
'((pine . cones)
  (oak . acorns)
  (maple . seeds))
```

The associated values in an alist may be any Lisp objects; so may the keys. For example, in the following alist, the symbol `a` is associated with the number `1`, and the string `"b"` is associated with the *list* `(2 3)`, which is the CDR of the alist element:

```
((a . 1) ("b" 2 3))
```

Sometimes it is better to design an alist to store the associated value in the CAR of the CDR of the element. Here is an example:

```
'((rose red) (lily white) (buttercup yellow))
```

Here we regard `red` as the value associated with `rose`. One advantage of this method is that you can store other related information--even a list of other items--in the CDR of the CDR. One disadvantage is that you cannot use `rassq` (see below) to find the element containing a given value. When neither of these considerations is important, the choice is a matter of taste, as long as you are consistent about it for any given alist.

Note that the same alist shown above could be regarded as having the associated value in the CDR of the element; the value associated with `rose` would be the list `(red)`.

Association lists are often used to record information that you might otherwise keep on a stack, since

new associations may be added easily to the front of the list. When searching an association list for an association with a given key, the first one found is returned, if there is more than one.

In Emacs Lisp, it is *not* an error if an element of an association list is not a cons cell. The alist search functions simply ignore such elements. Many other versions of Lisp signal errors in such cases.

Note that property lists are similar to association lists in several respects. A property list behaves like an association list in which each key can occur only once. See section Property Lists, for a comparison of property lists and association lists.

Function: **assoc** *key alist*

This function returns the first association for key in alist. It compares key against the alist elements using `equal` (see section Equality Predicates). It returns `nil` if no association in alist has a CAR `equal` to key. For example:

```
(setq trees '((pine . cones) (oak . acorns) (maple . seeds)))
     => ((pine . cones) (oak . acorns) (maple . seeds))
(assoc 'oak trees)
     => (oak . acorns)
(cdr (assoc 'oak trees))
     => acorns
(assoc 'birch trees)
     => nil
```

Here is another example in which the keys and values are not symbols:

```
(setq needles-per-cluster
      '((2 . ("Austrian Pine" "Red Pine"))
        (3 . "Pitch Pine")
        (5 . "White Pine")))

(cdr (assoc 3 needles-per-cluster))
     => "Pitch Pine"
(cdr (assoc 2 needles-per-cluster))
     => ("Austrian Pine" "Red Pine")
```

Function: **assq** *key alist*

This function is like `assoc` in that it returns the first association for key in alist, but it makes the comparison using `eq` instead of `equal`. `assq` returns `nil` if no association in alist has a CAR `eq` to key. This function is used more often than `assoc`, since `eq` is faster than `equal` and most alists use symbols as keys. See section Equality Predicates.

```
(setq trees '((pine . cones) (oak . acorns) (maple . seeds)))

(assq 'pine trees)
```


```
     => (pine . cones)
```

On the other hand, `assq` is not usually useful in alists where the keys may not be symbols:

```
(setq leaves
      '(("simple leaves" . oak)
        ("compound leaves" . horsechestnut)))

(assq "simple leaves" leaves)
     => nil
(assoc "simple leaves" leaves)
     => ("simple leaves" . oak)
```

Function: **rassq** *alist value*

This function returns the first association with value value in alist. It returns `nil` if no association in alist has a CDR `eq` to value.

`rassq` is like `assq` except that the CDR of the alist associations is tested instead of the CAR. You can think of this as "reverse `assq`", finding the key for a given value.

For example:

```
(setq trees '((pine . cones) (oak . acorns) (maple . seeds)))

(rassq 'acorns trees)
     => (oak . acorns)
(rassq 'spores trees)
     => nil
```

Note that `rassq` cannot be used to search for a value stored in the CAR of the CDR of an element:

```
(setq colors '((rose red) (lily white) (buttercup yellow)))

(rassq 'white colors)
     => nil
```

In this case, the CDR of the association `(lily white)` is not the symbol `white`, but rather the list `(white)`. This can be seen more clearly if the association is written in dotted pair notation:

```
(lily white) == (lily . (white))
```

Function: **copy-alist** *alist*

This function returns a two-level deep copy of alist: it creates a new copy of each association, so that you can alter the associations of the new alist without changing the old one.

```
(setq needles-per-cluster
```


```
      '((2 . ("Austrian Pine" "Red Pine"))
        (3 . "Pitch Pine")
        (5 . "White Pine")))
=>
((2 "Austrian Pine" "Red Pine")
 (3 . "Pitch Pine")
 (5 . "White Pine"))

(setq copy (copy-alist needles-per-cluster))
=>
((2 "Austrian Pine" "Red Pine")
 (3 . "Pitch Pine")
 (5 . "White Pine"))

(eq needles-per-cluster copy)
     => nil
(equal needles-per-cluster copy)
     => t
(eq (car needles-per-cluster) (car copy))
     => nil
(cdr (car (cdr needles-per-cluster)))
     => "Pitch Pine"
(eq (cdr (car (cdr needles-per-cluster)))
    (cdr (car (cdr copy))))
     => t
```

Go to the previous, next section.

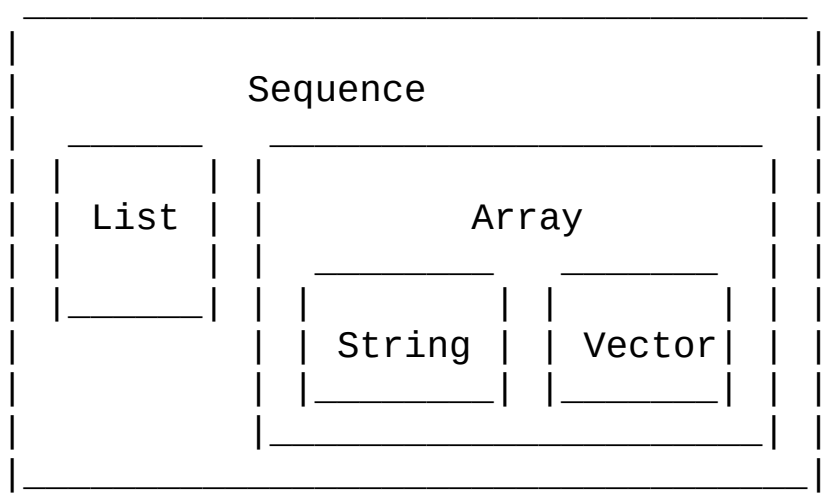
Go to the previous, next section.

Sequences, Arrays, and Vectors

Recall that the sequence type is the union of three other Lisp types: lists, vectors, and strings. In other words, any list is a sequence, any vector is a sequence, and any string is a sequence. The common property that all sequences have is that each is an ordered collection of elements.

An array is a single primitive object directly containing all its elements. Therefore, all the elements are accessible in constant time. The length of an existing array cannot be changed. Both strings and vectors are arrays. A list is a sequence of elements, but it is not a single primitive object; it is made of cons cells, one cell per element. Therefore, elements farther from the beginning of the list take longer to access, but it is possible to add elements to the list or remove elements. The elements of vectors and lists may be any Lisp objects. The elements of strings are all characters.

The following diagram shows the relationship between these types:



The Relationship between Sequences, Arrays, and Vectors

Sequences

In Emacs Lisp, a sequence is either a list, a vector or a string. The common property that all sequences have is that each is an ordered collection of elements. This section describes functions that accept any kind of sequence.

Function: **sequencep** *object*

Returns `t` if object is a list, vector, or string, `nil` otherwise.

Function: **copy-sequence** *sequence*

Returns a copy of sequence. The copy is the same type of object as the original sequence, and it has the

same elements in the same order.

Storing a new element into the copy does not affect the original sequence, and vice versa. However, the elements of the new sequence are not copies; they are identical (`eq`) to the elements of the original. Therefore, changes made within these elements, as found via the copied sequence, are also visible in the original sequence.

If the sequence is a string with text properties, the property list in the copy is itself a copy, not shared with the original's property list. However, the actual values of the properties are shared. See section Text Properties.

See also `append` in section Building Cons Cells and Lists, `concat` in section Creating Strings, and `vconcat` in section Vectors, for others ways to copy sequences.

```
(setq bar '(1 2))
     => (1 2)
(setq x (vector 'foo bar))
     => [foo (1 2)]
(setq y (copy-sequence x))
     => [foo (1 2)]

(eq x y)
     => nil
(equal x y)
     => t
(eq (elt x 1) (elt y 1))
     => t

;; Replacing an element of one sequence.
(aset x 0 'quux)
x => [quux (1 2)]
y => [foo (1 2)]

;; Modifying the inside of a shared element.
(setcar (aref x 1) 69)
x => [quux (69 2)]
y => [foo (69 2)]
```

Function: **length** *sequence*

Returns the number of elements in sequence. If sequence is a cons cell that is not a list (because the final CDR is not `nil`), a `wrong-type-argument` error is signaled.

```
(length '(1 2 3))
    => 3
(length ())
```


```
    => 0
(length "foobar")
    => 6
(length [1 2 3])
    => 3
```

Function: **elt** *sequence index*

This function returns the element of sequence indexed by index. Legitimate values of index are integers ranging from 0 up to one less than the length of sequence. If sequence is a list, then out-of-range values of index return `nil`; otherwise, they produce an `args-out-of-range` error.

```
(elt [1 2 3 4] 2)
     => 3
(elt '(1 2 3 4) 2)
     => 3
(char-to-string (elt "1234" 2))
     => "3"
(elt [1 2 3 4] 4)
     error-->Args out of range: [1 2 3 4], 4
(elt [1 2 3 4] -1)
     error-->Args out of range: [1 2 3 4], -1
```

This function duplicates `aref` (see section Functions that Operate on Arrays) and `nth` (see section Accessing Elements of Lists), except that it works for any kind of sequence.

Arrays

An array object refers directly to a number of other Lisp objects, called the elements of the array. Any element of an array may be accessed in constant time. In contrast, an element of a list requires access time that is proportional to the position of the element in the list.

When you create an array, you must specify how many elements it has. The amount of space allocated depends on the number of elements. Therefore, it is impossible to change the size of an array once it is created. You cannot add or remove elements. However, you can replace an element with a different value.

Emacs defines two types of array, both of which are one-dimensional: strings and vectors. A vector is a general array; its elements can be any Lisp objects. A string is a specialized array; its elements must be characters (i.e., integers between 0 and 255). Each type of array has its own read syntax. See section String Type, and section Vector Type.

Both kinds of arrays share these characteristics:

- The first element of an array has index zero, the second element has index 1, and so on. This is called zero-origin indexing. For example, an array of four elements has indices 0, 1, 2, and 3.
- The elements of an array may be referenced or changed with the functions `aref` and `aset`,

respectively (see section Functions that Operate on Arrays).

In principle, if you wish to have an array of characters, you could use either a string or a vector. In practice, we always choose strings for such applications, for four reasons:

- They occupy one-fourth the space of a vector of the same elements.
- Strings are printed in a way that shows the contents more clearly as characters.
- Strings can hold text properties. See section Text Properties.
- Many of the specialized editing and I/O facilities of Emacs accept only strings. For example, you cannot insert a vector of characters into a buffer the way you can insert a string. See section Strings and Characters.

Functions that Operate on Arrays

In this section, we describe the functions that accept both strings and vectors.

Function: **arrayp** *object*

This function returns `t` if object is an array (i.e., either a vector or a string).

```
(arrayp [a])
=> t
(arrayp "asdf")
=> t
```

Function: **aref** *array index*

This function returns the indexth element of array. The first element is at index zero.

```
(setq primes [2 3 5 7 11 13])
     => [2 3 5 7 11 13]
(aref primes 4)
     => 11
(elt primes 4)
     => 11

(aref "abcdefg" 1)
     => 98           ; `b' is ASCII code 98.
```

See also the function `elt`, in section Sequences.

Function: **aset** *array index object*

This function sets the indexth element of array to be object. It returns object.

```
(setq w [foo bar baz])
```


```
     => [foo bar baz]
(aset w 0 'fu)
     => fu
w
     => [fu bar baz]

(setq x "asdfasfd")
     => "asdfasfd"
(aset x 3 ?Z)
     => 90
x
     => "asdZasfd"
```

If array is a string and object is not a character, a `wrong-type-argument` error results.

Function: **fillarray** *array object*

This function fills the array array with pointers to object, replacing any previous values. It returns array.

```
(setq a [a b c d e f g])
     => [a b c d e f g]
(fillarray a 0)
     => [0 0 0 0 0 0 0]
a
     => [0 0 0 0 0 0 0]
(setq s "When in the course")
     => "When in the course"
(fillarray s ?-)
     => "------------------"
```

If array is a string and object is not a character, a `wrong-type-argument` error results.

The general sequence functions `copy-sequence` and `length` are often useful for objects known to be arrays. See section Sequences.

Vectors

Arrays in Lisp, like arrays in most languages, are blocks of memory whose elements can be accessed in constant time. A vector is a general-purpose array; its elements can be any Lisp objects. (The other kind of array provided in Emacs Lisp is the string, whose elements must be characters.) The main uses of vectors in Emacs are as syntax tables (vectors of integers) and keymaps (vectors of commands). They are also used internally as part of the representation of a byte-compiled function; if you print such a function, you will see a vector in it.

The indices of the elements of a vector are numbered starting with zero in Emacs Lisp.

Vectors are printed with square brackets surrounding the elements in their order. Thus, a vector

containing the symbols `a`, `b` and `c` is printed as `[a b c]`. You can write vectors in the same way in Lisp input.

A vector, like a string or a number, is considered a constant for evaluation: the result of evaluating it is the same vector. The elements of the vector are not evaluated. See section Self-Evaluating Forms.

Here are examples of these principles:

```
(setq avector [1 two '(three) "four" [five]])
     => [1 two (quote (three)) "four" [five]]
(eval avector)
     => [1 two (quote (three)) "four" [five]]
(eq avector (eval avector))
     => t
```

Here are some functions that relate to vectors:

Function: **vectorp** *object*

This function returns `t` if object is a vector.

```
(vectorp [a])
     => t
(vectorp "asdf")
     => nil
```

Function: **vector** *&rest objects*

This function creates and returns a vector whose elements are the arguments, objects.

```
(vector 'foo 23 [bar baz] "rats")
     => [foo 23 [bar baz] "rats"]
(vector)
     => []
```

Function: **make-vector** *integer object*

This function returns a new vector consisting of integer elements, each initialized to object.

```
(setq sleepy (make-vector 9 'Z))
     => [Z Z Z Z Z Z Z Z Z]
```

Function: **vconcat** *&rest sequences*

This function returns a new vector containing all the elements of the sequences. The arguments sequences may be lists, vectors, or strings. If no sequences are given, an empty vector is returned.

The value is a newly constructed vector that is not `eq` to any existing vector.


```
(setq a (vconcat '(A B C) '(D E F)))
     => [A B C D E F]
(eq a (vconcat a))
     => nil
(vconcat)
     => []
(vconcat [A B C] "aa" '(foo (6 7)))
     => [A B C 97 97 foo (6 7)]
```

When an argument is an integer (not a sequence of integers), it is converted to a string of digits making up the decimal printed representation of the integer. This special case exists for compatibility with Mocklisp, and we don't recommend you take advantage of it. If you want to convert an integer in this way, use `format` (see section Formatting Strings) or `int-to-string` (see section Conversion of Characters and Strings).

For other concatenation functions, see `mapconcat` in section Mapping Functions, `concat` in section Creating Strings, and `append` in section Building Cons Cells and Lists.

The `append` function may be used to convert a vector into a list with the same elements (see section Building Cons Cells and Lists):

```
(setq avector [1 two (quote (three)) "four" [five]])
     => [1 two (quote (three)) "four" [five]]
(append avector nil)
     => (1 two (quote (three)) "four" [five])
```

Go to the previous, next section.

Go to the previous, next section.

Symbols

A symbol is an object with a unique name. This chapter describes symbols, their components, and how they are created and interned. Property lists are also described. The uses of symbols as variables and as function names are described in separate chapters; see section Variables, and section Functions. For the precise syntax for symbols, see section Symbol Type.

You can test whether an arbitrary Lisp object is a symbol with `symbolp`:

Function: **symbolp** *object*

This function returns `t` if object is a symbol, `nil` otherwise.

Symbol Components

Each symbol has four components (or "cells"), each of which references another object:

Print name

> The print name cell holds a string which names the symbol for reading and printing. See `symbol-name` in section Creating and Interning Symbols.

Value

> The value cell holds the current value of the symbol as a variable. When a symbol is used as a form, the value of the form is the contents of the symbol's value cell. See `symbol-value` in section Accessing Variable Values.

Function

> The function cell holds the function definition of the symbol. When a symbol is used as a function, its function definition is used in its place. This cell is also used to make a symbol stand for a keymap or a keyboard macro, for editor command execution. Because each symbol has separate value and function cells, variables and function names do not conflict. See `symbol-function` in section Accessing Function Cell Contents.

Property list

> The property list cell holds the property list of the symbol. See `symbol-plist` in section Property Lists.

The print name cell always holds a string, and cannot be changed. The other three cells can be set individually to any specified Lisp object.

The print name cell holds the string that is the name of the symbol. Since symbols are represented textually by their names, it is important not to have two symbols with the same name. The Lisp reader ensures this: every time it reads a symbol, it looks for an existing symbol with the specified name before it creates a new one. (In GNU Emacs Lisp, this is done with a hashing algorithm that uses an obarray; see

section Creating and Interning Symbols.)

In normal usage, the function cell usually contains a function or macro, as that is what the Lisp interpreter expects to see there (see section Evaluation). Keyboard macros (see section Keyboard Macros), keymaps (see section Keymaps) and autoload objects (see section Autoloading) are also sometimes stored in the function cell of symbols. We often refer to "the function `foo`" when we really mean the function stored in the function cell of the symbol `foo`. We make the distinction only when necessary.

Similarly, the property list cell normally holds a correctly formatted property list (see section Property Lists), as a number of functions expect to see a property list there.

The function cell or the value cell may be void, which means that the cell does not reference any object. (This is not the same thing as holding the symbol `void`, nor the same as holding the symbol `nil`.) Examining the value of a cell which is void results in an error, such as `Symbol's value as variable is void'.

The four functions `symbol-name`, `symbol-value`, `symbol-plist`, and `symbol-function` return the contents of the four cells. Here as an example we show the contents of the four cells of the symbol `buffer-file-name`:

```
(symbol-name 'buffer-file-name)
     => "buffer-file-name"
(symbol-value 'buffer-file-name)
     => "/gnu/elisp/symbols.texi"
(symbol-plist 'buffer-file-name)
     => (variable-documentation 29529)
(symbol-function 'buffer-file-name)
     => #<subr buffer-file-name>
```

Because this symbol is the variable which holds the name of the file being visited in the current buffer, the value cell contents we see are the name of the source file of this chapter of the Emacs Lisp Manual. The property list cell contains the list `(variable-documentation 29529)` which tells the documentation functions where to find documentation about `buffer-file-name` in the `DOC'` file. (29529 is the offset from the beginning of the `DOC'` file where the documentation for the function begins.) The function cell contains the function for returning the name of the file. `buffer-file-name` names a primitive function, which has no read syntax and prints in hash notation (see section Primitive Function Type). A symbol naming a function written in Lisp would have a lambda expression (or a byte-code object) in this cell.

Defining Symbols

A definition in Lisp is a special form that announces your intention to use a certain symbol in a particular way. In Emacs Lisp, you can define a symbol as a variable, or define it as a function (or macro), or both independently.

A definition construct typically specifies a value or meaning for the symbol for one kind of use, plus documentation for its meaning when used in this way. Thus, when you define a symbol as a variable, you can supply an initial value for the variable, plus documentation for the variable.

`defvar` and `defconst` are special forms that define a symbol as a global variable. They are documented in detail in section Defining Global Variables.

`defun` defines a symbol as a function, creating a lambda expression and storing it in the function cell of the symbol. This lambda expression thus becomes the function definition of the symbol. (The term "function definition", meaning the contents of the function cell, is derived from the idea that `defun` gives the symbol its definition as a function.) See section Functions.

`defmacro` defines a symbol as a macro. It creates a macro object and stores it in the function cell of the symbol. Note that a given symbol can be a macro or a function, but not both at once, because both macro and function definitions are kept in the function cell, and that cell can hold only one Lisp object at any given time. See section Macros.

In GNU Emacs Lisp, a definition is not required in order to use a symbol as a variable or function. Thus, you can make a symbol a global variable with `setq`, whether you define it first or not. The real purpose of definitions is to guide programmers and programming tools. They inform programmers who read the code that certain symbols are *intended* to be used as variables, or as functions. In addition, utilities such as `` `etags' `` and `` `make-docfile' `` can recognize definitions, and add the appropriate information to tag tables and the `` `emacs/etc/DOC-version' `` file. See section Access to Documentation Strings.

Creating and Interning Symbols

To understand how symbols are created in GNU Emacs Lisp, you must know how Lisp reads them. Lisp must ensure that it finds the same symbol every time it reads the same set of characters. Failure to do so would cause complete confusion.

When the Lisp reader encounters a symbol, it reads all the characters of the name. Then it "hashes" those characters to find an index in a table called an obarray. Hashing is an efficient method of looking something up. For example, instead of searching a telephone book cover to cover when looking up Jan Jones, you start with the J's and go from there. That is a simple version of hashing. Each element of the obarray is a bucket which holds all the symbols with a given hash code; to look for a given name, it is sufficient to look through all the symbols in the bucket for that name's hash code.

If a symbol with the desired name is found, then it is used. If no such symbol is found, then a new symbol is created and added to the obarray bucket. Adding a symbol to an obarray is called interning it, and the symbol is then called an interned symbol. In Emacs Lisp, a symbol may be interned in only one obarray--if you try to intern the same symbol in more than one obarray, you will get unpredictable results.

It is possible for two different symbols to have the same name in different obarrays; these symbols are not `eq` or `equal`. However, this normally happens only as part of abbrev definition (see section Abbrevs And Abbrev Expansion).

Common Lisp note: in Common Lisp, a symbol may be interned in several obarrays at once.

If a symbol is not in the obarray, then there is no way for Lisp to find it when its name is read. Such a symbol is called an uninterned symbol relative to the obarray. An uninterned symbol has all the other characteristics of symbols.

In Emacs Lisp, an obarray is represented as a vector. Each element of the vector is a bucket; its value is either an interned symbol whose name hashes to that bucket, or 0 if the bucket is empty. Each interned symbol has an internal link (invisible to the user) to the next symbol in the bucket. Because these links are invisible, there is no way to scan the symbols in an obarray except using `mapatoms` (below). The order of symbols in a bucket is not significant.

In an empty obarray, every element is 0, and you can create an obarray with `(make-vector length 0)`. **This is the only valid way to create an obarray.** Prime numbers as lengths tend to result in good hashing; lengths one less than a power of two are also good.

Do not try to create an obarray that is not empty. This does not work--only `intern` can enter a symbol in an obarray properly. Also, don't try to put into an obarray of your own a symbol that is already interned in the main obarray, because in Emacs Lisp a symbol cannot be in two obarrays at once.

Most of the functions below take a name and sometimes an obarray as arguments. A `wrong-type-argument` error is signaled if the name is not a string, or if the obarray is not a vector.

Function: **symbol-name** *symbol*

This function returns the string that is symbol's name. For example:

```
(symbol-name 'foo)
     => "foo"
```

Changing the string by substituting characters, etc, does change the name of the symbol, but fails to update the obarray, so don't do it!

Function: **make-symbol** *name*

This function returns a newly-allocated, uninterned symbol whose name is name (which must be a string). Its value and function definition are void, and its property list is `nil`. In the example below, the value of `sym` is not `eq` to `foo` because it is a distinct uninterned symbol whose name is also `foo'.

```
(setq sym (make-symbol "foo"))
     => foo
(eq sym 'foo)
     => nil
```

Function: **intern** *name &optional obarray*

This function returns the interned symbol whose name is name. If there is no such symbol in the obarray, a new one is created, added to the obarray, and returned. If obarray is supplied, it specifies the obarray to

use; otherwise, the value of the global variable `obarray` is used.

```
(setq sym (intern "foo"))
     => foo
(eq sym 'foo)
     => t

(setq sym1 (intern "foo" other-obarray))
     => foo
(eq sym 'foo)
     => nil
```

Function: **intern-soft** *name &optional obarray*

This function returns the symbol whose name is name, or `nil` if a symbol with that name is not found in the obarray. Therefore, you can use `intern-soft` to test whether a symbol with a given name is interned. If obarray is supplied, it specifies the obarray to use; otherwise the value of the global variable `obarray` is used.

```
(intern-soft "frazzle")        ; No such symbol exists.
     => nil
(make-symbol "frazzle")        ; Create an uninterned one.
     => frazzle
(intern-soft "frazzle")        ; That one cannot be found.
     => nil
(setq sym (intern "frazzle"))  ; Create an interned one.
     => frazzle
(intern-soft "frazzle")        ; That one can be found!
     => frazzle
(eq sym 'frazzle)              ; And it is the same one.
     => t
```

Variable: **obarray**

This variable is the standard obarray for use by `intern` and `read`.

Function: **mapatoms** *function &optional obarray*

This function applies function to every symbol in obarray. It returns `nil`. If obarray is not supplied, it defaults to the value of `obarray`, the standard obarray for ordinary symbols.

```
(setq count 0)
     => 0
(defun count-syms (s)
  (setq count (1+ count)))
     => count-syms
(mapatoms 'count-syms)
```


```
     => nil
count
     => 1871
```

See `documentation` in section Access to Documentation Strings, for another example using `mapatoms`.

Property Lists

A property list (plist for short) is a list of paired elements stored in the property list cell of a symbol. Each of the pairs associates a property name (usually a symbol) with a property or value. Property lists are generally used to record information about a symbol, such as how to compile it, the name of the file where it was defined, or perhaps even the grammatical class of the symbol (representing a word) in a language understanding system.

Character positions in a string or buffer can also have property lists. See section Text Properties.

The property names and values in a property list can be any Lisp objects, but the names are usually symbols. They are compared using `eq`. Here is an example of a property list, found on the symbol `progn` when the compiler is loaded:

```
(lisp-indent-function 0 byte-compile byte-compile-progn)
```

Here `lisp-indent-function` and `byte-compile` are property names, and the other two elements are the corresponding values.

Association lists (see section Association Lists) are very similar to property lists. In contrast to association lists, the order of the pairs in the property list is not significant since the property names must be distinct.

Property lists are better than association lists when it is necessary to attach information to various Lisp function names or variables. If all the pairs are recorded in one association list, the program will need to search that entire list each time a function or variable is to be operated on. By contrast, if the information is recorded in the property lists of the function names or variables themselves, each search will scan only the length of one property list, which is usually short. For this reason, the documentation for a variable is recorded in a property named `variable-documentation`. The byte compiler likewise uses properties to record those functions needing special treatment.

However, association lists have their own advantages. Depending on your application, it may be faster to add an association to the front of an association list than to update a property. All properties for a symbol are stored in the same property list, so there is a possibility of a conflict between different uses of a property name. (For this reason, it is a good idea to use property names that are probably unique, such as by including the name of the library in the property name.) An association list may be used like a stack where associations are pushed on the front of the list and later discarded; this is not possible with a property list.

Function: **symbol-plist** *symbol*

This function returns the property list of symbol.

Function: **setplist** *symbol plist*

This function sets symbol's property list to plist. Normally, plist should be a well-formed property list, but this is not enforced.

```
(setplist 'foo '(a 1 b (2 3) c nil))
     => (a 1 b (2 3) c nil)
(symbol-plist 'foo)
     => (a 1 b (2 3) c nil)
```

For symbols in special obarrays, which are not used for ordinary purposes, it may make sense to use the property list cell in a nonstandard fashion; in fact, the abbrev mechanism does so (see section Abbrevs And Abbrev Expansion).

Function: **get** *symbol property*

This function finds the value of the property named property in symbol's property list. If there is no such property, `nil` is returned. Thus, there is no distinction between a value of `nil` and the absence of the property.

The name property is compared with the existing property names using `eq`, so any object is a legitimate property.

See `put` for an example.

Function: **put** *symbol property value*

This function puts value onto symbol's property list under the property name property, replacing any previous value.

```
(put 'fly 'verb 'transitive)
     =>'transitive
(put 'fly 'noun '(a buzzing little bug))
     => (a buzzing little bug)
(get 'fly 'verb)
     => transitive
(symbol-plist 'fly)
     => (verb transitive noun (a buzzing little bug))
```

Go to the previous, next section.

Go to the previous, next section.

Evaluation

The evaluation of expressions in Emacs Lisp is performed by the Lisp interpreter---a program that receives a Lisp object as input and computes its value as an expression. The value is computed in a fashion that depends on the data type of the object, following rules described in this chapter. The interpreter runs automatically to evaluate portions of your program, but can also be called explicitly via the Lisp primitive function `eval`.

A Lisp object which is intended for evaluation is called an expression or a form. The fact that expressions are data objects and not merely text is one of the fundamental differences between Lisp-like languages and typical programming languages. Any object can be evaluated, but in practice only numbers, symbols, lists and strings are evaluated very often.

It is very common to read a Lisp expression and then evaluate the expression, but reading and evaluation are separate activities, and either can be performed alone. Reading per se does not evaluate anything; it converts the printed representation of a Lisp object to the object itself. It is up to the caller of `read` whether this object is a form to be evaluated, or serves some entirely different purpose. See section Input Functions.

Do not confuse evaluation with command key interpretation. The editor command loop translates keyboard input into a command (an interactively callable function) using the active keymaps, and then uses `call-interactively` to invoke the command. The execution of the command itself involves evaluation if the command is written in Lisp, but that is not a part of command key interpretation itself. See section Command Loop.

Evaluation is a recursive process. That is, evaluation of a form may cause `eval` to be called again in order to evaluate parts of the form. For example, evaluation of a function call first evaluates each argument of the function call, and then evaluates each form in the function body. Consider evaluation of the form `(car x)`: the subform `x` must first be evaluated recursively, so that its value can be passed as an argument to the function `car`.

The evaluation of forms takes place in a context called the environment, which consists of the current values and bindings of all Lisp variables.(1) Whenever the form refers to a variable without creating a new binding for it, the value of the binding in the current environment is used. See section Variables.

Evaluation of a form may create new environments for recursive evaluation by binding variables (see section Local Variables). These environments are temporary and will be gone by the time evaluation of the form is complete. The form may also make changes that persist; these changes are called side effects. An example of a form that produces side effects is `(setq foo 1)`.

Finally, evaluation of one particular function call, `byte-code`, invokes the byte-code interpreter on its arguments. Although the byte-code interpreter is not the same as the Lisp interpreter, it uses the same environment as the Lisp interpreter, and may on occasion invoke the Lisp interpreter. (See section Byte Compilation.)

The details of what evaluation means for each kind of form are described below (see section Kinds of Forms).

Eval

Most often, forms are evaluated automatically, by virtue of their occurrence in a program being run. On rare occasions, you may need to write code that evaluates a form that is computed at run time, such as after reading a form from text being edited or getting one from a property list. On these occasions, use the `eval` function.

The functions and variables described in this section evaluate forms, specify limits to the evaluation process, or record recently returned values. Loading a file also does evaluation (see section Loading).

Function: **eval** *form*

This is the basic function for performing evaluation. It evaluates form in the current environment and returns the result. How the evaluation proceeds depends on the type of the object (see section Kinds of Forms).

Since `eval` is a function, the argument expression that appears in a call to `eval` is evaluated twice: once as preparation before `eval` is called, and again by the `eval` function itself. Here is an example:

```
(setq foo 'bar)
     => bar
(setq bar 'baz)
     => baz
;; eval receives argument bar, which is the value of foo
(eval foo)
     => baz
```

The number of currently active calls to `eval` is limited to `max-lisp-eval-depth` (see below).

Command: **eval-current-buffer** *&optional stream*

This function evaluates the forms in the current buffer. It reads forms from the buffer and calls `eval` on them until the end of the buffer is reached, or until an error is signaled and not handled.

If stream is supplied, the variable `standard-output` is bound to stream during the evaluation (see section Output Functions).

`eval-current-buffer` always returns `nil`.

Command: **eval-region** *start end &optional stream*

This function evaluates the forms in the current buffer in the region defined by the positions start and end. It reads forms from the region and calls `eval` on them until the end of the region is reached, or until an error is signaled and not handled.

If stream is supplied, `standard-output` is bound to it for the duration of the command.

`eval-region` always returns `nil`.

Variable: **max-lisp-eval-depth**

This variable defines the maximum depth allowed in calls to `eval`, `apply`, and `funcall` before an error is signaled (with error message `"Lisp nesting exceeds max-lisp-eval-depth"`). `eval` is called recursively to evaluate the arguments of Lisp function calls and to evaluate bodies of functions.

This limit, with the associated error when it is exceeded, is one way that Lisp avoids infinite recursion on an ill-defined function.

The default value of this variable is 200. If you set it to a value less than 100, Lisp will reset it to 100 if the given value is reached.

`max-specpdl-size` provides another limit on nesting. See section Local Variables.

Variable: **values**

The value of this variable is a list of values returned by all expressions which were read from buffers (including the minibuffer), evaluated, and printed. The elements are in order, most recent first.

```
(setq x 1)
     => 1
(list 'A (1+ 2) auto-save-default)
     => (A 3 t)
values
     => ((A 3 t) 1 ...)
```

This variable is useful for referring back to values of forms recently evaluated. It is generally a bad idea to print the value of `values` itself, since this may be very long. Instead, examine particular elements, like this:

```
;; Refer to the most recent evaluation result.
(nth 0 values)
     => (A 3 t)
;; That put a new element on,
;;   so all elements move back one.
(nth 1 values)
     => (A 3 t)
;; This gets the element that was next-to-last
;;   before this example.
(nth 3 values)
     => 1
```

Kinds of Forms

A Lisp object that is intended to be evaluated is called a form. How Emacs evaluates a form depends on its data type. Emacs has three different kinds of form that are evaluated differently: symbols, lists, and "all other types". All three kinds are described in this section, starting with "all other types" which are self-evaluating forms.

Self-Evaluating Forms

A self-evaluating form is any form that is not a list or symbol. Self-evaluating forms evaluate to themselves: the result of evaluation is the same object that was evaluated. Thus, the number 25 evaluates to 25, and the string `"foo"` evaluates to the string `"foo"`. Likewise, evaluation of a vector does not cause evaluation of the elements of the vector--it returns the same vector with its contents unchanged.

```
'123               ; An object, shown without evaluation.
     => 123
123                ; Evaluated as usual--result is the same.
     => 123
(eval '123)        ; Evaluated "by hand"---result is the same.
     => 123
(eval (eval '123)) ; Evaluating twice changes nothing.
     => 123
```

It is common to write numbers, characters, strings, and even vectors in Lisp code, taking advantage of the fact that they self-evaluate. However, it is quite unusual to do this for types that lack a read syntax, because it is inconvenient and not very useful; however, it is possible to put them inside Lisp programs when they are constructed from subexpressions rather than read. Here is an example:

```
;; Build such an expression.
(setq buffer (list 'print (current-buffer)))
     => (print #<buffer eval.texi>)
;; Evaluate it.
(eval buffer)
     -| #<buffer eval.texi>
     => #<buffer eval.texi>
```

Symbol Forms

When a symbol is evaluated, it is treated as a variable. The result is the variable's value, if it has one. If it has none (if its value cell is void), an error is signaled. For more information on the use of variables, see section Variables.

In the following example, we set the value of a symbol with `setq`. When the symbol is later evaluated, that value is returned.


```
(setq a 123)
     => 123
(eval 'a)
     => 123
a
     => 123
```

The symbols `nil` and `t` are treated specially, so that the value of `nil` is always `nil`, and the value of `t` is always `t`. Thus, these two symbols act like self-evaluating forms, even though `eval` treats them like any other symbol.

Classification of List Forms

A form that is a nonempty list is either a function call, a macro call, or a special form, according to its first element. These three kinds of forms are evaluated in different ways, described below. The rest of the list consists of arguments for the function, macro or special form.

The first step in evaluating a nonempty list is to examine its first element. This element alone determines what kind of form the list is and how the rest of the list is to be processed. The first element is *not* evaluated, as it would be in some Lisp dialects including Scheme.

Symbol Function Indirection

If the first element of the list is a symbol then evaluation examines the symbol's function cell, and uses its contents instead of the original symbol. If the contents are another symbol, this process, called symbol function indirection, is repeated until a non-symbol is obtained. See section Naming a Function, for more information about using a symbol as a name for a function stored in the function cell of the symbol.

One possible consequence of this process is an infinite loop, in the event that a symbol's function cell refers to the same symbol. Or a symbol may have a void function cell, causing a `void-function` error. But if neither of these things happens, we eventually obtain a non-symbol, which ought to be a function or other suitable object.

More precisely, we should now have a Lisp function (a lambda expression), a byte-code function, a primitive function, a Lisp macro, a special form, or an autoload object. Each of these types is a case described in one of the following sections. If the object is not one of these types, the error `invalid-function` is signaled.

The following example illustrates the symbol indirection process. We use `fset` to set the function cell of a symbol and `symbol-function` to get the function cell contents (see section Accessing Function Cell Contents). Specifically, we store the symbol `car` into the function cell of `first`, and the symbol `first` into the function cell of `erste`.

```
;; Build this function cell linkage:
;;   -------------       -----        -------        -------
```


```
;;  | #<subr car> | <-- | car |  <-- | first |  <-- | erste |
;;   -------------       -----        -------        -------

(symbol-function 'car)
     => #<subr car>
(fset 'first 'car)
     => car
(fset 'erste 'first)
     => first
(erste '(1 2 3))   ; Call the function referenced by erste.
     => 1
```

By contrast, the following example calls a function without any symbol function indirection, because the first element is an anonymous Lisp function, not a symbol.

```
((lambda (arg) (erste arg))
 '(1 2 3))
     => 1
```

After that function is called, its body is evaluated; this does involve symbol function indirection when calling `erste`.

The built-in function `indirect-function` provides an easy way to perform symbol function indirection explicitly.

Function: **indirect-function** *function*

This function returns the meaning of function as a function. If function is a symbol, then it finds function's function definition and starts over with that value. If function is not a symbol, then it returns function itself.

Here is how you could define `indirect-function` in Lisp:

```
(defun indirect-function (function)
  (if (symbolp function)
      (indirect-function (symbol-function function))
    function))
```

Evaluation of Function Forms

If the first element of a list being evaluated is a Lisp function object, byte-code object or primitive function object, then that list is a function call. For example, here is a call to the function `+`:

```
(+ 1 x)
```

When a function call is evaluated, the first step is to evaluate the remaining elements of the list in the order they appear. The results are the actual argument values, one argument from each element. Then the

function is called with this list of arguments, effectively using the function `apply` (see section Calli Functions). If the function is written in Lisp, the arguments are used to bind the argument variables of the function (see section Lambda Expressions); then the forms in the function body are evaluated in order, and the result of the last one is used as the value of the function call.

Lisp Macro Evaluation

If the first element of a list being evaluated is a macro object, then the list is a macro call. When a macro call is evaluated, the elements of the rest of the list are *not* initially evaluated. Instead, these elements themselves are used as the arguments of the macro. The macro definition computes a replacement form, called the expansion of the macro, which is evaluated in place of the original form. The expansion may be any sort of form: a self-evaluating constant, a symbol or a list. If the expansion is itself a macro call, this process of expansion repeats until some other sort of form results.

Normally, the argument expressions are not evaluated as part of computing the macro expansion, but instead appear as part of the expansion, so they are evaluated when the expansion is evaluated.

For example, given a macro defined as follows:

```
(defmacro cadr (x)
  (list 'car (list 'cdr x)))
```

an expression such as `(cadr (assq 'handler list))` is a macro call, and its expansion is:

```
(car (cdr (assq 'handler list)))
```

Note that the argument `(assq 'handler list)` appears in the expansion.

See section Macros, for a complete description of Emacs Lisp macros.

Special Forms

A special form is a primitive function specially marked so that its arguments are not all evaluated. Special forms define control structures or perform variable bindings--things which functions cannot do.

Each special form has its own rules for which arguments are evaluated and which are used without evaluation. Whether a particular argument is evaluated may depend on the results of evaluating other arguments.

Here is a list, in alphabetical order, of all of the special forms in Emacs Lisp with a reference to where each is described.

`and`

see section Constructs for Combining Conditions

`catch`

see section Explicit Nonlocal Exits: `catch` and `throw`

`cond`
see section Conditionals

`condition-case`
see section Writing Code to Handle Errors

`defconst`
see section Defining Global Variables

`defmacro`
see section Defining Macros

`defun`
see section Defining Named Functions

`defvar`
see section Defining Global Variables

`function`
see section Anonymous Functions

`if`
see section Conditionals

`interactive`
see section Interactive Call

`let`
`let*`
see section Local Variables

`or`
see section Constructs for Combining Conditions

`prog1`
`prog2`
`progn`
see section Sequencing

`quote`
see section Quoting

`save-excursion`
see section Excursions

`save-restriction`
see section Narrowing

`save-window-excursion`
see section Window Configurations

`setq`

see section How to Alter a Variable Value

`setq-default`

see section Creating and Destroying Buffer-local Bindings

`track-mouse`

see section Mouse Tracking

`unwind-protect`

see section Nonlocal Exits

`while`

see section Iteration

`with-output-to-temp-buffer`

see section Temporary Displays

Common Lisp note: here are some comparisons of special forms in GNU Emacs Lisp and Common Lisp. `setq`, `if`, and `catch` are special forms in both Emacs Lisp and Common Lisp. `defun` is a special form in Emacs Lisp, but a macro in Common Lisp. `save-excursion` is a special form in Emacs Lisp, but doesn't exist in Common Lisp. `throw` is a special form in Common Lisp (because it must be able to throw multiple values), but it is a function in Emacs Lisp (which doesn't have multiple values).

Autoloading

The autoload feature allows you to call a function or macro whose function definition has not yet been loaded into Emacs. When an autoload object appears as a symbol's function definition and that symbol is used as a function, Emacs will automatically install the real definition (plus other associated code) and then call that definition. (See section Autoload.)

Quoting

The special form `quote` returns its single argument "unchanged".

Special Form: **quote** *object*

This special form returns object, without evaluating it. This allows symbols and lists, which would normally be evaluated, to be included literally in a program. (It is not necessary to quote numbers, strings, and vectors since they are self-evaluating.)

Because `quote` is used so often in programs, Lisp provides a convenient read syntax for it. An apostrophe character (`'`) followed by a Lisp object (in read syntax) expands to a list whose first element is `quote`, and whose second element is the object. Thus, the read syntax `'x` is an abbreviation for `(quote x)`.

Here are some examples of expressions that use `quote`:


```
(quote (+ 1 2))
     => (+ 1 2)
(quote foo)
     => foo
'foo
     => foo
"foo
     => (quote foo)
'(quote foo)
     => (quote foo)
['foo]
     => [(quote foo)]
```

Other quoting constructs include `function` (see section Anonymous Functions), which causes an anonymous lambda expression written in Lisp to be compiled, and ` (see section Backquote), which is used to quote only part of a list, while computing and substituting other parts.

Go to the previous, next section.

Go to the previous, next section.

Control Structures

A Lisp program consists of expressions or forms (see section Kinds of Forms). We control the order of execution of the forms by enclosing them in control structures. Control structures are special forms which control when, whether, or how many times to execute the forms they contain.

The simplest control structure is sequential execution: first form a, then form b, and so on. This is what happens when you write several forms in succession in the body of a function, or at top level in a file of Lisp code--the forms are executed in the order they are written. We call this textual order. For example, if a function body consists of two forms a and b, evaluation of the function evaluates first a and then b, and the function's value is the value of b.

Naturally, Emacs Lisp has many kinds of control structures, including other varieties of sequencing, function calls, conditionals, iteration, and (controlled) jumps. The built-in control structures are special forms since their subforms are not necessarily evaluated. You can use macros to define your own control structure constructs (see section Macros).

Sequencing

Evaluating forms in the order they are written is the most common control structure. Sometimes this happens automatically, such as in a function body. Elsewhere you must use a control structure construct to do this: `progn`, the simplest control construct of Lisp.

A `progn` special form looks like this:

```
(progn a b c ...)
```

and it says to execute the forms a, b, c and so on, in that order. These forms are called the body of the `progn` form. The value of the last form in the body becomes the value of the entire `progn`.

When Lisp was young, `progn` was the only way to execute two or more forms in succession and use the value of the last of them. But programmers found they often needed to use a `progn` in the body of a function, where (at that time) only one form was allowed. So the body of a function was made into an "implicit `progn`": several forms are allowed just as in the body of an actual `progn`. Many other control structures likewise contain an implicit `progn`. As a result, `progn` is not used as often as it used to be. It is needed now most often inside of an `unwind-protect`, `and`, or `or`.

Special Form: **progn** *forms...*

This special form evaluates all of the forms, in textual order, returning the result of the final form.

```
(progn (print "The first form")
       (print "The second form")
```


```
       (print "The third form"))
     -| "The first form"
     -| "The second form"
     -| "The third form"
=> "The third form"
```

Two other control constructs likewise evaluate a series of forms but return a different value:

Special Form: **prog1** *form1 forms...*

This special form evaluates form1 and all of the forms, in textual order, returning the result of form1.

```
(prog1 (print "The first form")
       (print "The second form")
       (print "The third form"))
     -| "The first form"
     -| "The second form"
     -| "The third form"
=> "The first form"
```

Here is a way to remove the first element from a list in the variable `x`, then return the value of that former element:

```
(prog1 (car x) (setq x (cdr x)))
```

Special Form: **prog2** *form1 form2 forms...*

This special form evaluates form1, form2, and all of the following forms, in textual order, returning the result of form2.

```
(prog2 (print "The first form")
       (print "The second form")
       (print "The third form"))
     -| "The first form"
     -| "The second form"
     -| "The third form"
=> "The second form"
```

Conditionals

Conditional control structures choose among alternatives. Emacs Lisp has two conditional forms: `if`, which is much the same as in other languages, and `cond`, which is a generalized case statement.

Special Form: **if** *condition then-form else-forms...*

`if` chooses between the then-form and the else-forms based on the value of condition. If the evaluated condition is non-`nil`, then-form is evaluated and the result returned. Otherwise, the else-forms are

evaluated in textual order, and the value of the last one is returned. (The else part of `if` is an exampl an implicit `progn`. See section Sequencing.)

If condition has the value `nil`, and no else-forms are given, `if` returns `nil`.

`if` is a special form because the branch which is not selected is never evaluated--it is ignored. Thus, in the example below, `true` is not printed because `print` is never called.

```
(if nil
    (print 'true)
  'very-false)
=> very-false
```

Special Form: **cond** *clause...*

`cond` chooses among an arbitrary number of alternatives. Each clause in the `cond` must be a list. The CAR of this list is the condition; the remaining elements, if any, the body-forms. Thus, a clause looks like this:

```
(condition body-forms...)
```

`cond` tries the clauses in textual order, by evaluating the condition of each clause. If the value of condition is non-`nil`, the body-forms are evaluated, and the value of the last of body-forms becomes the value of the `cond`. The remaining clauses are ignored.

If the value of condition is `nil`, the clause "fails", so the `cond` moves on to the following clause, trying its condition.

If every condition evaluates to `nil`, so that every clause fails, `cond` returns `nil`.

A clause may also look like this:

```
(condition)
```

Then, if condition is non-`nil` when tested, the value of condition becomes the value of the `cond` form.

The following example has four clauses, which test for the cases where the value of `x` is a number, string, buffer and symbol, respectively:

```
(cond ((numberp x) x)
      ((stringp x) x)
      ((bufferp x)
       (setq temporary-hack x) ; multiple body-forms
       (buffer-name x))        ; in one clause
      ((symbolp x) (symbol-value x)))
```

Often we want the last clause to be executed whenever none of the previous clauses was successful. To do this, we use `t` as the condition of the last clause, like this: `(t body-forms)`. The form `t` evaluates

to `t`, which is never `nil`, so this clause never fails, provided the `cond` gets to it at all.

For example,

```
(cond ((eq a 1) 'foo)
      (t "default"))
=> "default"
```

This expression is a `cond` which returns `foo` if the value of `a` is 1, and returns the string `"default"` otherwise.

Both `cond` and `if` can usually be written in terms of the other. Therefore, the choice between them is a matter of taste and style. For example:

```
(if a b c)
==
(cond (a b) (t c))
```

Constructs for Combining Conditions

This section describes three constructs that are often used together with `if` and `cond` to express complicated conditions. The constructs `and` and `or` can also be used individually as kinds of multiple conditional constructs.

Function: **not** *condition*

This function tests for the falsehood of condition. It returns `t` if condition is `nil`, and `nil` otherwise. The function `not` is identical to `null`, and we recommend using `null` if you are testing for an empty list.

Special Form: **and** *conditions...*

The `and` special form tests whether all the conditions are true. It works by evaluating the conditions one by one in the order written.

If any of the conditions evaluates to `nil`, then the result of the `and` must be `nil` regardless of the remaining conditions; so the remaining conditions are ignored and the `and` returns right away.

If all the conditions turn out non-`nil`, then the value of the last of them becomes the value of the `and` form.

Here is an example. The first condition returns the integer 1, which is not `nil`. Similarly, the second condition returns the integer 2, which is not `nil`. The third condition is `nil`, so the remaining condition is never evaluated.

```
(and (print 1) (print 2) nil (print 3))
     -| 1
     -| 2
```


```
=> nil
```

Here is a more realistic example of using `and`:

```
(if (and (consp foo) (eq (car foo) 'x))
    (message "foo is a list starting with x"))
```

Note that `(car foo)` is not executed if `(consp foo)` returns `nil`, thus avoiding an error.

`and` can be expressed in terms of either `if` or `cond`. For example:

```
(and arg1 arg2 arg3)
==
(if arg1 (if arg2 arg3))
==
(cond (arg1 (cond (arg2 arg3))))
```

Special Form: **or** *conditions...*

The `or` special form tests whether at least one of the conditions is true. It works by evaluating all the conditions one by one in the order written.

If any of the conditions evaluates to a non-`nil` value, then the result of the `or` must be non-`nil`; so the remaining conditions are ignored and the `or` returns right away. The value it returns is the non-`nil` value of the condition just evaluated.

If all the conditions turn out `nil`, then the `or` expression returns `nil`.

For example, this expression tests whether `x` is either 0 or `nil`:

```
(or (eq x nil) (= x 0))
```

Like the `and` construct, `or` can be written in terms of `cond`. For example:

```
(or arg1 arg2 arg3)
==
(cond (arg1)
      (arg2)
      (arg3))
```

You could almost write `or` in terms of `if`, but not quite:

```
(if arg1 arg1
  (if arg2 arg2
    arg3))
```

This is not completely equivalent because it can evaluate arg1 or arg2 twice. By contrast, `(or arg1 arg2 arg3)` never evaluates any argument more than once.

Iteration

Iteration means executing part of a program repetitively. For example, you might want to repeat some expressions once for each element of a list, or once for each integer from 0 to n. You can do this in Emacs Lisp with the special form `while`:

Special Form: **while** *condition forms...*

`while` first evaluates condition. If the result is non-`nil`, it evaluates forms in textual order. Then it reevaluates condition, and if the result is non-`nil`, it evaluates forms again. This process repeats until condition evaluates to `nil`.

There is no limit on the number of iterations that may occur. The loop will continue until either condition evaluates to `nil` or until an error or `throw` jumps out of it (see section Nonlocal Exits).

The value of a `while` form is always `nil`.

```
(setq num 0)
     => 0
(while (< num 4)
  (princ (format "Iteration %d." num))
  (setq num (1+ num)))
     -| Iteration 0.
     -| Iteration 1.
     -| Iteration 2.
     -| Iteration 3.
     => nil
```

If you would like to execute something on each iteration before the end-test, put it together with the end-test in a `progn` as the first argument of `while`, as shown here:

```
(while (progn
         (forward-line 1)
         (not (looking-at "^$"))))
```

This moves forward one line and continues moving by lines until an empty line is reached.

Nonlocal Exits

A nonlocal exit is a transfer of control from one point in a program to another remote point. Nonlocal exits can occur in Emacs Lisp as a result of errors; you can also use them under explicit control. Nonlocal exits unbind all variable bindings made by the constructs being exited.

Explicit Nonlocal Exits: `catch` and `throw`

Most control constructs affect only the flow of control within the construct itself. The function `throw` is the exception to this rule for of normal program execution: it performs a nonlocal exit on request. (There are other exceptions, but they are for error handling only.) `throw` is used inside a `catch`, and jumps back to that `catch`. For example:

```
(catch 'foo
  (progn
    ...
      (throw 'foo t)
    ...))
```

The `throw` transfers control straight back to the corresponding `catch`, which returns immediately. The code following the `throw` is not executed. The second argument of `throw` is used as the return value of the `catch`.

The `throw` and the `catch` are matched through the first argument: `throw` searches for a `catch` whose first argument is `eq` to the one specified. Thus, in the above example, the `throw` specifies `foo`, and the `catch` specifies the same symbol, so that `catch` is applicable. If there is more than one applicable `catch`, the innermost one takes precedence.

All Lisp constructs between the `catch` and the `throw`, including function calls, are exited automatically along with the `catch`. When binding constructs such as `let` or function calls are exited in this way, the bindings are unbound, just as they are when these constructs are exited normally (see section Local Variables). Likewise, the buffer and position saved by `save-excursion` (see section Excursions) are restored, and so is the narrowing status saved by `save-restriction` and the window selection saved by `save-window-excursion` (see section Window Configurations). Any cleanups established with the `unwind-protect` special form are executed if the `unwind-protect` is exited with a `throw`.

The `throw` need not appear lexically within the `catch` that it jumps to. It can equally well be called from another function called within the `catch`. As long as the `throw` takes place chronologically after entry to the `catch`, and chronologically before exit from it, it has access to that `catch`. This is why `throw` can be used in commands such as `exit-recursive-edit` which throw back to the editor command loop (see section Recursive Editing).

> **Common Lisp note:** most other versions of Lisp, including Common Lisp, have several ways of transferring control nonsequentially: `return`, `return-from`, and `go`, for example. Emacs Lisp has only `throw`.

Special Form: **catch** *tag body...*

`catch` establishes a return point for the `throw` function. The return point is distinguished from other such return points by tag, which may be any Lisp object. The argument tag is evaluated normally before the return point is established.

With the return point in effect, the forms of the body are evaluated in textual order. If the forms execute

normally, without error or nonlocal exit, the value of the last body form is returned from the `catch`.

If a `throw` is done within body specifying the same value tag, the `catch` exits immediately; the value it returns is whatever was specified as the second argument of `throw`.

Function: **throw** *tag value*

The purpose of `throw` is to return from a return point previously established with `catch`. The argument tag is used to choose among the various existing return points; it must be `eq` to the value specified in the `catch`. If multiple return points match tag, the innermost one is used.

The argument value is used as the value to return from that `catch`.

If no return point is in effect with tag tag, then a `no-catch` error is signaled with data `(tag value)`.

Examples of `catch` and `throw`

One way to use `catch` and `throw` is to exit from a doubly nested loop. (In most languages, this would be done with a "go to".) Here we compute `(foo i j)` for i and j varying from 0 to 9:

```
(defun search-foo ()
  (catch 'loop
    (let ((i 0))
      (while (< i 10)
        (let ((j 0))
          (while (< j 10)
            (if (foo i j)
                (throw 'loop (list i j)))
            (setq j (1+ j))))
        (setq i (1+ i))))))
```

If `foo` ever returns non-`nil`, we stop immediately and return a list of i and j. If `foo` always returns `nil`, the `catch` returns normally, and the value is `nil`, since that is the result of the `while`.

Here are two tricky examples, slightly different, showing two return points at once. First, two return points with the same tag, `hack`:

```
(defun catch2 (tag)
  (catch tag
    (throw 'hack 'yes)))
=> catch2

(catch 'hack
  (print (catch2 'hack))
  'no)
-| yes
=> no
```

Since both return points have tags that match the `throw`, it goes to the inner one, the one established in `catch2`. Therefore, `catch2` returns normally with value `yes`, and this value is printed. Finally the second body form in the outer `catch`, which is `'no`, is evaluated and returned from the outer `catch`.

Now let's change the argument given to `catch2`:

```
(defun catch2 (tag)
  (catch tag
    (throw 'hack 'yes)))
=> catch2

(catch 'hack
  (print (catch2 'quux))
  'no)
=> yes
```

We still have two return points, but this time only the outer one has the tag `hack`; the inner one has the tag `quux` instead. Therefore, the `throw` returns the value `yes` from the outer return point. The function `print` is never called, and the body-form `'no` is never evaluated.

Errors

When Emacs Lisp attempts to evaluate a form that, for some reason, cannot be evaluated, it signals an error.

When an error is signaled, Emacs's default reaction is to print an error message and terminate execution of the current command. This is the right thing to do in most cases, such as if you type C-f at the end of the buffer.

In complicated programs, simple termination may not be what you want. For example, the program may have made temporary changes in data structures, or created temporary buffers which should be deleted before the program is finished. In such cases, you would use `unwind-protect` to establish cleanup expressions to be evaluated in case of error. Occasionally, you may wish the program to continue execution despite an error in a subroutine. In these cases, you would use `condition-case` to establish error handlers to recover control in case of error.

Resist the temptation to use error handling to transfer control from one part of the program to another; use `catch` and `throw`. See section Explicit Nonlocal Exits: `catch` and `throw`.

How to Signal an Error

Most errors are signaled "automatically" within Lisp primitives which you call for other purposes, such as if you try to take the CAR of an integer or move forward a character at the end of the buffer; you can also signal errors explicitly with the functions `error` and `signal`.

Quitting, which happens when the user types C-g, is not considered an error, but it handled almost like an error. See section Quitting.

Function: **error** *format-string &rest args*

This function signals an error with an error message constructed by applying `format` (see section Conversion of Characters and Strings) to format-string and args.

Typical uses of `error` is shown in the following examples:

```
(error "You have committed an error.
        Try something else.")
     error--> You have committed an error.
        Try something else.

(error "You have committed %d errors." 10)
     error--> You have committed 10 errors.
```

`error` works by calling `signal` with two arguments: the error symbol `error`, and a list containing the string returned by `format`.

If you want to use a user-supplied string as an error message verbatim, don't just write `(error string)`. If string contains `%', it will be interpreted as a format specifier, with undesirable results. Instead, use `(error "%s" string)`.

Function: **signal** *error-symbol data*

This function signals an error named by error-symbol. The argument data is a list of additional Lisp objects relevant to the circumstances of the error.

The argument error-symbol must be an error symbol---a symbol bearing a property `error-conditions` whose value is a list of condition names. This is how different sorts of errors are classified.

The number and significance of the objects in data depends on error-symbol. For example, with a `wrong-type-arg` error, there are two objects in the list: a predicate which describes the type that was expected, and the object which failed to fit that type. See section Error Symbols and Condition Names, for a description of error symbols.

Both error-symbol and data are available to any error handlers which handle the error: a list `(error-symbol . data)` is constructed to become the value of the local variable bound in the `condition-case` form (see section Writing Code to Handle Errors). If the error is not handled, both of them are used in printing the error message.

The function `signal` never returns (though in older Emacs versions it could sometimes return).

```
(signal 'wrong-number-of-arguments '(x y))
     error--> Wrong number of arguments: x, y

(signal 'no-such-error '("My unknown error condition."))
     error--> peculiar error: "My unknown error condition."
```

Common Lisp note: Emacs Lisp has nothing like the Common Lisp concept of continuable errors.

How Emacs Processes Errors

When an error is signaled, Emacs searches for an active handler for the error. A handler is a specially marked place in the Lisp code of the current function or any of the functions by which it was called. If an applicable handler exists, its code is executed, and control resumes following the handler. The handler executes in the environment of the `condition-case` which established it; all functions called within that `condition-case` have already been exited, and the handler cannot return to them.

If no applicable handler is in effect in your program, the current command is terminated and control returns to the editor command loop, because the command loop has an implicit handler for all kinds of errors. The command loop's handler uses the error symbol and associated data to print an error message.

When an error is not handled explicitly, it may cause the Lisp debugger to be called. The debugger is enabled if the variable `debug-on-error` (see section Entering the Debugger on an Error) is non-`nil`. Unlike error handlers, the debugger runs in the environment of the error, so that you can examine values of variables precisely as they were at the time of the error.

Writing Code to Handle Errors

The usual effect of signaling an error is to terminate the command that is running and return immediately to the Emacs editor command loop. You can arrange to trap errors occurring in a part of your program by establishing an error handler with the special form `condition-case`. A simple example looks like this:

```
(condition-case nil
    (delete-file filename)
  (error nil))
```

This deletes the file named filename, catching any error and returning `nil` if an error occurs.

The second argument of `condition-case` is called the protected form. (In the example above, the protected form is a call to `delete-file`.) The error handlers go into effect when this form begins execution and are deactivated when this form returns. They remain in effect for all the intervening time. In particular, they are in effect during the execution of subroutines called by this form, and their subroutines, and so on. This is a good thing, since, strictly speaking, errors can be signaled only by Lisp primitives (including `signal` and `error`) called by the protected form, not by the protected form itself.

The arguments after the protected form are handlers. Each handler lists one or more condition names (which are symbols) to specify which errors it will handle. The error symbol specified when an error is signaled also defines a list of condition names. A handler applies to an error if they have any condition names in common. In the example above, there is one handler, and it specifies one condition name, `error`, which covers all errors.

The search for an applicable handler checks all the established handlers starting with the most recently established one. Thus, if two nested `condition-case` forms try to handle the same error, the inner of

the two will actually handle it.

When an error is handled, control returns to the handler. Before this happens, Emacs unbinds all variable bindings made by binding constructs that are being exited and executes the cleanups of all `unwind-protect` forms that are exited. Once control arrives at the handler, the body of the handler is executed.

After execution of the handler body, execution continues by returning from the `condition-case` form. Because the protected form is exited completely before execution of the handler, the handler cannot resume execution at the point of the error, nor can it examine variable bindings that were made within the protected form. All it can do is clean up and proceed.

`condition-case` is often used to trap errors that are predictable, such as failure to open a file in a call to `insert-file-contents`. It is also used to trap errors that are totally unpredictable, such as when the program evaluates an expression read from the user.

Error signaling and handling have some resemblance to `throw` and `catch`, but they are entirely separate facilities. An error cannot be caught by a `catch`, and a `throw` cannot be handled by an error handler (though using `throw` when there is no suitable `catch` signals an error which can be handled).

Special Form: **condition-case** *var protected-form handlers...*

This special form establishes the error handlers handlers around the execution of protected-form. If protected-form executes without error, the value it returns becomes the value of the `condition-case` form; in this case, the `condition-case` has no effect. The `condition-case` form makes a difference when an error occurs during protected-form.

Each of the handlers is a list of the form `(conditions body...)`. conditions is an error condition name to be handled, or a list of condition names; body is one or more Lisp expressions to be executed when this handler handles an error. Here are examples of handlers:

```
(error nil)

(arith-error (message "Division by zero"))

((arith-error file-error)
 (message
  "Either division by zero or failure to open a file"))
```

Each error that occurs has an error symbol which describes what kind of error it is. The `error-conditions` property of this symbol is a list of condition names (see section Error Symbols and Condition Names). Emacs searches all the active `condition-case` forms for a handler which specifies one or more of these names; the innermost matching `condition-case` handles the error. The handlers in this `condition-case` are tested in the order in which they appear.

The body of the handler is then executed, and the `condition-case` returns normally, using the value of the last form in the body as the overall value.

The argument var is a variable. `condition-case` does not bind this variable when executing the

protected-form, only when it handles an error. At that time, var is bound locally to a list of the form `(error-symbol . data)`, giving the particulars of the error. The handler can refer to this list to decide what to do. For example, if the error is for failure opening a file, the file name is the second element of data---the third element of var.

If var is `nil`, that means no variable is bound. Then the error symbol and associated data are not made available to the handler.

Here is an example of using `condition-case` to handle the error that results from dividing by zero. The handler prints out a warning message and returns a very large number.

```
(defun safe-divide (dividend divisor)
  (condition-case err
      ;; Protected form.
      (/ dividend divisor)
    ;; The handler.
    (arith-error                        ; Condition.
     (princ (format "Arithmetic error: %s" err))
     1000000)))
=> safe-divide

(safe-divide 5 0)
     -| Arithmetic error: (arith-error)
=> 1000000
```

The handler specifies condition name `arith-error` so that it will handle only division-by-zero errors. Other kinds of errors will not be handled, at least not by this `condition-case`. Thus,

```
(safe-divide nil 3)
     error--> Wrong type argument: integer-or-marker-p, nil
```

Here is a `condition-case` that catches all kinds of errors, including those signaled with `error`:

```
(setq baz 34)
     => 34

(condition-case err
    (if (eq baz 35)
        t
      ;; This is a call to the function error.
      (error "Rats!  The variable %s was %s, not 35." 'baz baz))
  ;; This is the handler; it is not a form.
  (error (princ (format "The error was: %s" err))
         2))
-| The error was: (error "Rats!  The variable baz was 34, not 35.")
=> 2
```

Error Symbols and Condition Names

When you signal an error, you specify an error symbol to specify the kind of error you have in mind. Each error has one and only one error symbol to categorize it. This is the finest classification of errors defined by the Lisp language.

These narrow classifications are grouped into a hierarchy of wider classes called error conditions, identified by condition names. The narrowest such classes belong to the error symbols themselves: each error symbol is also a condition name. There are also condition names for more extensive classes, up to the condition name `error` which takes in all kinds of errors. Thus, each error has one or more condition names: `error`, the error symbol if that is distinct from `error`, and perhaps some intermediate classifications.

In order for a symbol to be usable as an error symbol, it must have an `error-conditions` property which gives a list of condition names. This list defines the conditions which this kind of error belongs to. (The error symbol itself, and the symbol `error`, should always be members of this list.) Thus, the hierarchy of condition names is defined by the `error-conditions` properties of the error symbols.

In addition to the `error-conditions` list, the error symbol should have an `error-message` property whose value is a string to be printed when that error is signaled but not handled. If the `error-message` property exists, but is not a string, the error message `peculiar error' is used.

Here is how we define a new error symbol, `new-error`:

```
(put 'new-error
     'error-conditions
     '(error my-own-errors new-error))
     => (error my-own-errors new-error)
(put 'new-error 'error-message "A new error")
     => "A new error"
```

This error has three condition names: `new-error`, the narrowest classification; `my-own-errors`, which we imagine is a wider classification; and `error`, which is the widest of all. Naturally, Emacs will never signal a `new-error` on its own; only an explicit call to `signal` (see section Errors) in your code can do this:

```
(signal 'new-error '(x y))
     error--> A new error: x, y
```

This error can be handled through any of the three condition names. This example handles `new-error` and any other errors in the class `my-own-errors`:

```
(condition-case foo
    (bar nil t)
  (my-own-errors nil))
```

The significant way that errors are classified is by their condition names--the names used to match errors

with handlers. An error symbol serves only as a convenient way to specify the intended error messag and list of condition names. If `signal` were given a list of condition names rather than one error symbol, that would be cumbersome.

By contrast, using only error symbols without condition names would seriously decrease the power of `condition-case`. Condition names make it possible to categorize errors at various levels of generality when you write an error handler. Using error symbols alone would eliminate all but the narrowest level of classification.

See section Standard Errors, for a list of all the standard error symbols and their conditions.

Cleaning Up from Nonlocal Exits

The `unwind-protect` construct is essential whenever you temporarily put a data structure in an inconsistent state; it permits you to ensure the data are consistent in the event of an error or throw.

Special Form: **unwind-protect** *body cleanup-forms...*

`unwind-protect` executes the body with a guarantee that the cleanup-forms will be evaluated if control leaves body, no matter how that happens. The body may complete normally, or execute a `throw` out of the `unwind-protect`, or cause an error; in all cases, the cleanup-forms will be evaluated.

Only the body is actually protected by the `unwind-protect`. If any of the cleanup-forms themselves exit nonlocally (e.g., via a `throw` or an error), it is *not* guaranteed that the rest of them will be executed. If the failure of one of the cleanup-forms has the potential to cause trouble, then it should be protected by another `unwind-protect` around that form.

The number of currently active `unwind-protect` forms counts, together with the number of local variable bindings, against the limit `max-specpdl-size` (see section Local Variables).

For example, here we make an invisible buffer for temporary use, and make sure to kill it before finishing:

```
(save-excursion
  (let ((buffer (get-buffer-create " *temp*")))
    (set-buffer buffer)
    (unwind-protect
        body
      (kill-buffer buffer))))
```

You might think that we could just as well write `(kill-buffer (current-buffer))` and dispense with the variable `buffer`. However, the way shown above is safer, if body happens to get an error after switching to a different buffer! (Alternatively, you could write another `save-excursion` around the body, to ensure that the temporary buffer becomes current in time to kill it.)

Here is an actual example taken from the file `` `ftp.el' ``. It creates a process (see section Processes) to try to establish a connection to a remote machine. As the function `ftp-login` is highly susceptible to numerous problems which the writer of the function cannot anticipate, it is protected with a form that

guarantees deletion of the process in the event of failure. Otherwise, Emacs might fill up with useless subprocesses.

```
(let ((win nil))
  (unwind-protect
      (progn
        (setq process (ftp-setup-buffer host file))
        (if (setq win (ftp-login process host user password))
            (message "Logged in")
          (error "Ftp login failed")))
    (or win (and process (delete-process process)))))
```

This example actually has a small bug: if the user types C-g to quit, and the quit happens immediately after the function `ftp-setup-buffer` returns but before the variable `process` is set, the process will not be killed. There is no easy way to fix this bug, but at least it is very unlikely.

Go to the previous, next section.

Go to the previous, next section.

Variables

A variable is a name used in a program to stand for a value. Nearly all programming languages have variables of some sort. In the text for a Lisp program, variables are written using the syntax for symbols.

In Lisp, unlike most programming languages, programs are represented primarily as Lisp objects and only secondarily as text. The Lisp objects used for variables are symbols: the symbol name is the variable name, and the variable's value is stored in the value cell of the symbol. The use of a symbol as a variable is independent of whether the same symbol has a function definition. See section Symbol Components.

The textual form of a program is determined by its Lisp object representation; it is the read syntax for the Lisp object which constitutes the program. This is why a variable in a textual Lisp program is written as the read syntax for the symbol that represents the variable.

Global Variables

The simplest way to use a variable is globally. This means that the variable has just one value at a time, and this value is in effect (at least for the moment) throughout the Lisp system. The value remains in effect until you specify a new one. When a new value replaces the old one, no trace of the old value remains in the variable.

You specify a value for a symbol with `setq`. For example,

```
(setq x '(a b))
```

gives the variable `x` the value `(a b)`. Note that the first argument of `setq`, the name of the variable, is not evaluated, but the second argument, the desired value, is evaluated normally.

Once the variable has a value, you can refer to it by using the symbol by itself as an expression. Thus,

```
x
     => (a b)
```

assuming the `setq` form shown above has already been executed.

If you do another `setq`, the new value replaces the old one:

```
x
     => (a b)
(setq x 4)
     => 4
x
     => 4
```

Variables that Never Change

Emacs Lisp has two special symbols, `nil` and `t`, that always evaluate to themselves. These symbols cannot be rebound, nor can their value cells be changed. An attempt to change the value of `nil` or `t` signals a `setting-constant` error.

```
nil == 'nil
     => nil
(setq nil 500)
error--> Attempt to set constant symbol: nil
```

Local Variables

Global variables are given values that last until explicitly superseded with new values. Sometimes it is useful to create variable values that exist temporarily--only while within a certain part of the program. These values are called local, and the variables so used are called local variables.

For example, when a function is called, its argument variables receive new local values which last until the function exits. Similarly, the `let` special form explicitly establishes new local values for specified variables; these last until exit from the `let` form.

When a local value is established, the previous value (or lack of one) of the variable is saved away. When the life span of the local value is over, the previous value is restored. In the mean time, we say that the previous value is shadowed and not visible. Both global and local values may be shadowed (see section Scope).

If you set a variable (such as with `setq`) while it is local, this replaces the local value; it does not alter the global value, or previous local values that are shadowed. To model this behavior, we speak of a local binding of the variable as well as a local value.

The local binding is a conceptual place that holds a local value. Entry to a function, or a special form such as `let`, creates the local binding; exit from the function or from the `let` removes the local binding. As long as the local binding lasts, the variable's value is stored within it. Use of `setq` or `set` while there is a local binding stores a different value into the local binding; it does not create a new binding.

We also speak of the global binding, which is where (conceptually) the global value is kept.

A variable can have more than one local binding at a time (for example, if there are nested `let` forms that bind it). In such a case, the most recently created local binding that still exists is the current binding of the variable. (This is called dynamic scoping; see section Scoping Rules for Variable Bindings.) If there are no local bindings, the variable's global binding is its current binding. We also call the current binding the most-local existing binding, for emphasis. Ordinary evaluation of a symbol always returns the value of its current binding.

The special forms `let` and `let*` exist to create local bindings.

Special Form: **let** *(bindings...) forms...*

This function binds variables according to bindings and then evaluates all of the forms in textual order. The `let`-form returns the value of the last form in forms.

Each of the bindings is either (i) a symbol, in which case that symbol is bound to `nil`; or (ii) a list of the form `(symbol value-form)`, in which case symbol is bound to the result of evaluating value-form. If value-form is omitted, `nil` is used.

All of the value-forms in bindings are evaluated in the order they appear and *before* any of the symbols are bound. Here is an example of this: `Z` is bound to the old value of `Y`, which is 2, not the new value, 1.

```
(setq Y 2)
     => 2
(let ((Y 1)
      (Z Y))
  (list Y Z))
     => (1 2)
```

Special Form: **let*** *(bindings...) forms...*

This special form is like `let`, except that each symbol in bindings is bound as soon as its new value is computed, before the computation of the values of the following local bindings. Therefore, an expression in bindings may reasonably refer to the preceding symbols bound in this `let*` form. Compare the following example with the example above for `let`.

```
(setq Y 2)
     => 2
(let* ((Y 1)
       (Z Y))    ; Use the just-established value of Y.
  (list Y Z))
     => (1 1)
```

Here is a complete list of the other facilities which create local bindings:

- Function calls (see section Functions).
- Macro calls (see section Macros).
- `condition-case` (see section Errors).

Variable: **max-specpdl-size**

This variable defines the limit on the total number of local variable bindings and `unwind-protect` cleanups (see section Nonlocal Exits) that are allowed before signaling an error (with data `"Variable binding depth exceeds max-specpdl-size"`).

This limit, with the associated error when it is exceeded, is one way that Lisp avoids infinite recursion on an ill-defined function.

The default value is 600.

`max-lisp-eval-depth` provides another limit on depth of nesting. See section Eval.

When a Variable is "Void"

If you have never given a symbol any value as a global variable, we say that that symbol's global value is void. In other words, the symbol's value cell does not have any Lisp object in it. If you try to evaluate the symbol, you get a `void-variable` error rather than a value.

Note that a value of `nil` is not the same as void. The symbol `nil` is a Lisp object and can be the value of a variable just as any other object can be; but it is *a value*. A void variable does not have any value.

After you have given a variable a value, you can make it void once more using `makunbound`.

Function: **makunbound** *symbol*

This function makes the current binding of symbol void. This causes any future attempt to use this symbol as a variable to signal the error `void-variable`, unless or until you set it again.

`makunbound` returns symbol.

```
(makunbound 'x)      ; Make the global value
                     ;   of x void.
     => x
x
error--> Symbol's value as variable is void: x
```

If symbol is locally bound, `makunbound` affects the most local existing binding. This is the only way a symbol can have a void local binding, since all the constructs that create local bindings create them with values. In this case, the voidness lasts at most as long as the binding does; when the binding is removed due to exit from the construct that made it, the previous or global binding is reexposed as usual, and the variable is no longer void unless the newly reexposed binding was void all along.

```
(setq x 1)               ; Put a value in the global binding.
     => 1
(let ((x 2))             ; Locally bind it.
  (makunbound 'x)        ; Void the local binding.
  x)
error--> Symbol's value as variable is void: x
x                        ; The global binding is unchanged.
     => 1

(let ((x 2))             ; Locally bind it.
  (let ((x 3))           ; And again.
    (makunbound 'x)      ; Void the innermost-local binding.
    x))                  ; And refer: it's void.
```


```
error--> Symbol's value as variable is void: x

(let ((x 2))
  (let ((x 3))
    (makunbound 'x))     ; Void inner binding, then remove it.
  x)                     ; Now outer let binding is visible.
     => 2
```

A variable that has been made void with `makunbound` is indistinguishable from one that has never received a value and has always been void.

You can use the function `boundp` to test whether a variable is currently void.

Function: **boundp** *variable*

`boundp` returns `t` if variable (a symbol) is not void; more precisely, if its current binding is not void. It returns `nil` otherwise.

```
(boundp 'abracadabra)          ; Starts out void.
     => nil
(let ((abracadabra 5))         ; Locally bind it.
  (boundp 'abracadabra))
     => t
(boundp 'abracadabra)          ; Still globally void.
     => nil
(setq abracadabra 5)           ; Make it globally nonvoid.
     => 5
(boundp 'abracadabra)
     => t
```

Defining Global Variables

You may announce your intention to use a symbol as a global variable with a definition, using `defconst` or `defvar`.

In Emacs Lisp, definitions serve three purposes. First, they inform the user who reads the code that certain symbols are *intended* to be used as variables. Second, they inform the Lisp system of these things, supplying a value and documentation. Third, they provide information to utilities such as `etags` and `make-docfile`, which create data bases of the functions and variables in a program.

The difference between `defconst` and `defvar` is primarily a matter of intent, serving to inform human readers of whether programs will change the variable. Emacs Lisp does not restrict the ways in which a variable can be used based on `defconst` or `defvar` declarations. However, it also makes a difference for initialization: `defconst` unconditionally initializes the variable, while `defvar` initializes it only if it is void.

One would expect user option variables to be defined with `defconst`, since programs do not change

them. Unfortunately, this has bad results if the definition is in a library that is not preloaded: `defcu` would override any prior value when the library is loaded. Users would like to be able to set the opti their init files, and override the default value given in the definition. For this reason, user options must be defined with `defvar`.

Special Form: **defvar** *symbol [value [doc-string]]*

This special form informs a person reading your code that symbol will be used as a variable that the programs are likely to set or change. It is also used for all user option variables except in the preloaded parts of Emacs. Note that symbol is not evaluated; the symbol to be defined must appear explicitly in the `defvar`.

If symbol already has a value (i.e., it is not void), value is not even evaluated, and symbol's value remains unchanged. If symbol is void and value is specified, it is evaluated and symbol is set to the result. (If value is not specified, the value of symbol is not changed in any case.)

If symbol has a buffer-local binding in the current buffer, `defvar` sets the default value, not the local value.

If the doc-string argument appears, it specifies the documentation for the variable. (This opportunity to specify documentation is one of the main benefits of defining the variable.) The documentation is stored in the symbol's `variable-documentation` property. The Emacs help functions (see section Documentation) look for this property.

If the first character of doc-string is `*', it means that this variable is considered to be a user option. This affects commands such as `set-variable` and `edit-options`.

For example, this form defines `foo` but does not set its value:

```
(defvar foo)
     => foo
```

The following example sets the value of `bar` to `23`, and gives it a documentation string:

```
(defvar bar 23
  "The normal weight of a bar.")
     => bar
```

The following form changes the documentation string for `bar`, making it a user option, but does not change the value, since `bar` already has a value. (The addition `(1+ 23)` is not even performed.)

```
(defvar bar (1+ 23)
  "*The normal weight of a bar.")
     => bar
bar
     => 23
```

Here is an equivalent expression for the `defvar` special form:


```
(defvar symbol value doc-string)
==
(progn
  (if (not (boundp 'symbol))
      (setq symbol value))
  (put 'symbol 'variable-documentation 'doc-string)
  'symbol)
```

The `defvar` form returns symbol, but it is normally used at top level in a file where its value does not matter.

Special Form: **defconst** *symbol [value [doc-string]]*

This special form informs a person reading your code that symbol has a global value, established here, that will not normally be changed or locally bound by the execution of the program. The user, however, may be welcome to change it. Note that symbol is not evaluated; the symbol to be defined must appear explicitly in the `defconst`.

`defconst` always evaluates value and sets the global value of symbol to the result, provided value is given. If symbol has a buffer-local binding in the current buffer, `defconst` sets the default value, not the local value.

Please note: don't use `defconst` for user option variables in libraries that are not normally loaded. The user should be able to specify a value for such a variable in the `` `.emacs' `` file, so that it will be in effect if and when the library is loaded later.

Here, `pi` is a constant that presumably ought not to be changed by anyone (attempts by the Indiana State Legislature notwithstanding). As the second form illustrates, however, this is only advisory.

```
(defconst pi 3 "Pi to one place.")
     => pi
(setq pi 4)
     => pi
pi
     => 4
```

Function: **user-variable-p** *variable*

This function returns `t` if variable is a user option, intended to be set by the user for customization, `nil` otherwise. (Variables other than user options exist for the internal purposes of Lisp programs, and users need not know about them.)

User option variables are distinguished from other variables by the first character of the `variable-documentation` property. If the property exists and is a string, and its first character is `*', then the variable is a user option.

Note that if the `defconst` and `defvar` special forms are used while the variable has a local binding, the local binding's value is set, and the global binding is not changed. This would be confusing. But the

normal way to use these special forms is at top level in a file, where no local binding should be in eff

Accessing Variable Values

The usual way to reference a variable is to write the symbol which names it (see section Symbol Forms). This requires you to specify the variable name when you write the program. Usually that is exactly what you want to do. Occasionally you need to choose at run time which variable to reference; then you can use `symbol-value`.

Function: **symbol-value** *symbol*

This function returns the value of symbol. This is the value in the innermost local binding of the symbol, or its global value if it has no local bindings.

```
(setq abracadabra 5)
     => 5
(setq foo 9)
     => 9

;; Here the symbol abracadabra
;;   is the symbol whose value is examined.
(let ((abracadabra 'foo))
  (symbol-value 'abracadabra))
     => foo

;; Here the value of abracadabra,
;;   which is foo,
;;   is the symbol whose value is examined.
(let ((abracadabra 'foo))
  (symbol-value abracadabra))
     => 9

(symbol-value 'abracadabra)
     => 5
```

A `void-variable` error is signaled if symbol has neither a local binding nor a global value.

How to Alter a Variable Value

The usual way to change the value of a variable is with the special form `setq`. When you need to compute the choice of variable at run time, use the function `set`.

Special Form: **setq** *[symbol form]...*

This special form is the most common method of changing a variable's value. Each symbol is given a

new value, which is the result of evaluating the corresponding form. The most-local existing binding of the symbol is changed.

The value of the `setq` form is the value of the last form.

```
(setq x (1+ 2))
     => 3
x                    ; x now has a global value.
     => 3
(let ((x 5))
  (setq x 6)         ; The local binding of x is set.
  x)
     => 6
x                    ; The global value is unchanged.
     => 3
```

Note that the first form is evaluated, then the first symbol is set, then the second form is evaluated, then the second symbol is set, and so on:

```
(setq x 10           ; Notice that x is set before
      y (1+ x))      ;   the value of y is computed.
     => 11
```

Function: **set** *symbol value*

This function sets symbol's value to value, then returns value. Since `set` is a function, the expression written for symbol is evaluated to obtain the symbol to be set.

The most-local existing binding of the variable is the binding that is set; shadowed bindings are not affected. If symbol is not actually a symbol, a `wrong-type-argument` error is signaled.

```
(set one 1)
error--> Symbol's value as variable is void: one
(set 'one 1)
     => 1
(set 'two 'one)
     => one
(set two 2)          ; two evaluates to symbol one.
     => 2
one                  ; So it is one that was set.
     => 2
(let ((one 1))       ; This binding of one is set,
  (set 'one 3)       ;   not the global value.
  one)
     => 3
one
     => 2
```

Logically speaking, `set` is a more fundamental primitive that `setq`. Any use of `setq` can be trivially rewritten to use `set`; `setq` could even be defined as a macro, given the availability of `set`. However, `set` itself is rarely used; beginners hardly need to know about it. It is needed only when the choice of variable to be set is made at run time. For example, the command `set-variable`, which reads a variable name from the user and then sets the variable, needs to use `set`.

> **Common Lisp note:** in Common Lisp, `set` always changes the symbol's special value, ignoring any lexical bindings. In Emacs Lisp, all variables and all bindings are special, so `set` always affects the most local existing binding.

Scoping Rules for Variable Bindings

A given symbol `foo` may have several local variable bindings, established at different places in the Lisp program, as well as a global binding. The most recently established binding takes precedence over the others.

Local bindings in Emacs Lisp have indefinite scope and dynamic extent. Scope refers to *where* textually in the source code the binding can be accessed. Indefinite scope means that any part of the program can potentially access the variable binding. Extent refers to *when*, as the program is executing, the binding exists. Dynamic extent means that the binding lasts as long as the activation of the construct that established it.

The combination of dynamic extent and indefinite scope is called dynamic scoping. By contrast, most programming languages use lexical scoping, in which references to a local variable must be textually within the function or block that binds the variable.

> **Common Lisp note:** variables declared "special" in Common Lisp are dynamically scoped like variables in Emacs Lisp.

Scope

Emacs Lisp uses indefinite scope for local variable bindings. This means that any function anywhere in the program text might access a given binding of a variable. Consider the following function definitions:

```
(defun binder (x)   ; x is bound in binder.
  (foo 5))          ; foo is some other function.

(defun user ()      ; x is used in user.
  (list x))
```

In a lexically scoped language, the binding of `x` from `binder` would never be accessible in `user`, because `user` is not textually contained within the function `binder`. However, in dynamically scoped Emacs Lisp, `user` may or may not refer to the binding of `x` established in `binder`, depending on circumstances:

- If we call `user` directly without calling `binder` at all, then whatever binding of `x` is found, it cannot come from `binder`.

- If we define `foo` as follows and call `binder`, then the binding made in `binder` will be seen in `user`:

```
(defun foo (lose)
  (user))
```

- If we define `foo` as follows and call `binder`, then the binding made in `binder` *will not* be seen in `user`:

```
(defun foo (x)
  (user))
```

  Here, when `foo` is called by `binder`, it binds `x`. (The binding in `foo` is said to shadow the one made in `binder`.) Therefore, `user` will access the `x` bound by `foo` instead of the one bound by `binder`.

Extent

Extent refers to the time during program execution that a variable name is valid. In Emacs Lisp, a variable is valid only while the form that bound it is executing. This is called dynamic extent. "Local" or "automatic" variables in most languages, including C and Pascal, have dynamic extent.

One alternative to dynamic extent is indefinite extent. This means that a variable binding can live on past the exit from the form that made the binding. Common Lisp and Scheme, for example, support this, but Emacs Lisp does not.

To illustrate this, the function below, `make-add`, returns a function that purports to add n to its own argument m. This would work in Common Lisp, but it does not work as intended in Emacs Lisp, because after the call to `make-add` exits, the variable `n` is no longer bound to the actual argument 2.

```
(defun make-add (n)
    (function (lambda (m) (+ n m))))  ; Return a function.
     => make-add
(fset 'add2 (make-add 2))  ; Define function add2
                           ;   with (make-add 2).
     => (lambda (m) (+ n m))
(add2 4)                   ; Try to add 2 to 4.
error--> Symbol's value as variable is void: n
```

Implementation of Dynamic Scoping

A simple sample implementation (which is not how Emacs Lisp actually works) may help you understand dynamic binding. This technique is called deep binding and was used in early Lisp systems.

Suppose there is a stack of bindings: variable-value pairs. At entry to a function or to a `let` form, we can push bindings on the stack for the arguments or local variables created there. We can pop those bindings from the stack at exit from the binding construct.

We can find the value of a variable by searching the stack from top to bottom for a binding for that variable; the value from that binding is the value of the variable. To set the variable, we search for the current binding, then store the new value into that binding.

As you can see, a function's bindings remain in effect as long as it continues execution, even during its calls to other functions. That is why we say the extent of the binding is dynamic. And any other function can refer to the bindings, if it uses the same variables while the bindings are in effect. That is why we say the scope is indefinite.

The actual implementation of variable scoping in GNU Emacs Lisp uses a technique called shallow binding. Each variable has a standard place in which its current value is always found--the value cell of the symbol.

In shallow binding, setting the variable works by storing a value in the value cell. When a new local binding is created, the local value is stored in the value cell, and the old value (belonging to a previous binding) is pushed on a stack. When a binding is eliminated, the old value is popped off the stack and stored in the value cell.

We use shallow binding because it has the same results as deep binding, but runs faster, since there is never a need to search for a binding.

Proper Use of Dynamic Scoping

Binding a variable in one function and using it in another is a powerful technique, but if used without restraint, it can make programs hard to understand. There are two clean ways to use this technique:

- Use or bind the variable only in a few related functions, written close together in one file. Such a variable is used for communication within one program.

  You should write comments to inform other programmers that they can see all uses of the variable before them, and to advise them not to add uses elsewhere.
- Give the variable a well-defined, documented meaning, and make all appropriate functions refer to it (but not bind it or set it) wherever that meaning is relevant. For example, the variable `case-fold-search` is defined as "non-`nil` means ignore case when searching"; various search and replace functions refer to it directly or through their subroutines, but do not bind or set it.

  Then you can bind the variable in other programs, knowing reliably what the effect will be.

Buffer-Local Variables

Global and local variable bindings are found in most programming languages in one form or another. Emacs also supports another, unusual kind of variable binding: buffer-local bindings, which apply only to one buffer. Emacs Lisp is meant for programming editing commands, and having different values for a variable in different buffers is an important customization method.

Introduction to Buffer-Local Variables

A buffer-local variable has a buffer-local binding associated with a particular buffer. The binding is in effect when that buffer is current; otherwise, it is not in effect. If you set the variable while a buffer-local binding is in effect, the new value goes in that binding, so the global binding is unchanged; this means that the change is visible in that buffer alone.

A variable may have buffer-local bindings in some buffers but not in others. The global binding is shared by all the buffers that don't have their own bindings. Thus, if you set the variable in a buffer that does not have a buffer-local binding for it, the new value is visible in all buffers except those with buffer-local bindings. (Here we are assuming that there are no `let`-style local bindings to complicate the issue.)

The most common use of buffer-local bindings is for major modes to change variables that control the behavior of commands. For example, C mode and Lisp mode both set the variable `paragraph-start` to specify that only blank lines separate paragraphs. They do this by making the variable buffer-local in the buffer that is being put into C mode or Lisp mode, and then setting it to the new value for that mode.

The usual way to make a buffer-local binding is with `make-local-variable`, which is what major mode commands use. This affects just the current buffer; all other buffers (including those yet to be created) continue to share the global value.

A more powerful operation is to mark the variable as automatically buffer-local by calling `make-variable-buffer-local`. You can think of this as making the variable local in all buffers, even those yet to be created. More precisely, the effect is that setting the variable automatically makes the variable local to the current buffer if it is not already so. All buffers start out by sharing the global value of the variable as usual, but any `setq` creates a buffer-local binding for the current buffer. The new value is stored in the buffer-local binding, leaving the (default) global binding untouched. The global value can no longer be changed with `setq`; you need to use `setq-default` to do that.

Warning: when a variable has local values in one or more buffers, you can get Emacs very confused by binding the variable with `let`, changing to a different current buffer in which a different binding is in effect, and then exiting the `let`. To preserve your sanity, it is wise to avoid such situations. If you use `save-excursion` around each piece of code that changes to a different current buffer, you will not have this problem. Here is an example of incorrect code:

```
(setq foo 'b)
(set-buffer "a")
(make-local-variable 'foo)
(setq foo 'a)
(let ((foo 'temp))
  (set-buffer "b")
  ...)
foo => 'a       ; The old buffer-local value from buffer `a'
                ;   is now the default value.
(set-buffer "a")
foo => 'temp    ; The local value that should be gone
                ;   is now the buffer-local value in buffer `a'.
```

But `save-excursion` as shown here avoids the problem:

```
(let ((foo 'temp))
  (save-excursion
    (set-buffer "b")
    ...))
```

Local variables in a file you edit are also represented by buffer-local bindings for the buffer that holds the file within Emacs. See section How Emacs Chooses a Major Mode.

Creating and Destroying Buffer-local Bindings

Command: **make-local-variable** *variable*

This function creates a buffer-local binding in the current buffer for variable (a symbol). Other buffers are not affected. The value returned is variable.

The buffer-local value of variable starts out as the same value variable previously had. If variable was void, it remains void.

```
;; In buffer `b1':
(setq foo 5)                ; Affects all buffers.
     => 5
(make-local-variable 'foo)  ; Now it is local in `b1'.
     => foo
foo                         ; That did not change
     => 5                   ;   the value.
(setq foo 6)                ; Change the value
     => 6                   ;   in `b1'.
foo
     => 6

;; In buffer `b2', the value hasn't changed.
(save-excursion
  (set-buffer "b2")
  foo)
     => 5
```

Command: **make-variable-buffer-local** *variable*

This function marks variable (a symbol) automatically buffer-local, so that any attempt to set it will make it local to the current buffer at the time.

The value returned is variable.

Function: **buffer-local-variables** *&optional buffer*

This function tells you what the buffer-local variables are in buffer buffer. It returns an association li
(see section Association Lists) in which each association contains one buffer-local variable and its va
If buffer is omitted, the current buffer is used.

```
(setq lcl (buffer-local-variables))
=> ((fill-column . 75)
    (case-fold-search . t)
    ...
    (mark-ring #<marker at 5454 in buffers.texi>)
    (require-final-newline . t))
```

Note that storing new values into the CDRs of the elements in this list does *not* change the local values of the variables.

Command: **kill-local-variable** *variable*

This function deletes the buffer-local binding (if any) for variable (a symbol) in the current buffer. As a result, the global (default) binding of variable becomes visible in this buffer. Usually this results in a change in the value of variable, since the global value is usually different from the buffer-local value just eliminated.

It is possible to kill the local binding of a variable that automatically becomes local when set. This causes the variable to show its global value in the current buffer. However, if you set the variable again, this will once again create a local value.

`kill-local-variable` returns variable.

Function: **kill-all-local-variables**

This function eliminates all the buffer-local variable bindings of the current buffer except for variables marker as "permanent". As a result, the buffer will see the default values of most variables.

This function also resets certain other information pertaining to the buffer: its local keymap is set to `nil`, its syntax table is set to the value of `standard-syntax-table`, and its abbrev table is set to the value of `fundamental-mode-abbrev-table`.

Every major mode command begins by calling this function, which has the effect of switching to Fundamental mode and erasing most of the effects of the previous major mode. To ensure that this does its job, the variables that major modes set should not be marked permanent.

`kill-all-local-variables` returns `nil`.

A local variable is permanent if the variable name (a symbol) has a `permanent-local` property that is non-`nil`. Permanent locals are appropriate for data pertaining to where the file came from or how to save it, rather than with how to edit the contents.

The Default Value of a Buffer-Local Variable

The global value of a variable with buffer-local bindings is also called the default value, because it is the value that is in effect except when specifically overridden.

The functions `default-value` and `setq-default` allow you to access and change the default value regardless of whether the current buffer has a buffer-local binding. For example, you could use `setq-default` to change the default setting of `paragraph-start` for most buffers; and this would work even when you are in a C or Lisp mode buffer which has a buffer-local value for this variable.

The special forms `defvar` and `defconst` also set the default value (if they set the variable at all), rather than any local value.

Function: **default-value** *symbol*

This function returns symbol's default value. This is the value that is seen in buffers that do not have their own values for this variable. If symbol is not buffer-local, this is equivalent to `symbol-value` (see section Accessing Variable Values).

Function: **default-boundp** *variable*

The function `default-boundp` tells you whether variable's default value is nonvoid. If `(default-boundp 'foo)` returns `nil`, then `(default-value 'foo)` would get an error.

`default-boundp` is to `default-value` as `boundp` is to `symbol-value`.

Special Form: **setq-default** *symbol value*

This sets the default value of symbol to value. symbol is not evaluated, but value is. The value of the `setq-default` form is value.

If a symbol is not buffer-local for the current buffer, and is not marked automatically buffer-local, this has the same effect as `setq`. If symbol is buffer-local for the current buffer, then this changes the value that other buffers will see (as long as they don't have a buffer-local value), but not the value that the current buffer sees.

```
;; In buffer `foo':
(make-local-variable 'local)
     => local
(setq local 'value-in-foo)
     => value-in-foo
(setq-default local 'new-default)
     => new-default
local
     => value-in-foo
(default-value 'local)
     => new-default

;; In (the new) buffer `bar':
```


```
local
     => new-default
(default-value 'local)
     => new-default
(setq local 'another-default)
     => another-default
(default-value 'local)
     => another-default

;; Back in buffer `foo':
local
     => value-in-foo
(default-value 'local)
     => another-default
```

Function: **set-default** *symbol value*

This function is like `setq-default`, except that symbol is evaluated.

```
(set-default (car '(a b c)) 23)
     => 23
(default-value 'a)
     => 23
```

Go to the previous, next section.

Go to the previous, next section.

Functions

A Lisp program is composed mainly of Lisp functions. This chapter explains what functions are, how they accept arguments, and how to define them.

What Is a Function?

In a general sense, a function is a rule for carrying on a computation given several values called arguments. The result of the computation is called the value of the function. The computation can also have side effects: lasting changes in the values of variables or the contents of data structures.

Here are important terms for functions in Emacs Lisp and for other function-like objects.

function

In Emacs Lisp, a function is anything that can be applied to arguments in a Lisp program. In some cases, we use it more specifically to mean a function written in Lisp. Special forms and macros are not functions.

primitive

A primitive is a function callable from Lisp that is written in C, such as `car` or `append`. These functions are also called built-in functions or subrs. (Special forms are also considered primitives.)

Usually the reason that a function is a primitives is because it is fundamental, or provides a low-level interface to operating system services, or because it needs to run fast. Primitives can be modified or added only by changing the C sources and recompiling the editor. See section Writing Emacs Primitives.

lambda expression

A lambda expression is a function written in Lisp. These are described in the following section.

special form

A special form is a primitive that is like a function but does not evaluate all of its arguments in the usual way. It may evaluate only some of the arguments, or may evaluate them in an unusual order, or several times. Many special forms are described in section Control Structures.

macro

A macro is a construct defined in Lisp by the programmer. It differs from a function in that it translates a Lisp expression that you write into an equivalent expression to be evaluated instead of the original expression. See section Macros, for how to define and use macros.

command

A command is an object that `command-execute` can invoke; it is a possible definition for a key sequence. Some functions are commands; a function written in Lisp is a command if it contains an interactive declaration (see section Defining Commands). Such a function can be called from Lisp

expressions like other functions; in this case, the fact that the function is a command makes no difference.

Strings are commands also, even though they are not functions. A symbol is a command if its function definition is a command; such symbols can be invoked with M-x. The symbol is a function as well if the definition is a function. See section Command Loop Overview.

keystroke command

A keystroke command is a command that is bound to a key sequence (typically one to three keystrokes). The distinction is made here merely to avoid confusion with the meaning of "command" in non-Emacs editors; for programmers, the distinction is normally unimportant.

byte-code function

A byte-code function is a function that has been compiled by the byte compiler. See section Byte-Code Function Type.

Function: **subrp** *object*

This function returns `t` if object is a built-in function (i.e. a Lisp primitive).

```
(subrp 'message)            ; message is a symbol,
     => nil                 ;   not a subr object.
(subrp (symbol-function 'message))
     => t
```

Function: **byte-code-function-p** *object*

This function returns `t` if object is a byte-code function. For example:

```
(byte-code-function-p (symbol-function 'next-line))
     => t
```

Lambda Expressions

A function written in Lisp is a list that looks like this:

```
(lambda (arg-variables...)
  [documentation-string]
  [interactive-declaration]
  body-forms...)
```

(Such a list is called a lambda expression for historical reasons, even though it is not really an expression at all--it is not a form that can be evaluated meaningfully.)

Components of a Lambda Expression

The first element of a lambda expression is always the symbol `lambda`. This indicates that the list represents a function. The reason functions are defined to start with `lambda` is so that other lists, intended for other uses, will not accidentally be valid as functions.

The second element is a list of argument variable names (symbols). This is called the lambda list. When a Lisp function is called, the argument values are matched up against the variables in the lambda list, which are given local bindings with the values provided. See section Local Variables.

The documentation string is an actual string that serves to describe the function for the Emacs help facilities. See section Documentation Strings of Functions.

The interactive declaration is a list of the form `(interactive code-string)`. This declares how to provide arguments if the function is used interactively. Functions with this declaration are called commands; they can be called using M-x or bound to a key. Functions not intended to be called in this way should not have interactive declarations. See section Defining Commands, for how to write an interactive declaration.

The rest of the elements are the body of the function: the Lisp code to do the work of the function (or, as a Lisp programmer would say, "a list of Lisp forms to evaluate"). The value returned by the function is the value returned by the last element of the body.

A Simple Lambda-Expression Example

Consider for example the following function:

```
(lambda (a b c) (+ a b c))
```

We can call this function by writing it as the CAR of an expression, like this:

```
((lambda (a b c) (+ a b c))
 1 2 3)
```

The body of this lambda expression is evaluated with the variable `a` bound to 1, `b` bound to 2, and `c` bound to 3. Evaluation of the body adds these three numbers, producing the result 6; therefore, this call to the function returns the value 6.

Note that the arguments can be the results of other function calls, as in this example:

```
((lambda (a b c) (+ a b c))
 1 (* 2 3) (- 5 4))
```

Here all the arguments `1`, `(* 2 3)`, and `(- 5 4)` are evaluated, left to right. Then the lambda expression is applied to the argument values 1, 6 and 1 to produce the value 8.

It is not often useful to write a lambda expression as the CAR of a form in this way. You can get the

same result, of making local variables and giving them values, using the special form `let` (see sectio Local Variables). And `let` is clearer and easier to use. In practice, lambda expressions are either sto as the function definitions of symbols, to produce named functions, or passed as arguments to other functions (see section Anonymous Functions).

However, calls to explicit lambda expressions were very useful in the old days of Lisp, before the special form `let` was invented. At that time, they were the only way to bind and initialize local variables.

Advanced Features of Argument Lists

Our simple sample function, `(lambda (a b c) (+ a b c))`, specifies three argument variables, so it must be called with three arguments: if you try to call it with only two arguments or four arguments, you get a `wrong-number-of-arguments` error.

It is often convenient to write a function that allows certain arguments to be omitted. For example, the function `substring` accepts three arguments--a string, the start index and the end index--but the third argument defaults to the end of the string if you omit it. It is also convenient for certain functions to accept an indefinite number of arguments, as the functions `and` and `+` do.

To specify optional arguments that may be omitted when a function is called, simply include the keyword `&optional` before the optional arguments. To specify a list of zero or more extra arguments, include the keyword `&rest` before one final argument.

Thus, the complete syntax for an argument list is as follows:

```
(required-vars...
 [&optional optional-vars...]
 [&rest rest-var])
```

The square brackets indicate that the `&optional` and `&rest` clauses, and the variables that follow them, are optional.

A call to the function requires one actual argument for each of the required-vars. There may be actual arguments for zero or more of the optional-vars, and there cannot be any more actual arguments than these unless `&rest` exists. In that case, there may be any number of extra actual arguments.

If actual arguments for the optional and rest variables are omitted, then they always default to `nil`. However, the body of the function is free to consider `nil` an abbreviation for some other meaningful value. This is what `substring` does; `nil` as the third argument means to use the length of the string supplied. There is no way for the function to distinguish between an explicit argument of `nil` and an omitted argument.

> **Common Lisp note:** Common Lisp allows the function to specify what default value to use when an optional argument is omitted; GNU Emacs Lisp always uses `nil`.

For example, an argument list that looks like this:

```
(a b &optional c d &rest e)
```

binds `a` and `b` to the first two actual arguments, which are required. If one or two more arguments are provided, `c` and `d` are bound to them respectively; any arguments after the first four are collected into a list and `e` is bound to that list. If there are only two arguments, `c` is `nil`; if two or three arguments, `d` is `nil`; if four arguments or fewer, `e` is `nil`.

There is no way to have required arguments following optional ones--it would not make sense. To see why this must be so, suppose that `c` in the example were optional and `d` were required. If three actual arguments are given; then which variable would the third argument be for? Similarly, it makes no sense to have any more arguments (either required or optional) after a `&rest` argument.

Here are some examples of argument lists and proper calls:

```
((lambda (n) (1+ n))                    ; One required:
 1)                                     ; requires exactly one argument.
     => 2
((lambda (n &optional n1)               ; One required and one optional:
         (if n1 (+ n n1) (1+ n)))       ; 1 or 2 arguments.
 1 2)
     => 3
((lambda (n &rest ns)                   ; One required and one rest:
         (+ n (apply '+ ns)))           ; 1 or more arguments.
 1 2 3 4 5)
     => 15
```

Documentation Strings of Functions

A lambda expression may optionally have a documentation string just after the lambda list. This string does not affect execution of the function; it is a kind of comment, but a systematized comment which actually appears inside the Lisp world and can be used by the Emacs help facilities. See section Documentation, for how the documentation-string is accessed.

It is a good idea to provide documentation strings for all commands, and for all other functions in your program that users of your program should know about; internal functions might as well have only comments, since comments don't take up any room when your program is loaded.

The first line of the documentation string should stand on its own, because `apropos` displays just this first line. It should consist of one or two complete sentences that summarize the function's purpose.

The start of the documentation string is usually indented, but since these spaces come before the starting double-quote, they are not part of the string. Some people make a practice of indenting any additional lines of the string so that the text lines up. *This is a mistake.* The indentation of the following lines is inside the string; what looks nice in the source code will look ugly when displayed by the help commands.

You may wonder how the documentation string could be optional, since there are required components of the function that follow it (the body). Since evaluation of a string returns that string, without any side effects, it has no effect if it is not the last form in the body. Thus, in practice, there is no confusion

between the first form of the body and the documentation string; if the only body form is a string the serves both as the return value and as the documentation.

Naming a Function

In most computer languages, every function has a name; the idea of a function without a name is nonsensical. In Lisp, a function in the strictest sense has no name. It is simply a list whose first element is `lambda`, or a primitive subr-object.

However, a symbol can serve as the name of a function. This happens when you put the function in the symbol's function cell (see section Symbol Components). Then the symbol itself becomes a valid, callable function, equivalent to the list or subr-object that its function cell refers to. The contents of the function cell are also called the symbol's function definition. When the evaluator finds the function definition to use in place of the symbol, we call that symbol function indirection; see section Symbol Function Indirection.

In practice, nearly all functions are given names in this way and referred to through their names. For example, the symbol `car` works as a function and does what it does because the primitive subr-object `#<subr car>` is stored in its function cell.

We give functions names because it is more convenient to refer to them by their names in other functions. For primitive subr-objects such as `#<subr car>`, names are the only way you can refer to them: there is no read syntax for such objects. For functions written in Lisp, the name is more convenient to use in a call than an explicit lambda expression. Also, a function with a name can refer to itself--it can be recursive. Writing the function's name in its own definition is much more convenient than making the function definition point to itself (something that is not impossible but that has various disadvantages in practice).

Functions are often identified with the symbols used to name them. For example, we often speak of "the function `car`", not distinguishing between the symbol `car` and the primitive subr-object that is its function definition. For most purposes, there is no need to distinguish.

Even so, keep in mind that a function need not have a unique name. While a given function object *usually* appears in the function cell of only one symbol, this is just a matter of convenience. It is easy to store it in several symbols using `fset`; then each of the symbols is equally well a name for the same function.

A symbol used as a function name may also be used as a variable; these two uses of a symbol are independent and do not conflict.

Defining Named Functions

We usually give a name to a function when it is first created. This is called defining a function, and it is done with the `defun` special form.

Special Form: **defun** *name argument-list body-forms*

`defun` is the usual way to define new Lisp functions. It defines the symbol name as a function that l like this:

```
(lambda argument-list . body-forms)
```

This lambda expression is stored in the function cell of name. The value returned by evaluating the `defun` form is name, but usually we ignore this value.

As described previously (see section Lambda Expressions), argument-list is a list of argument names and may include the keywords `&optional` and `&rest`. Also, the first two forms in body-forms may be a documentation string and an interactive declaration.

Note that the same symbol name may also be used as a global variable, since the value cell is independent of the function cell.

Here are some examples:

```
(defun foo () 5)
     => foo
(foo)
     => 5

(defun bar (a &optional b &rest c)
    (list a b c))
     => bar
(bar 1 2 3 4 5)
     => (1 2 (3 4 5))
(bar 1)
     => (1 nil nil)
(bar)
error--> Wrong number of arguments.

(defun capitalize-backwards ()
  "Upcase the last letter of a word."
  (interactive)
  (backward-word 1)
  (forward-word 1)
  (backward-char 1)
  (capitalize-word 1))
     => capitalize-backwards
```

Be careful not to redefine existing functions unintentionally. `defun` redefines even primitive functions such as `car` without any hesitation or notification. Redefining a function already defined is often done deliberately, and there is no way to distinguish deliberate redefinition from unintentional redefinition.

Calling Functions

Defining functions is only half the battle. Functions don't do anything until you call them, i.e., tell them to run. This process is also known as invocation.

The most common way of invoking a function is by evaluating a list. For example, evaluating the list `(concat "a" "b")` calls the function `concat`. See section Evaluation, for a description of evaluation.

When you write a list as an expression in your program, the function name is part of the program. This means that the choice of which function to call is made when you write the program. Usually that's just what you want. Occasionally you need to decide at run time which function to call. Then you can use the functions `funcall` and `apply`.

Function: **funcall** *function &rest arguments*

`funcall` calls function with arguments, and returns whatever function returns.

Since `funcall` is a function, all of its arguments, including function, are evaluated before `funcall` is called. This means that you can use any expression to obtain the function to be called. It also means that `funcall` does not see the expressions you write for the arguments, only their values. These values are *not* evaluated a second time in the act of calling function; `funcall` enters the normal procedure for calling a function at the place where the arguments have already been evaluated.

The argument function must be either a Lisp function or a primitive function. Special forms and macros are not allowed, because they make sense only when given the "unevaluated" argument expressions. `funcall` cannot provide these because, as we saw above, it never knows them in the first place.

```
(setq f 'list)
     => list
(funcall f 'x 'y 'z)
     => (x y z)
(funcall f 'x 'y '(z))
     => (x y (z))
(funcall 'and t nil)
error--> Invalid function: #<subr and>
```

Compare this example with that of `apply`.

Function: **apply** *function &rest arguments*

`apply` calls function with arguments, just like `funcall` but with one difference: the last of arguments is a list of arguments to give to function, rather than a single argument. We also say that this list is appended to the other arguments.

`apply` returns the result of calling function. As with `funcall`, function must either be a Lisp function or a primitive function; special forms and macros do not make sense in `apply`.


```
(setq f 'list)
     => list
(apply f 'x 'y 'z)
error--> Wrong type argument: listp, z
(apply '+ 1 2 '(3 4))
     => 10
(apply '+ '(1 2 3 4))
     => 10

(apply 'append '((a b c) nil (x y z) nil))
     => (a b c x y z)
```

An interesting example of using `apply` is found in the description of `mapcar`; see the following section.

It is common for Lisp functions to accept functions as arguments or find them in data structures (especially in hook variables and property lists) and call them using `funcall` or `apply`. Functions that accept function arguments are often called functionals.

Sometimes, when you call such a function, it is useful to supply a no-op function as the argument. Here are two different kinds of no-op function:

Function: **identity** *arg*

This function returns arg and has no side effects.

Function: **ignore** *&rest args*

This function ignores any arguments and returns `nil`.

Mapping Functions

A mapping function applies a given function to each element of a list or other collection. Emacs Lisp has three such functions; `mapcar` and `mapconcat`, which scan a list, are described here. For the third mapping function, `mapatoms`, see section Creating and Interning Symbols.

Function: **mapcar** *function sequence*

`mapcar` applies function to each element of sequence in turn. The results are made into a `nil`-terminated list.

The argument sequence may be a list, a vector or a string. The result is always a list. The length of the result is the same as the length of sequence.

```
For example:

(mapcar 'car '((a b) (c d) (e f)))
     => (a c e)
```


```
(mapcar '1+ [1 2 3])
     => (2 3 4)
(mapcar 'char-to-string "abc")
     => ("a" "b" "c")

;; Call each function in my-hooks.
(mapcar 'funcall my-hooks)

(defun mapcar* (f &rest args)
  "Apply FUNCTION to successive cars of all ARGS, until one ends.
Return the list of results."
  ;; If no list is exhausted,
  (if (not (memq 'nil args))
      ;; Apply function to CARs.
      (cons (apply f (mapcar 'car args))
            (apply 'mapcar* f
                   ;; Recurse for rest of elements.
                   (mapcar 'cdr args)))))

(mapcar* 'cons '(a b c) '(1 2 3 4))
     => ((a . 1) (b . 2) (c . 3))
```

Function: **mapconcat** *function sequence separator*

`mapconcat` applies function to each element of sequence: the results, which must be strings, are concatenated. Between each pair of result strings, `mapconcat` inserts the string separator. Usually separator contains a space or comma or other suitable punctuation.

The argument function must be a function that can take one argument and returns a string.

```
(mapconcat 'symbol-name
           '(The cat in the hat)
           " ")
     => "The cat in the hat"

(mapconcat (function (lambda (x) (format "%c" (1+ x))))
           "HAL-8000"
           "")
     => "IBM.9111"
```

Anonymous Functions

In Lisp, a function is a list that starts with `lambda` (or alternatively a primitive subr-object); names are "extra". Although usually functions are defined with `defun` and given names at the same time, it is occasionally more concise to use an explicit lambda expression--an anonymous function. Such a list is valid wherever a function name is.

Any method of creating such a list makes a valid function. Even this:

```
(setq silly (append '(lambda (x)) (list (list '+ (* 3 4) 'x))))
     => (lambda (x) (+ 12 x))
```

This computes a list that looks like `(lambda (x) (+ 12 x))` and makes it the value (*not* the function definition!) of `silly`.

Here is how we might call this function:

```
(funcall silly 1)
     => 13
```

(It does *not* work to write `(silly 1)`, because this function is not the *function definition* of `silly`. We have not given `silly` any function definition, just a value as a variable.)

Most of the time, anonymous functions are constants that appear in your program. For example, you might want to pass one as an argument to the function `mapcar`, which applies any given function to each element of a list. Here we pass an anonymous function that multiplies a number by two:

```
(defun double-each (list)
  (mapcar '(lambda (x) (* 2 x)) list))
     => double-each
(double-each '(2 11))
     => (4 22)
```

In such cases, we usually use the special form `function` instead of simple quotation to quote the anonymous function.

Special Form: **function** *function-object*

This special form returns function-object without evaluating it. In this, it is equivalent to `quote`. However, it serves as a note to the Emacs Lisp compiler that function-object is intended to be used only as a function, and therefore can safely be compiled. See section Quoting, for comparison.

Using `function` instead of `quote` makes a difference inside a function or macro that you are going to compile. For example:

```
(defun double-each (list)
  (mapcar (function (lambda (x) (* 2 x))) list))
     => double-each
(double-each '(2 11))
     => (4 22)
```

If this definition of `double-each` is compiled, the anonymous function is compiled as well. By contrast, in the previous definition where ordinary `quote` is used, the argument passed to `mapcar` is the precise list shown:


```
(lambda (arg) (+ arg 5))
```

The Lisp compiler cannot assume this list is a function, even though it looks like one, since it does not know what `mapcar` does with the list. Perhaps `mapcar` will check that the CAR of the third element is the symbol `+`! The advantage of `function` is that it tells the compiler to go ahead and compile the constant function.

We sometimes write `function` instead of `quote` when quoting the name of a function, but this usage is just a sort of comment.

```
(function symbol) == (quote symbol) == 'symbol
```

See `documentation` in section Access to Documentation Strings, for a realistic example using `function` and an anonymous function.

Accessing Function Cell Contents

The function definition of a symbol is the object stored in the function cell of the symbol. The functions described here access, test, and set the function cell of symbols.

Function: **symbol-function** *symbol*

This returns the object in the function cell of symbol. If the symbol's function cell is void, a `void-function` error is signaled.

This function does not check that the returned object is a legitimate function.

```
(defun bar (n) (+ n 2))
     => bar
(symbol-function 'bar)
     => (lambda (n) (+ n 2))
(fset 'baz 'bar)
     => bar
(symbol-function 'baz)
     => bar
```

If you have never given a symbol any function definition, we say that that symbol's function cell is void. In other words, the function cell does not have any Lisp object in it. If you try to call such a symbol as a function, it signals a `void-function` error.

Note that void is not the same as `nil` or the symbol `void`. The symbols `nil` and `void` are Lisp objects, and can be stored into a function cell just as any other object can be (and they can be valid functions if you define them in turn with `defun`); but `nil` or `void` is *an object*. A void function cell contains no object whatsoever.

You can test the voidness of a symbol's function definition with `fboundp`. After you have given a

symbol a function definition, you can make it void once more using `fmakunbound`.

Function: **fboundp** *symbol*

Returns `t` if the symbol has an object in its function cell, `nil` otherwise. It does not check that the object is a legitimate function.

Function: **fmakunbound** *symbol*

This function makes symbol's function cell void, so that a subsequent attempt to access this cell will cause a `void-function` error. (See also `makunbound`, in section Local Variables.)

```
(defun foo (x) x)
     => x
(fmakunbound 'foo)
     => x
(foo 1)
error--> Symbol's function definition is void: foo
```

Function: **fset** *symbol object*

This function stores object in the function cell of symbol. The result is object. Normally object should be a function or the name of a function, but this is not checked.

There are three normal uses of this function:

- Copying one symbol's function definition to another. (In other words, making an alternate name for a function.)
- Giving a symbol a function definition that is not a list and therefore cannot be made with `defun`. See section Classification of List Forms, for an example of this usage.
- In constructs for defining or altering functions. If `defun` were not a primitive, it could be written in Lisp (as a macro) using `fset`.

Here are examples of the first two uses:

```
;; Give first the same definition car has.
(fset 'first (symbol-function 'car))
     => #<subr car>
(first '(1 2 3))
     => 1

;; Make the symbol car the function definition of xfirst.
(fset 'xfirst 'car)
     => car
(xfirst '(1 2 3))
     => 1
(symbol-function 'xfirst)
     => car
```


```
(symbol-function (symbol-function 'xfirst))
     => #<subr car>

;; Define a named keyboard macro.
(fset 'kill-two-lines "\^u2\^k")
     => "\^u2\^k"
```

When writing a function that extends a previously defined function, the following idiom is often used:

```
(fset 'old-foo (symbol-function 'foo))

(defun foo ()
  "Just like old-foo, except more so."
  (old-foo)
  (more-so))
```

This does not work properly if `foo` has been defined to autoload. In such a case, when `foo` calls `old-foo`, Lisp attempts to define `old-foo` by loading a file. Since this presumably defines `foo` rather than `old-foo`, it does not produce the proper results. The only way to avoid this problem is to make sure the file is loaded before moving aside the old definition of `foo`.

See also the function `indirect-function` in section Symbol Function Indirection.

Inline Functions

You can define an inline function by using `defsubst` instead of `defun`. An inline function works just like an ordinary function except for one thing: when you compile a call to the function, the function's definition is open-coded into the caller.

Making a function inline makes explicit calls run faster. But it also has disadvantages. For one thing, it reduces flexibility; if you change the definition of the function, calls already inlined still use the old definition until you recompile them.

Another disadvantage is that making a large function inline can increase the size of compiled code both in files and in memory. Since the advantages of inline functions are greatest for small functions, you generally should not make large functions inline.

It's possible to define a macro to expand into the same code that an inline function would execute. But the macro would have a limitation: you can use it only explicitly--a macro cannot be called with `apply`, `mapcar` and so on. Also, it takes some work to convert an ordinary function into a macro. (See section Macros.) To convert it into an inline function is very easy; simply replace `defun` with `defsubst`.

Inline functions can be used and open coded later on in the same file, following the definition, just like macros.

Emacs versions prior to 19 did not have inline functions.

Other Topics Related to Functions

Here is a table of several functions that do things related to function calling and function definitions. They are documented elsewhere, but we provide cross references here.

`apply`
: See section Calling Functions.

`autoload`
: See section Autoload.

`call-interactively`
: See section Interactive Call.

`commandp`
: See section Interactive Call.

`documentation`
: See section Access to Documentation Strings.

`eval`
: See section Eval.

`funcall`
: See section Calling Functions.

`ignore`
: See section Calling Functions.

`indirect-function`
: See section Symbol Function Indirection.

`interactive`
: See section Using `interactive`.

`interactive-p`
: See section Interactive Call.

`mapatoms`
: See section Creating and Interning Symbols.

`mapcar`
: See section Mapping Functions.

`mapconcat`
: See section Mapping Functions.

`undefined`
: See section Key Lookup.

Go to the previous, next section.

Go to the previous, next section.

Macros

Macros enable you to define new control constructs and other language features. A macro is defined much like a function, but instead of telling how to compute a value, it tells how to compute another Lisp expression which will in turn compute the value. We call this expression the expansion of the macro.

Macros can do this because they operate on the unevaluated expressions for the arguments, not on the argument values as functions do. They can therefore construct an expansion containing these argument expressions or parts of them.

If you are using a macro to do something an ordinary function could do, just for the sake of speed, consider using an inline function instead. See section Inline Functions.

A Simple Example of a Macro

Suppose we would like to define a Lisp construct to increment a variable value, much like the `++` operator in C. We would like to write `(inc x)` and have the effect of `(setq x (1+ x))`. Here's a macro definition that does the job:

```
(defmacro inc (var)
   (list 'setq var (list '1+ var)))
```

When this is called with `(inc x)`, the argument `var` has the value `x`---*not* the *value* of `x`. The body of the macro uses this to construct the expansion, which is `(setq x (1+ x))`. Once the macro definition returns this expansion, Lisp proceeds to evaluate it, thus incrementing `x`.

Expansion of a Macro Call

A macro call looks just like a function call in that it is a list which starts with the name of the macro. The rest of the elements of the list are the arguments of the macro.

Evaluation of the macro call begins like evaluation of a function call except for one crucial difference: the macro arguments are the actual expressions appearing in the macro call. They are not evaluated before they are given to the macro definition. By contrast, the arguments of a function are results of evaluating the elements of the function call list.

Having obtained the arguments, Lisp invokes the macro definition just as a function is invoked. The argument variables of the macro are bound to the argument values from the macro call, or to a list of them in the case of a `&rest` argument. And the macro body executes and returns its value just as a function body does.

The second crucial difference between macros and functions is that the value returned by the macro body

is not the value of the macro call. Instead, it is an alternate expression for computing that value, also known as the expansion of the macro. The Lisp interpreter proceeds to evaluate the expansion as soon it comes back from the macro.

Since the expansion is evaluated in the normal manner, it may contain calls to other macros. It may even be a call to the same macro, though this is unusual.

You can see the expansion of a given macro call by calling `macroexpand`.

Function: **macroexpand** *form &optional environment*

This function expands form, if it is a macro call. If the result is another macro call, it is expanded in turn, until something which is not a macro call results. That is the value returned by `macroexpand`. If form is not a macro call to begin with, it is returned as given.

Note that `macroexpand` does not look at the subexpressions of form (although some macro definitions may do so). Even if they are macro calls themselves, `macroexpand` does not expand them.

The function `macroexpand` does not expand calls to inline functions. Normally there is no need for that, since a call to an inline function is no harder to understand than a call to an ordinary function.

If environment is provided, it specifies an alist of macro definitions that shadow the currently defined macros. This is used by byte compilation.

```
(defmacro inc (var)
    (list 'setq var (list '1+ var)))
     => inc

(macroexpand '(inc r))
     => (setq r (1+ r))

(defmacro inc2 (var1 var2)
    (list 'progn (list 'inc var1) (list 'inc var2)))
     => inc2

(macroexpand '(inc2 r s))
     => (progn (inc r) (inc s))  ; inc not expanded here.
```

Macros and Byte Compilation

You might ask why we take the trouble to compute an expansion for a macro and then evaluate the expansion. Why not have the macro body produce the desired results directly? The reason has to do with compilation.

When a macro call appears in a Lisp program being compiled, the Lisp compiler calls the macro definition just as the interpreter would, and receives an expansion. But instead of evaluating this expansion, it compiles the expansion as if it had appeared directly in the program. As a result, the

compiled code produces the value and side effects intended for the macro, but executes at full compi speed. This would not work if the macro body computed the value and side effects itself--they woul computed at compile time, which is not useful.

In order for compilation of macro calls to work, the macros must be defined in Lisp when the calls to them are compiled. The compiler has a special feature to help you do this: if a file being compiled contains a `defmacro` form, the macro is defined temporarily for the rest of the compilation of that file. To use this feature, you must define the macro in the same file where it is used and before its first use.

While byte-compiling a file, any `require` calls at top-level are executed. One way to ensure that necessary macro definitions are available during compilation is to require the file that defines them. See section Features.

Defining Macros

A Lisp macro is a list whose CAR is `macro`. Its CDR should be a function; expansion of the macro works by applying the function (with `apply`) to the list of unevaluated argument-expressions from the macro call.

It is possible to use an anonymous Lisp macro just like an anonymous function, but this is never done, because it does not make sense to pass an anonymous macro to mapping functions such as `mapcar`. In practice, all Lisp macros have names, and they are usually defined with the special form `defmacro`.

Special Form: **defmacro** *name argument-list body-forms...*

`defmacro` defines the symbol name as a macro that looks like this:

```
(macro lambda argument-list . body-forms)
```

This macro object is stored in the function cell of name. The value returned by evaluating the `defmacro` form is name, but usually we ignore this value.

The shape and meaning of argument-list is the same as in a function, and the keywords `&rest` and `&optional` may be used (see section Advanced Features of Argument Lists). Macros may have a documentation string, but any `interactive` declaration is ignored since macros cannot be called interactively.

Backquote

It could prove rather awkward to write macros of significant size, simply due to the number of times the function `list` needs to be called. To make writing these forms easier, a macro ``' (often called backquote) exists.

Backquote allows you to quote a list, but selectively evaluate elements of that list. In the simplest case, it is identical to the special form `quote` (see section Quoting). For example, these two forms yield identical results:


```
(` (a list of (+ 2 3) elements))
     => (a list of (+ 2 3) elements)
(quote (a list of (+ 2 3) elements))
     => (a list of (+ 2 3) elements)
```

By inserting a special marker, `,', inside of the argument to backquote, it is possible to evaluate desired portions of the argument:

```
(list 'a 'list 'of (+ 2 3) 'elements)
     => (a list of 5 elements)
(` (a list of (, (+ 2 3)) elements))
     => (a list of 5 elements)
```

It is also possible to have an evaluated list spliced into the resulting list by using the special marker `,@'. The elements of the spliced list become elements at the same level as the other elements of the resulting list. The equivalent code without using ` is often unreadable. Here are some examples:

```
(setq some-list '(2 3))
     => (2 3)
(cons 1 (append some-list '(4) some-list))
     => (1 2 3 4 2 3)
(` (1 (,@ some-list) 4 (,@ some-list)))
     => (1 2 3 4 2 3)

(setq list '(hack foo bar))
     => (hack foo bar)
(cons 'use
  (cons 'the
    (cons 'words (append (cdr list) '(as elements)))))
     => (use the words foo bar as elements)
(` (use the words (,@ (cdr list)) as elements (,@ nil)))
     => (use the words foo bar as elements)
```

The reason for `(,@ nil)` is to avoid a bug in Emacs version 18. The bug occurs when a call to `,@` is followed only by constant elements. Thus,

```
(` (use the words (,@ (cdr list)) as elements))
```

would not work, though it really ought to. `(,@ nil)` avoids the problem by being a nonconstant element that does not affect the result.

Macro: ` *list*

This macro returns list as `quote` would, except that the list is copied each time this expression is evaluated, and any sublist of the form `(, subexp)` is replaced by the value of subexp. Any sublist of the form `(,@ listexp)` is replaced by evaluating listexp and splicing its elements into the containing

list in place of this sublist. (A single sublist can in this way be replaced by any number of new elements in the containing list.)

There are certain contexts in which `,' would not be recognized and should not be used:

```
;; Use of a `,' expression as the CDR of a list.
(` (a . (, 1)))                               ; Not (a . 1)
     => (a \, 1)

;; Use of `,' in a vector.
(` [a (, 1) c])                               ; Not [a 1 c]
     error--> Wrong type argument

;; Use of a `,' as the entire argument of ``'.
(` (, 2))                                     ; Not 2
     => (\, 2)
```

Common Lisp note: in Common Lisp, `,' and `,@' are implemented as reader macros, so they do not require parentheses. Emacs Lisp implements them as functions because reader macros are not supported (to save space).

Common Problems Using Macros

The basic facts of macro expansion have all been described above, but there consequences are often counterintuitive. This section describes some important consequences that can lead to trouble, and rules to follow to avoid trouble.

Evaluating Macro Arguments Too Many Times

When defining a macro you must pay attention to the number of times the arguments will be evaluated when the expansion is executed. The following macro (used to facilitate iteration) illustrates the problem. This macro allows us to write a simple "for" loop such as one might find in Pascal.

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple \"for\" loop, e.g.,
    (for i from 1 to 10 do (print i))."
  (list 'let (list (list var init))
        (cons 'while (cons (list '<= var final)
                           (append body (list (list 'inc var)))))))
=> for

(for i from 1 to 3 do
   (setq square (* i i))
   (princ (format "\n%d %d" i square)))
==>
```


```
(let ((i 1))
  (while (<= i 3)
    (setq square (* i i))
    (princ (format "%d      %d" i square))
    (inc i)))

     -|1       1
     -|2       4
     -|3       9
=> nil
```

(The arguments `from`, `to`, and `do` in this macro are "syntactic sugar"; they are entirely ignored. The idea is that you will write noise words (such as `from`, `to`, and `do`) in those positions in the macro call.)

This macro suffers from the defect that final is evaluated on every iteration. If final is a constant, this is not a problem. If it is a more complex form, say `(long-complex-calculation x)`, this can slow down the execution significantly. If final has side effects, executing it more than once is probably incorrect.

A well-designed macro definition takes steps to avoid this problem by producing an expansion that evaluates the argument expressions exactly once unless repeated evaluation is part of the intended purpose of the macro. Here is a correct expansion for the `for` macro:

```
(let ((i 1)
      (max 3))
  (while (<= i max)
    (setq square (* i i))
    (princ (format "%d      %d" i square))
    (inc i)))
```

Here is a macro definition that creates this expansion:

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple for loop: (for i from 1 to 10 do (print i))."
  (` (let ((, var) (, init))
           (max (, final)))
       (while (<= (, var) max)
         (,@ body)
         (inc (, var))))))
```

Unfortunately, this introduces another problem.

Local Variables in Macro Expansions

The new definition of `for` has a new problem: it introduces a local variable named `max` which the user does not expect. This causes trouble in examples such as the following:


```
(let ((max 0))
  (for x from 0 to 10 do
    (let ((this (frob x)))
      (if (< max this)
          (setq max this)))))
```

The references to `max` inside the body of the `for`, which are supposed to refer to the user's binding of `max`, really access the binding made by `for`.

The way to correct this is to use an uninterned symbol instead of `max` (see section Creating and Interning Symbols). The uninterned symbol can be bound and referred to just like any other symbol, but since it is created by `for`, we know that it cannot appear in the user's program. Since it is not interned, there is no way the user can put it into the program later. It will never appear anywhere except where put by `for`. Here is a definition of `for` which works this way:

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple for loop: (for i from 1 to 10 do (print i))."
  (let ((tempvar (make-symbol "max")))
    (` (let (((, var) (, init))
             ((, tempvar) (, final)))
         (while (<= (, var) (, tempvar))
                (,@ body)
                (inc (, var)))))))
```

This creates an uninterned symbol named `max` and puts it in the expansion instead of the usual interned symbol `max` that appears in expressions ordinarily.

Evaluating Macro Arguments in Expansion

Another problem can happen if you evaluate any of the macro argument expressions during the computation of the expansion, such as by calling `eval` (see section Eval). If the argument is supposed to refer to the user's variables, you may have trouble if the user happens to use a variable with the same name as one of the macro arguments. Inside the macro body, the macro argument binding is the most local binding of this variable, so any references inside the form being evaluated do refer to it. Here is an example:

```
(defmacro foo (a)
  (list 'setq (eval a) t))
     => foo
(setq x 'b)
(foo x) ==> (setq b t)
     => t                  ; and b has been set.
;; but
(setq a 'b)
(foo a) ==> (setq 'b t)     ; invalid!
```


```
error--> Symbol's value is void: b
```

It makes a difference whether the user types `a` or `x`, because `a` conflicts with the macro argument variable `a`.

In general it is best to avoid calling `eval` in a macro definition at all.

How Many Times is the Macro Expanded?

Occasionally problems result from the fact that a macro call is expanded each time it is evaluated in an interpreted function, but is expanded only once (during compilation) for a compiled function. If the macro definition has side effects, they will work differently depending on how many times the macro is expanded.

In particular, constructing objects is a kind of side effect. If the macro is called once, then the objects are constructed only once. In other words, the same structure of objects is used each time the macro call is executed. In interpreted operation, the macro is reexpanded each time, producing a fresh collection of objects each time. Usually this does not matter--the objects have the same contents whether they are shared or not. But if the surrounding program does side effects on the objects, it makes a difference whether they are shared. Here is an example:

```
(defmacro new-object ()
  (list 'quote (cons nil nil)))

(defun initialize (condition)
  (let ((object (new-object)))
    (if condition
        (setcar object condition))
    object))
```

If `initialize` is interpreted, a new list `(nil)` is constructed each time `initialize` is called. Thus, no side effect survives between calls. If `initialize` is compiled, then the macro `new-object` is expanded during compilation, producing a single "constant" `(nil)` that is reused and altered each time `initialize` is called.

Go to the previous, next section.

Go to the previous, next section.

Loading

Loading a file of Lisp code means bringing its contents into the Lisp environment in the form of Lisp objects. Emacs finds and opens the file, reads the text, evaluates each form, and then closes the file.

The load functions evaluate all the expressions in a file just as the `eval-current-buffer` function evaluates all the expressions in a buffer. The difference is that the load functions read and evaluate the text in the file as found on disk, not the text in an Emacs buffer.

The loaded file must contain Lisp expressions, either as source code or, optionally, as byte-compiled code. Each form in the file is called a top-level form. There is no special format for the forms in a loadable file; any form in a file may equally well be typed directly into a buffer and evaluated there. (Indeed, most code is tested this way.) Most often, the forms are function definitions and variable definitions.

A file containing Lisp code is often called a library. Thus, the "Rmail library" is a file containing code for Rmail mode. Similarly, a "Lisp library directory" is a directory of files containing Lisp code.

How Programs Do Loading

There are several interface functions for loading. For example, the `autoload` function creates a Lisp object that loads a file when it is evaluated (see section Autoload). `require` also causes files to be loaded (see section Features). Ultimately, all these facilities call the `load` function to do the work.

Function: **load** *filename &optional missing-ok nomessage nosuffix*

This function finds and opens a file of Lisp code, evaluates all the forms in it, and closes the file.

To find the file, `load` first looks for a file named `` `filename.elc' ``, that is, for a file whose name has `.elc' appended. If such a file exists, it is loaded. But if there is no file by that name, then `load` looks for a file whose name has `.el' appended. If that file exists, it is loaded. Finally, if there is no file by either name, `load` looks for a file named filename with nothing appended, and loads it if it exists. (The `load` function is not clever about looking at filename. In the perverse case of a file named `` `foo.el.el' ``, evaluation of `(load "foo.el")` will indeed find it.)

If the optional argument nosuffix is non-`nil`, then the suffixes `.elc' and `.el' are not tried. In this case, you must specify the precise file name you want.

If filename is a relative file name, such as `` `foo' `` or `` `baz/foo.bar' ``, `load` searches for the file using the variable `load-path`. It appends filename to each of the directories listed in `load-path`, and loads the first file it finds whose name matches. The current default directory is tried only if it is specified in `load-path`, where it is represented as `nil`. All three possible suffixes are tried in the first directory in `load-path`, then all three in the second directory in `load-path`, etc.

If you get a warning that `foo.elc' is older than `foo.el', it means you should consider recompiling `foo.el'. See section Byte Compilation.

Messages like `Loading foo...' and `Loading foo...done' appear in the echo area during loading unless nomessage is non-nil.

Any errors that are encountered while loading a file cause load to abort. If the load was done for the sake of autoload, certain kinds of top-level forms, those which define functions, are undone.

The error file-error is signaled (with `Cannot open load file filename') if no file is found. No error is signaled if missing-ok is non-nil---then load just returns nil.

load returns t if the file loads successfully.

User Option: **load-path**

The value of this variable is a list of directories to search when loading files with load. Each element is a string (which must be a directory name) or nil (which stands for the current working directory). The value of load-path is initialized from the environment variable EMACSLOADPATH, if it exists; otherwise it is set to the default specified in `emacs/src/paths.h' when Emacs is built.

The syntax of EMACSLOADPATH is the same as that of PATH; fields are separated by `:', and `.' is used for the current default directory. Here is an example of how to set your EMACSLOADPATH variable from a csh `.login' file:

```
setenv EMACSLOADPATH .:/user/bil/emacs:/usr/local/lib/emacs/lisp
```

Here is how to set it using sh:

```
export EMACSLOADPATH
EMACSLOADPATH=.:/user/bil/emacs:/usr/local/lib/emacs/lisp
```

Here is an example of code you can place in a `.emacs' file to add several directories to the front of your default load-path:

```
(setq load-path
      (append
       (list nil
             "/user/bil/emacs"
             "/usr/local/lisplib")
       load-path))
```

In this example, the path searches the current working directory first, followed then by the `/user/bil/emacs' directory and then by the `/usr/local/lisplib' directory, which are then followed by the standard directories for Lisp code.

When Emacs version 18 processes command options `-l' or `-load' which specify Lisp libraries to be loaded, it temporarily adds the current directory to the front of load-path so that files in the current directory can be specified easily. Newer Emacs versions also find such files in the current directory, but

without altering `load-path`.

Variable: **load-in-progress**

This variable is non-`nil` if Emacs is in the process of loading a file, and it is `nil` otherwise. This is how `defun` and `provide` determine whether a load is in progress, so that their effect can be undone if the load fails.

To learn how `load` is used to build Emacs, see section Building Emacs.

Autoload

The autoload facility allows you to make a function or macro available but put off loading its actual definition. An attempt to call a symbol whose definition is an autoload object automatically reads the file to install the real definition and its other associated code, and then calls the real definition.

To prepare a function or macro for autoloading, you must call `autoload`, specifying the function name and the name of the file to be loaded. A file such as `` `emacs/lisp/loaddefs.el' `` usually does this when Emacs is first built.

The following example shows how `doctor` is prepared for autoloading in `` `loaddefs.el' ``:

```
(autoload 'doctor "doctor"
  "\
Switch to *doctor* buffer and start giving psychotherapy."
  t)
```

The backslash and newline immediately following the double-quote are a convention used only in the preloaded Lisp files such as `` `loaddefs.el' ``; they cause the documentation string to be put in the `` `etc/DOC' `` file. (See section Building Emacs.) In any other source file, you would write just this:

```
(autoload 'doctor "doctor"
  "Switch to *doctor* buffer and start giving psychotherapy."
  t)
```

Calling `autoload` creates an autoload object containing the name of the file and some other information, and makes this the function definition of the specified symbol. When you later try to call that symbol as a function or macro, the file is loaded; the loading should redefine that symbol with its proper definition. After the file completes loading, the function or macro is called as if it had been there originally.

If, at the end of loading the file, the desired Lisp function or macro has not been defined, then the error `error` is signaled (with data `"Autoloading failed to define function function-name"`).

The autoloaded file may, of course, contain other definitions and may require or provide one or more features. If the file is not completely loaded (due to an error in the evaluation of the contents) any

function definitions or `provide` calls that occurred during the load are undone. This is to ensure that next attempt to call any function autoloading from this file will try again to load the file. If not for this then some of the functions in the file might appear defined, but they may fail to work properly for the lack of certain subroutines defined later in the file and not loaded successfully.

Emacs as distributed comes with many autoloaded functions. The calls to `autoload` are in the file `` `loaddefs.el' ``. There is a convenient way of updating them automatically.

Write `;;;###autoload' on a line by itself before a function definition before the real definition of the function, in its autoloadable source file; then the command M-x update-file-autoloads automatically puts the `autoload` call into `` `loaddefs.el' ``. M-x update-directory-autoloads is more powerful; it updates autoloads for all files in the current directory.

You can also put other kinds of forms into `` `loaddefs.el' ``, by writing `;;;###autoload' followed on the same line by the form. M-x update-file-autoloads copies the form from that line.

The commands for updating autoloads work by visiting and editing the file `` `loaddefs.el' ``. To make the result take effect, you must save that file's buffer.

Function: **autoload** *symbol filename &optional docstring interactive type*

This function defines the function (or macro) named symbol so as to load automatically from filename. The string filename is a file name which will be passed to `load` when the function is called.

The argument docstring is the documentation string for the function. Normally, this is the same string that is in the function definition itself. This makes it possible to look at the documentation without loading the real definition.

If interactive is non-`nil`, then the function can be called interactively. This lets completion in M-x work without loading the function's real definition. The complete interactive specification need not be given here. If type is `macro`, then the function is really a macro. If type is `keymap`, then the function is really a keymap.

If symbol already has a non-`nil` function definition that is not an autoload object, `autoload` does nothing and returns `nil`. If the function cell of symbol is void, or is already an autoload object, then it is set to an autoload object that looks like this:

```
(autoload filename docstring interactive type)
```

For example,

```
(symbol-function 'run-prolog)
     => (autoload "prolog" 169681 t nil)
```

In this case, `"prolog"` is the name of the file to load, 169681 refers to the documentation string in the `` `emacs/etc/DOC' `` file (see section Documentation Basics), `t` means the function is interactive, and `nil` that it is not a macro.

Repeated Loading

You may load a file more than once in an Emacs session. For example, after you have rewritten and reinstalled a function definition by editing it in a buffer, you may wish to return to the original version; you can do this by reloading the file in which it is located.

When you load or reload files, bear in mind that the `load` and `load-library` functions automatically load a byte-compiled file rather than a non-compiled file of similar name. If you rewrite a file that you intend to save and reinstall, remember to byte-compile it if necessary; otherwise you may find yourself inadvertently reloading the older, byte-compiled file instead of your newer, non-compiled file!

When writing the forms in a library, keep in mind that the library might be loaded more than once. For example, the choice of `defvar` vs. `defconst` for defining a variable depends on whether it is desirable to reinitialize the variable if the library is reloaded: `defconst` does so, and `defvar` does not. (See section Defining Global Variables.)

The simplest way to add an element to an alist is like this:

```
(setq minor-mode-alist
      (cons '(leif-mode " Leif") minor-mode-alist))
```

But this would add multiple elements if the library is reloaded. To avoid the problem, write this:

```
(or (assq 'leif-mode minor-mode-alist)
    (setq minor-mode-alist
          (cons '(leif-mode " Leif") minor-mode-alist)))
```

Occasionally you will want to test explicitly whether a library has already been loaded; you can do so as follows:

```
(if (not (boundp 'foo-was-loaded))
    execute-first-time-only)

(setq foo-was-loaded t)
```

Features

`provide` and `require` are an alternative to `autoload` for loading files automatically. They work in terms of named features. Autoloading is triggered by calling a specific function, but a feature is loaded the first time another program asks for it by name.

The use of named features simplifies the task of determining whether required definitions have been defined. A feature name is a symbol that stands for a collection of functions, variables, etc. A program that needs the collection may ensure that they are defined by requiring the feature. If the file that contains the feature has not yet been loaded, then it will be loaded (or an error will be signaled if it cannot be

loaded). The file thus loaded must provide the required feature or an error will be signaled.

To require the presence of a feature, call `require` with the feature name as argument. `require` looks in the global variable `features` to see whether the desired feature has been provided already. If not, it loads the feature from the appropriate file. This file should call `provide` at the top-level to add the feature to `features`.

Features are normally named after the files they are provided in so that `require` need not be given the file name.

For example, in `` `emacs/lisp/prolog.el' ``, the definition for `run-prolog` includes the following code:

```
(defun run-prolog ()
  "Run an inferior Prolog process,\
 input and output via buffer *prolog*."
  (interactive)
  (require 'comint)
  (switch-to-buffer (make-comint "prolog" prolog-program-name))
  (inferior-prolog-mode))
```

The expression `(require 'shell)` loads the file `` `shell.el' `` if it has not yet been loaded. This ensures that `make-shell` is defined.

The `` `shell.el' `` file contains the following top-level expression:

```
(provide 'shell)
```

This adds `shell` to the global `features` list when the `` `shell' `` file is loaded, so that `(require 'shell)` will henceforth know that nothing needs to be done.

When `require` is used at top-level in a file, it takes effect if you byte-compile that file (see section Byte Compilation). This is in case the required package contains macros that the byte compiler must know about.

Although top-level calls to `require` are evaluated during byte compilation, `provide` calls are not. Therefore, you can ensure that a file of definitions is loaded before it is byte-compiled by including a `provide` followed by a `require` for the same feature, as in the following example.

```
(provide 'my-feature)  ; Ignored by byte compiler,
                       ;   evaluated by load.
(require 'my-feature)  ; Evaluated by byte compiler.
```

Function: **provide** *feature*

This function announces that feature is now loaded, or being loaded, into the current Emacs session. This means that the facilities associated with feature are or will be available for other Lisp programs.

The direct effect of calling `provide` is to add feature to the front of the list `features` if it is not

already in the list. The argument feature must be a symbol. `provide` returns feature.

```
features
     => (bar bish)

(provide 'foo)
     => foo
features
     => (foo bar bish)
```

During autoloading, if the file is not completely loaded (due to an error in the evaluation of the contents) any function definitions or `provide` calls that occurred during the load are undone. See section Autoload.

Function: **require** *feature &optional filename*

This function checks whether feature is present in the current Emacs session (using `(featurep feature)`; see below). If it is not, then `require` loads filename with `load`. If filename is not supplied, then the name of the symbol feature is used as the file name to load.

If feature is not provided after the file has been loaded, Emacs will signal the error `error` (with data `Required feature feature was not provided').

Function: **featurep** *feature*

This function returns `t` if feature has been provided in the current Emacs session (i.e., feature is a member of `features`.)

Variable: **features**

The value of this variable is a list of symbols that are the features loaded in the current Emacs session. Each symbol was put in this list with a call to `provide`. The order of the elements in the `features` list is not significant.

Unloading

You can discard the functions and variables loaded by a library to reclaim memory for other Lisp objects. To do this, use the function `unload-feature`:

Command: **unload-feature** *feature*

This command unloads the library that provided feature feature. It undefines all functions and variables defined with `defvar`, `defmacro`, `defconst`, `defsubst` and `defalias` by the library which provided feature feature. It then restores any autoloads associated with those symbols.

The `unload-feature` function is written in Lisp; its actions are based on the variable `load-history`.

Variable: **load-history** *feature association list*

This variable's value is an alist connecting library names with the names of functions and variables th define, the features they provide, and the features they require.

Each element is a list and describes one library. The CAR of the list is the name of the library, as a string. The rest of the list is composed of these kinds of objects:

- Symbols, which were defined as functions or variables.
- Lists of the form `(require . feature)` indicating the features that are required.
- Lists of the form `(provide . feature)` indicating the features that are provided.

The value of `load-history` may have one element whose CAR is `nil`. This element describes definitions made with `eval-buffer` on a buffer that is not visiting a file.

The command `eval-region` updates `load-history`, but does so by adding the symbols defined to the element for the file being visited, rather than replacing that element.

Hooks for Loading

You can ask for code to be executed if and when a particular library is loaded, by calling `eval-after-load`.

Function: **eval-after-load** *library form*

This function arranges to evaluate form at the end of loading the library library, if and when library is loaded.

The library name library must exactly match the argument of `load`. To get the proper results when an installed library is found by searching `load-path`, you should not include any directory names in library.

An error in form does not undo the load, but does prevent execution of the rest of form.

Variable: **after-load-alist**

An alist of expressions to evaluate if and when particular libraries are loaded. Each element looks like this:

```
(filename forms...)
```

The function `load` checks `after-load-alist` in order to implement `eval-after-load`.

Go to the previous, next section.

Go to the previous, next section.

Byte Compilation

GNU Emacs Lisp has a compiler that translates functions written in Lisp into a special representation called byte-code that can be executed more efficiently. The compiler replaces Lisp function definitions with byte-code. When a byte-code function is called, its definition is evaluated by the byte-code interpreter.

Because the byte-compiled code is evaluated by the byte-code interpreter, instead of being executed directly by the machine's hardware (as true compiled code is), byte-code is completely transportable from machine to machine without recompilation. It is not, however, as fast as true compiled code.

In general, any version of Emacs can run byte-compiled code produced by recent earlier versions of Emacs, but the reverse is not true. In particular, if you compile a program with Emacs 18, you can run the compiled code in Emacs 19, but not vice versa.

See section Debugging Problems in Compilation, for how to investigate errors occurring in byte compilation.

The Compilation Functions

You can byte-compile an individual function or macro definition with the `byte-compile` function. You can compile a whole file with `byte-compile-file`, or several files with `byte-recompile-directory` or `batch-byte-compile`.

When you run the byte compiler, you may get warnings in a buffer called `*Compile-Log*'. These report usage in your program that suggest a problem, but are not necessarily erroneous.

Be careful when byte-compiling code that uses macros. Macro calls are expanded when they are compiled, so the macros must already be defined for proper compilation. For more details, see section Macros and Byte Compilation.

While byte-compiling a file, any `require` calls at top-level are executed. One way to ensure that necessary macro definitions are available during compilation is to require the file that defines them. See section Features.

A byte-compiled function is not as efficient as a primitive function written in C, but runs much faster than the version written in Lisp. For a rough comparison, consider the example below:

```
(defun silly-loop (n)
  "Return time before and after N iterations of a loop."
  (let ((t1 (current-time-string)))
    (while (> (setq n (1- n))
              0))
```


```
    (list t1 (current-time-string))))
=> silly-loop

(silly-loop 100000)
=> ("Thu Jan 12 20:18:38 1989"
    "Thu Jan 12 20:19:29 1989")  ; 51 seconds

(byte-compile 'silly-loop)
=> [Compiled code not shown]

(silly-loop 100000)
=> ("Thu Jan 12 20:21:04 1989"
    "Thu Jan 12 20:21:17 1989")  ; 13 seconds
```

In this example, the interpreted code required 51 seconds to run, whereas the byte-compiled code required 13 seconds. These results are representative, but actual results will vary greatly.

Function: **byte-compile** *symbol*

This function byte-compiles the function definition of symbol, replacing the previous definition with the compiled one. The function definition of symbol must be the actual code for the function; i.e., the compiler does not follow indirection to another symbol. `byte-compile` does not compile macros. `byte-compile` returns the new, compiled definition of symbol.

```
(defun factorial (integer)
  "Compute factorial of INTEGER."
  (if (= 1 integer) 1
    (* integer (factorial (1- integer)))))
     => factorial

(byte-compile 'factorial)
     =>
#[(integer)
  "^H\301U\203^H^@\301\207\302^H\303^HS!\"\207"
  [integer 1 * factorial]
  4 "Compute factorial of INTEGER."]
```

The result is a compiled function object. The string it contains is the actual byte-code; each character in it is an instruction. The vector contains all the constants, variable names and function names used by the function, except for certain primitives that are coded as special instructions.

Command: **compile-defun**

This command reads the defun containing point, compiles it, and evaluates the result. If you use this on a defun that is actually a function definition, the effect is to install a compiled version of that function.

Command: **byte-compile-file** *filename*

This function compiles a file of Lisp code named filename into a file of byte-code. The output file's n is made by appending `c' to the end of filename.

Compilation works by reading the input file one form at a time. If it is a definition of a function or macro, the compiled function or macro definition is written out. Other forms are batched together, then each batch is compiled, and written so that its compiled code will be executed when the file is read. All comments are discarded when the input file is read.

This command returns `t`. When called interactively, it prompts for the file name.

```
% ls -l push*
-rw-r--r--  1 lewis     791 Oct  5 20:31 push.el

(byte-compile-file "~/emacs/push.el")
     => t

% ls -l push*
-rw-r--r--  1 lewis     791 Oct  5 20:31 push.el
-rw-rw-rw-  1 lewis     638 Oct  8 20:25 push.elc
```

Command: **byte-recompile-directory** *directory flag*

This function recompiles every `.el' file in directory that needs recompilation. A file needs recompilation if a `.elc' file exists but is older than the `.el' file.

If a `.el' file exists, but there is no corresponding `.elc' file, then flag is examined. If it is `nil`, the file is ignored. If it is non-`nil`, the user is asked whether the file should be compiled.

The returned value of this command is unpredictable.

Function: **batch-byte-compile**

This function runs `byte-compile-file` on the files remaining on the command line. This function must be used only in a batch execution of Emacs, as it kills Emacs on completion. An error in one file does not prevent processing of subsequent files. (The file which gets the error will not, of course, produce any compiled code.)

```
% emacs -batch -f batch-byte-compile *.el
```

Function: **byte-code** *code-string data-vector max-stack*

This function actually interprets byte-code. A byte-compiled function is actually defined with a body that calls `byte-code`. Don't call this function yourself. Only the byte compiler knows how to generate valid calls to this function.

In newer Emacs versions (19 and up), byte-code is usually executed as part of a compiled function object, and only rarely as part of a call to `byte-code`.

Evaluation During Compilation

These features permit you to write code to be evaluated during compilation of a program.

Special Form: **eval-and-compile** *body*

This form marks body to be evaluated both when you compile the containing code and when you run it (whether compiled or not).

You can get a similar result by putting body in a separate file and referring to that file with `require`. Using `require` is preferable if there is a substantial amount of code to be executed in this way.

Special Form: **eval-when-compile** *body*

This form marks body to be evaluated at compile time *only*. The result of evaluation by the compiler becomes a constant which appears in the compiled program. When the program is interpreted, not compiled at all, body is evaluated normally.

At top-level, this is analogous to the Common Lisp idiom `(eval-when (compile) ...)`. Elsewhere, the Common Lisp `#.' reader macro (but not when interpreting) is closer to what `eval-when-compile` does.

Byte-Code Objects

Byte-compiled functions have a special data type: they are byte-code function objects.

Internally, a byte-code function object is much like a vector; however, the evaluator handles this data type specially when it appears as a function to be called. The printed representation for a byte-code function object is like that for a vector, with an additional `#' before the opening `['.

In Emacs version 18, there was no byte-code function object data type; compiled functions used the function `byte-code` to run the byte code.

A byte-code function object must have at least four elements; there is no maximum number, but only the first six elements are actually used. They are:

arglist
: The list of argument symbols.

byte-code
: The string containing the byte-code instructions.

constants
: The vector of constants referenced by the byte code.

stacksize
: The maximum stack size this function needs.

docstring
: The documentation string (if any); otherwise, `nil`. For functions preloaded before Emacs is

dumped, this is usually an integer which is an index into the `DOC' file; use `documentati` to convert this into a string (see section Access to Documentation Strings).

interactive

The interactive spec (if any). This can be a string or a Lisp expression. It is `nil` for a function that isn't interactive.

Here's an example of a byte-code function object, in printed representation. It is the definition of the command `backward-sexp`.

```
#[(&optional arg)
  "^H\204^F^@\301^P\302^H[!\207"
  [arg 1 forward-sexp]
  2
  254435
  "p"]
```

The primitive way to create a byte-code object is with `make-byte-code`:

Function: **make-byte-code** *&rest elements*

This function constructs and returns a byte-code function object with elements as its elements.

You should not try to come up with the elements for a byte-code function yourself, because if they are inconsistent, Emacs may crash when you call the function. Always leave it to the byte-compiler to create these objects; it, we hope, always makes the elements consistent.

You can access the elements of a byte-code object using `aref`; you can also use `vconcat` to create a vector with the same elements.

Disassembled Byte-Code

People do not write byte-code; that job is left to the byte compiler. But we provide a disassembler to satisfy a cat-like curiosity. The disassembler converts the byte-compiled code into humanly readable form.

The byte-code interpreter is implemented as a simple stack machine. Values get stored by being pushed onto the stack, and are popped off and manipulated, the results being pushed back onto the stack. When a function returns, the top of the stack is popped and returned as the value of the function.

In addition to the stack, values used during byte-code execution can be stored in ordinary Lisp variables. Variable values can be pushed onto the stack, and variables can be set by popping the stack.

Command: **disassemble** *object &optional stream*

This function prints the disassembled code for object. If stream is supplied, then output goes there. Otherwise, the disassembled code is printed to the stream `standard-output`. The argument object can be a function name or a lambda expression.

As a special exception, if this function is used interactively, it outputs to a buffer named `*Disassembl

Here are two examples of using the `disassemble` function. We have added explanatory comments to help you relate the byte-code to the Lisp source; these do not appear in the output of `disassemble`. These examples show unoptimized byte-code. Nowadays byte-code is usually optimized, but we did not want to rewrite these examples, since they still serve their purpose.

```
(defun factorial (integer)
  "Compute factorial of an integer."
  (if (= 1 integer) 1
    (* integer (factorial (1- integer)))))
     => factorial

(factorial 4)
     => 24

(disassemble 'factorial)
     -| byte-code for factorial:
 doc: Compute factorial of an integer.
 args: (integer)

0   constant 1              ; Push 1 onto stack.

1   varref   integer        ; Get value of integer
                            ;   from the environment
                            ;   and push the value
                            ;   onto the stack.

2   eqlsign                 ; Pop top two values off stack,
                            ;   compare them,
                            ;   and push result onto stack.

3   goto-if-nil 10          ; Pop and test top of stack;
                            ;   if nil, go to 10,
                            ;   else continue.

6   constant 1              ; Push 1 onto top of stack.

7   goto     17             ; Go to 17 (in this case, 1 will be
                            ;   returned by the function).

10  constant *              ; Push symbol * onto stack.

11  varref   integer        ; Push value of integer onto stack.

12  constant factorial      ; Push factorial onto stack.
```


```
13  varref   integer        ; Push value of integer onto stack.

14  sub1                    ; Pop integer, decrement value,
                            ;   push new value onto stack.

                            ; Stack now contains:
                            ;   - decremented value of integer
                            ;   - factorial
                            ;   - value of integer
                            ;   - *

15  call     1              ; Call function factorial using
                            ;   the first (i.e., the top) element
                            ;   of the stack as the argument;
                            ;   push returned value onto stack.

                            ; Stack now contains:
                            ;   - result of result of recursive
                            ;       call to factorial
                            ;   - value of integer
                            ;   - *

16  call     2              ; Using the first two
                            ;   (i.e., the top two)
                            ;   elements of the stack
                            ;   as arguments,
                            ;   call the function *,
                            ;   pushing the result onto the stack.

17  return                  ; Return the top element
                            ;   of the stack.
     => nil
```

The `silly-loop` function is somewhat more complex:

```
(defun silly-loop (n)
  "Return time before and after N iterations of a loop."
  (let ((t1 (current-time-string)))
    (while (> (setq n (1- n))
              0))
    (list t1 (current-time-string))))
     => silly-loop

(disassemble 'silly-loop)
     -| byte-code for silly-loop:
```


```
 doc: Return time before and after N iterations of a loop.
 args: (n)

0   constant current-time-string  ; Push
                                  ;   current-time-string
                                  ;   onto top of stack.

1   call     0                    ; Call current-time-string
                                  ;    with no argument,
                                  ;    pushing result onto stack.

2   varbind  t1                   ; Pop stack and bind t1
                                  ;   to popped value.

3   varref   n                    ; Get value of n from
                                  ;   the environment and push
                                  ;   the value onto the stack.

4   sub1                          ; Subtract 1 from top of stack.

5   dup                           ; Duplicate the top of the stack;
                                  ;   i.e. copy the top of
                                  ;   the stack and push the
                                  ;   copy onto the stack.

6   varset   n                    ; Pop the top of the stack,
                                  ;   and bind n to the value.

                                  ; In effect, the sequence dup varset
                                  ;   copies the top of the stack
                                  ;   into the value of n
                                  ;   without popping it.

7   constant 0                    ; Push 0 onto stack.

8   gtr                           ; Pop top two values off stack,
                                  ;   test if n is greater than 0
                                  ;   and push result onto stack.

9   goto-if-nil-else-pop 17       ; Goto 17 if n > 0
                                  ;   else pop top of stack
                                  ;   and continue
                                  ;   (this exits the while loop).

12  constant nil                  ; Push nil onto stack
```


```
                                ;   (this is the body of the loop).

13  discard                     ; Discard result of the body
                                ;   of the loop (a while loop
                                ;   is always evaluated for
                                ;   its side effects).

14  goto    3                   ; Jump back to beginning
                                ;   of while loop.

17  discard                     ; Discard result of while loop
                                ;   by popping top of stack.

18  varref  t1                  ; Push value of t1 onto stack.

19  constant current-time-string  ; Push
                                  ;   current-time-string
                                  ;   onto top of stack.

20  call    0                   ; Call current-time-string again.

21  list2                       ; Pop top two elements off stack,
                                ;   create a list of them,
                                ;   and push list onto stack.

22  unbind  1                   ; Unbind t1 in local environment.

23  return                      ; Return value of the top of stack.

     => nil
```

Go to the previous, next section.

Go to the previous, next section.

Debugging Lisp Programs

There are three ways to investigate a problem in an Emacs Lisp program, depending on what you are doing with the program when the problem appears.

- If the problem occurs when you run the program, you can use the Lisp debugger to investigate what is happening during execution.
- If the problem is syntactic, so that Lisp cannot even read the program, you can use the Emacs facilities for editing Lisp to localize it.
- If the problem occurs when trying to compile the program with the byte compiler, you need to know how to examine the compiler's input buffer.

Another useful debugging tool is a dribble file. When a dribble file is open, Emacs copies all keyboard input characters to that file. Afterward, you can examine the file to find out what input was used. See section Terminal Input.

For debugging problems in terminal descriptions, the `open-termscript` function can be useful. See section Terminal Output.

The Lisp Debugger

The Lisp debugger provides you with the ability to suspend evaluation of a form. While evaluation is suspended (a state that is commonly known as a break), you may examine the run time stack, examine the values of local or global variables, or change those values. Since a break is a recursive edit, all the usual editing facilities of Emacs are available; you can even run programs that will enter the debugger recursively. See section Recursive Editing.

Entering the Debugger on an Error

The most important time to enter the debugger is when a Lisp error happens. This allows you to investigate the immediate causes of the error.

However, entry to the debugger is not a normal consequence of an error. Many commands frequently get Lisp errors when invoked in inappropriate contexts (such as C-f at the end of the buffer) and during ordinary editing it would be very unpleasant to enter the debugger each time this happens. If you want errors to enter the debugger, set the variable `debug-on-error` to non-`nil`.

User Option: **debug-on-error**

This variable determines whether the debugger is called when a error is signaled and not handled. If `debug-on-error` is `t`, all errors call the debugger. If it is `nil`, none call the debugger.

The value can also be a list of error conditions that should call the debugger. For example, if you set it to

the list `(void-variable)`, then only errors about a variable that has no value invoke the debugge

Debugging Infinite Loops

When a program loops infinitely and fails to return, your first problem is to stop the loop. On most operating systems, you can do this with C-g, which causes quit.

Ordinary quitting gives no information about why the program was looping. To get more information, you can set the variable `debug-on-quit` to non-`nil`. Quitting with C-g is not considered an error, and `debug-on-error` has no effect on the handling of C-g. Contrariwise, `debug-on-quit` has no effect on errors.

Once you have the debugger running in the middle of the infinite loop, you can proceed from the debugger using the stepping commands. If you step through the entire loop, you will probably get enough information to solve the problem.

User Option: **debug-on-quit**

This variable determines whether the debugger is called when `quit` is signaled and not handled. If `debug-on-quit` is non-`nil`, then the debugger is called whenever you quit (that is, type C-g). If `debug-on-quit` is `nil`, then the debugger is not called when you quit. See section Quitting.

Entering the Debugger on a Function Call

To investigate a problem that happens in the middle of a program, one useful technique is to cause the debugger to be entered when a certain function is called. You can do this to the function in which the problem occurs, and then step through the function, or you can do this to a function called shortly before the problem, step quickly over the call to that function, and then step through its caller.

Command: **debug-on-entry** *function-name*

This function requests function-name to invoke the debugger each time it is called. It works by inserting the form `(debug 'debug)` into the function definition as the first form.

Any function defined as Lisp code may be set to break on entry, regardless of whether it is interpreted code or compiled code. Even functions that are commands may be debugged--they will enter the debugger when called inside a function, or when called interactively (after the reading of the arguments). Primitive functions (i.e., those written in C) may not be debugged.

When `debug-on-entry` is called interactively, it prompts for function-name in the minibuffer.

Caveat: if `debug-on-entry` is called more than once on the same function, the second call does nothing. If you redefine a function after using `debug-on-entry` on it, the code to enter the debugger is lost.

`debug-on-entry` returns function-name.

```
(defun fact (n)
  (if (zerop n) 1
```


```
      (* n (fact (1- n)))))
     => fact
(debug-on-entry 'fact)
     => fact
(fact 3)
     => 6

------ Buffer: *Backtrace* ------
Entering:
* fact(3)
  eval-region(4870 4878 t)
  byte-code("...")
  eval-last-sexp(nil)
  (let ...)
  eval-insert-last-sexp(nil)
* call-interactively(eval-insert-last-sexp)
------ Buffer: *Backtrace* ------

(symbol-function 'fact)
     => (lambda (n)
          (debug (quote debug))
          (if (zerop n) 1 (* n (fact (1- n)))))
```

Command: **cancel-debug-on-entry** *function-name*

This function undoes the effect of `debug-on-entry` on function-name. When called interactively, it prompts for function-name in the minibuffer.

If `cancel-debug-on-entry` is called more than once on the same function, the second call does nothing. `cancel-debug-on-entry` returns function-name.

Explicit Entry to the Debugger

You can cause the debugger to be called at a certain point in your program by writing the expression `(debug)` at that point. To do this, visit the source file, insert the text `(debug)' at the proper place, and type C-M-x. Be sure to undo this insertion before you save the file!

The place where you insert `(debug)' must be a place where an additional form can be evaluated and its value ignored. (If the value isn't ignored, it will alter the execution of the program!) Usually this means inside a `progn` or an implicit `progn` (see section Sequencing).

Using the Debugger

When the debugger is entered, it displays the previously selected buffer in one window and a buffer named `*Backtrace*' in another window. The backtrace buffer contains one line for each level of Lisp function execution currently going on. At the beginning of this buffer is a message describing the reason that the debugger was invoked (such as the error message and associated data, if it was invoked due to an

error).

The backtrace buffer is read-only and uses a special major mode, Debugger mode, in which letters are defined as debugger commands. The usual Emacs editing commands are available; thus, you can switch windows to examine the buffer that was being edited at the time of the error, switch buffers, visit files, or do any other sort of editing. However, the debugger is a recursive editing level (see section Recursive Editing) and it is wise to go back to the backtrace buffer and exit the debugger (with the q command) when you are finished with it. Exiting the debugger gets out of the recursive edit and kills the backtrace buffer.

The contents of the backtrace buffer show you the functions that are executing and the arguments that were given to them. It also allows you to specify a stack frame by moving point to the line describing that frame. (A stack frame is the place where the Lisp interpreter records information about a particular invocation of a function. The frame whose line point is on is considered the current frame.) Some of the debugger commands operate on the current frame.

The debugger itself should always be run byte-compiled, since it makes assumptions about how many stack frames are used for the debugger itself. These assumptions are false if the debugger is running interpreted.

Debugger Commands

Inside the debugger (in Debugger mode), these special commands are available in addition to the usual cursor motion commands. (Keep in mind that all the usual facilities of Emacs, such as switching windows or buffers, are still available.)

The most important use of debugger commands is for stepping through code, so that you can see how control flows. The debugger can step through the control structures of an interpreted function, but cannot do so in a byte-compiled function. If you would like to step through a byte-compiled function, replace it with an interpreted definition of the same function. (To do this, visit the source file for the function and type C-M-x on its definition.)

c

Exit the debugger and continue execution. When continuing is possible, it resumes execution of the program as if the debugger had never been entered (aside from the effect of any variables or data structures you may have changed while inside the debugger).

Continuing is possible after entry to the debugger due to function entry or exit, explicit invocation, quitting or certain errors. Most errors cannot be continued; trying to continue an unsuitable error causes the same error to occur again.

d

Continue execution, but enter the debugger the next time any Lisp function is called. This allows you to step through the subexpressions of an expression, seeing what values the subexpressions compute, and what else they do.

The stack frame made for the function call which enters the debugger in this way will be flagged automatically so that the debugger will be called again when the frame is exited. You can use the u

command to cancel this flag.

b

Flag the current frame so that the debugger will be entered when the frame is exited. Frames flagged in this way are marked with stars in the backtrace buffer.

u

Don't enter the debugger when the current frame is exited. This cancels a b command on that frame.

e

Read a Lisp expression in the minibuffer, evaluate it, and print the value in the echo area. This is the same as the command M-ESC, except that e is not normally disabled like M-ESC.

q

Terminate the program being debugged; return to top-level Emacs command execution.

If the debugger was entered due to a C-g but you really want to quit, and not debug, use the q command.

r

Return a value from the debugger. The value is computed by reading an expression with the minibuffer and evaluating it.

The r command makes a difference when the debugger was invoked due to exit from a Lisp call frame (as requested with b); then the value specified in the r command is used as the value of that frame.

You can't use r when the debugger was entered due to an error.

Invoking the Debugger

Here we describe fully the function used to invoke the debugger.

Function: **debug** *&rest debugger-args*

This function enters the debugger. It switches buffers to a buffer named `*Backtrace*' (or `*Backtrace*<2>' if it is the second recursive entry to the debugger, etc.), and fills it with information about the stack of Lisp function calls. It then enters a recursive edit, leaving that buffer in Debugger mode and displayed in the selected window.

Debugger mode provides a c command which operates by exiting the recursive edit, switching back to the previous buffer, and returning to whatever called `debug`. The r command also returns from `debug`. These are the only ways the function `debug` can return to its caller.

If the first of the debugger-args passed to `debug` is `nil` (or if it is not one of the following special values), then the rest of the arguments to `debug` are printed at the top of the `*Backtrace*' buffer. This mechanism is used to display a message to the user.

However, if the first argument passed to `debug` is one of the following special values, then it has special significance. Normally, these values are passed to `debug` only by the internals of Emacs and the

debugger, and not by programmers calling `debug`.

The special values are:

`lambda`

When the first argument is `lambda`, the debugger displays `Entering:' as a line of text at the top of the buffer. This means that a function is being entered when `debug-on-next-call` is non-`nil`.

`debug`

When the first argument is `debug`, the debugger displays `Entering:' just as in the `lambda` case. However, `debug` as the argument indicates that the reason for entering the debugger is that a function set to debug on entry is being entered.

In addition, `debug` as the first argument directs the debugger to mark the function that called `debug` so that it will invoke the debugger when exited. (When `lambda` is the first argument, the debugger does not do this, because it has already been done by the interpreter.)

`t`

When the first argument is `t`, the debugger displays the following as the top line in the buffer:

```
Beginning evaluation of function call form:
```

This indicates that it was entered due to the evaluation of a list form at a time when `debug-on-next-call` is non-`nil`.

`exit`

When the first argument is `exit`, it indicates the exit of a stack frame previously marked to invoke the debugger on exit. The second argument given to `debug` in this case is the value being returned from the frame. The debugger displays `Return value:' on the top line of the buffer, followed by the value being returned.

`error`

When the first argument is `error`, the debugger indicates that it is being entered because an error or `quit` was signaled and not handled, by displaying `Signaling:' followed by the error signaled and any arguments to `signal`. For example,

```
(let ((debug-on-error t))
     (/ 1 0))

------ Buffer: *Backtrace* ------
Signaling: (arith-error)
  /(1 0)
...
------ Buffer: *Backtrace* ------
```

If an error was signaled, presumably the variable `debug-on-error` is non-`nil`. If `quit` was signaled, then presumably the variable `debug-on-quit` is non-`nil`.

`nil`

Use `nil` as the first of the debugger-args when you want to enter the debugger explicitly. The rest of the debugger-args are printed on the top line of the buffer. You can use this feature to display messages--for example, to remind yourself of the conditions under which `debug` is called.

Internals of the Debugger

This section describes functions and variables used internally by the debugger.

Variable: **debugger**

The value of this variable is the function to call to invoke the debugger. Its value must be a function of any number of arguments (or, more typically, the name of a function). Presumably this function will enter some kind of debugger. The default value of the variable is `debug`.

The first argument that Lisp hands to the function indicates why it was called. The convention for arguments is detailed in the description of `debug`.

Command: **backtrace**

This function prints a trace of Lisp function calls currently active. This is the function used by `debug` to fill up the `*Backtrace*' buffer. It is written in C, since it must have access to the stack to determine which function calls are active. The return value is always `nil`.

In the following example, `backtrace` is called explicitly in a Lisp expression. When the expression is evaluated, the backtrace is printed to the stream `standard-output`: in this case, to the buffer `backtrace-output'. Each line of the backtrace represents one function call. If the arguments of the function call are all known, they are displayed; if they are being computed, that fact is stated. The arguments of special forms are elided.

```
(with-output-to-temp-buffer "backtrace-output"
  (let ((var 1))
    (save-excursion
      (setq var (eval '(progn
                         (1+ var)
                         (list 'testing (backtrace)))))))

     => nil

---------- Buffer: backtrace-output ------------
  backtrace()
  (list ...computing arguments...)
  (progn ...)
  eval((progn (1+ var) (list (quote testing) (backtrace))))
  (setq ...)
  (save-excursion ...)
  (let ...)
  (with-output-to-temp-buffer ...)
```


```
  eval-region(1973 2142 #<buffer *scratch*>)
  byte-code("...  for eval-print-last-sexp ...")
  eval-print-last-sexp(nil)
* call-interactively(eval-print-last-sexp)
---------- Buffer: backtrace-output ------------
```

The character `*' indicates a frame whose debug-on-exit flag is set.

Variable: **debug-on-next-call**

This variable determines whether the debugger is called before the next `eval`, `apply` or `funcall`. It is automatically reset to `nil` when the debugger is entered.

The d command in the debugger works by setting this variable.

Function: **backtrace-debug** *level flag*

This function sets the debug-on-exit flag of the eval frame level levels down to flag. If flag is non-`nil`, this will cause the debugger to be entered when that frame exits. Even a nonlocal exit through that frame will enter the debugger.

The debug-on-exit flag is an entry in the stack frame of a function call. This flag is examined on every exit from a function.

Normally, this function is only called by the debugger.

Variable: **command-debug-status**

This variable records the debugging status of current interactive command. Each time a command is called interactively, this variable is bound to `nil`. The debugger can set this variable to leave information for future debugger invocations during the same command.

The advantage of using this variable rather that defining another global variable is that the data will never carry over to a later other command invocation.

Function: **backtrace-frame** *frame-number*

The function `backtrace-frame` is intended for use in Lisp debuggers. It returns information about what computation is happening in the eval frame level levels down.

If that frame has not evaluated the arguments yet (or is a special form), the value is `(nil function arg-forms...)`.

If that frame has evaluated its arguments and called its function already, the value is `(t function arg-values...)`.

In the return value, function is whatever was supplied as CAR of evaluated list, or a `lambda` expression in the case of a macro call. If the function has a `&rest` argument, that is represented as the tail of the list arg-values.

If the argument is out of range, `backtrace-frame` returns `nil`.

Debugging Invalid Lisp Syntax

The Lisp reader reports invalid syntax, but cannot say where the real problem is. For example, the error "End of file during parsing" in evaluating an expression indicates an excess of open parentheses (or square brackets). The reader detects this imbalance at the end of the file, but it cannot figure out where the close parenthesis should have been. Likewise, "Invalid read syntax: ")"" indicates an excess close parenthesis or missing open parenthesis, but not where the missing parenthesis belongs. How, then, to find what to change?

If the problem is not simply an imbalance of parentheses, a useful technique is to try C-M-e at the beginning of each defun, and see if it goes to the place where that defun appears to end. If it does not, there is a problem in that defun.

However, unmatched parentheses are the most common syntax errors in Lisp, and we can give further advice for those cases.

Excess Open Parentheses

The first step is to find the defun that is unbalanced. If there is an excess open parenthesis, the way to do this is to insert a close parenthesis at the end of the file and type C-M-b (`backward-sexp`). This will move you to the beginning of the defun that is unbalanced. (Then type C-SPC C-_ C-u C-SPC to set the mark there, undo the insertion of the close parenthesis, and finally return to the mark.)

The next step is to determine precisely what is wrong. There is no way to be sure of this except to study the program, but often the existing indentation is a clue to where the parentheses should have been. The easiest way to use this clue is to reindent with C-M-q and see what moves.

Before you do this, make sure the defun has enough close parentheses. Otherwise, C-M-q will get an error, or will reindent all the rest of the file until the end. So move to the end of the defun and insert a close parenthesis there. Don't use C-M-e to move there, since that too will fail to work until the defun is balanced.

Then go to the beginning of the defun and type C-M-q. Usually all the lines from a certain point to the end of the function will shift to the right. There is probably a missing close parenthesis, or a superfluous open parenthesis, near that point. (However, don't assume this is true; study the code to make sure.) Once you have found the discrepancy, undo the C-M-q, since the old indentation is probably appropriate to the intended parentheses.

After you think you have fixed the problem, use C-M-q again. It should not change anything, if the problem is really fixed.

Excess Close Parentheses

To deal with an excess close parenthesis, first insert an open parenthesis at the beginning of the file and type C-M-f to find the end of the unbalanced defun. (Then type C-SPC C-_ C-u C-SPC to set the mark there, undo the insertion of the open parenthesis, and finally return to the mark.)

Then find the actual matching close parenthesis by typing C-M-f at the beginning of the defun. This will leave you somewhere short of the place where the defun ought to end. It is possible that you will find a spurious close parenthesis in that vicinity.

If you don't see a problem at that point, the next thing to do is to type C-M-q at the beginning of the defun. A range of lines will probably shift left; if so, the missing open parenthesis or spurious close parenthesis is probably near the first of those lines. (However, don't assume this is true; study the code to make sure.) Once you have found the discrepancy, undo the C-M-q, since the old indentation is probably appropriate to the intended parentheses.

Debugging Problems in Compilation

When an error happens during byte compilation, it is normally due to invalid syntax in the program you are compiling. The compiler prints a suitable error message in the `*Compile-Log*' buffer, and then stops. The message may state a function name in which the error was found, or it may not. Regardless, here is how to find out where in the file the error occurred.

What you should do is switch to the buffer ` *Compiler Input*'. (Note that the buffer name starts with a space, so it does not show up in M-x list-buffers.) This buffer contains the program being compiled, and point shows how far the byte compiler was able to read.

If the error was due to invalid Lisp syntax, point shows exactly where the invalid syntax was *detected*. The cause of the error is not necessarily near by! Use the techniques in the previous section to find the error.

If the error was detected while compiling a form that had been read successfully, then point is located at the end of the form. In this case, it can't localize the error precisely, but can still show you which function to check.

Edebug

Edebug is a source-level debugger for Emacs Lisp programs that provides the following features:

- Step through evaluation, stopping before and after each expression.
- Set conditional or unconditional breakpoints.
- Trace slow or fast stopping briefly at each stop point, or each breakpoint.
- Evaluate expressions as if outside of Edebug.
- Automatically reevaluate a list of expressions and display their results each time Edebug updates the display.
- Output trace info on function enter and exit.

The first three sections of this chapter should tell you enough about Edebug to enable you to use it.

Using Edebug

To debug a Lisp program with Edebug, you must first prepare the Lisp functions that you want to debug. See section Preparing Functions for Edebug.

Once a function is prepared, any call to the function activates Edebug. This involves entering a recursive edit which is a level of Edebug activation.

Activating Edebug may stop execution and let you step through the function, or it may continue execution while checking for debugging commands, depending on the selected Edebug execution mode. See section Edebug Modes.

Within Edebug, you normally view an Emacs buffer showing the source of the Lisp function you are debugging. We call this the Edebug buffer---but note that it is not always the same buffer, and it is not reserved for Edebug use.

An arrow at the left margin indicates the line where the function is executing. Point initially shows where within the line the function is executing, but this ceases to be true if you move point yourself.

If you prepare the definition of `fac` (shown below) for Edebug and then execute `(fac 3)`, here is what you normally see. Point is at the open-parenthesis before `if`.

```
(defun fac (n)
=>-!-(if (< 0 n)
       (* n (fac (1- n)))
    1))
```

The places within a function where Edebug can stop execution are called stop points. These occur both before and after each subexpression that is a list, and also after each variable reference. Stop points before variables are optional, under the control of the value of `edebug-stop-before-symbols`. Here we show with periods the stop points normally found in the function `fac`:

```
(defun fac (n)
  .(if .(< 0 n.).
       .(* n. .(fac (1- n.).).).
    1).)
```

While a buffer is the Edebug buffer, the special commands of Edebug are available in it, instead of many usual editing commands. Type ? to display a list of Edebug commands. In particular, you can exit the innermost Edebug activation level with C-], and you can return all the way to top level with q.

For example, you can type the Edebug command SPC to execute until the next stop point. If you type SPC once after entry to `fac`, here is the state that you get:

```
(defun fac (n)
=>(if -!-(< 0 n)
       (* n (fac (1- n)))
```


```
    1))
```

When Edebug stops execution after an expression, it displays the expression's value in the echo area. Use the r command to display the value again later.

While Edebug is active, it catches all errors (if `debug-on-error` is non-`nil`) and quits (if `debug-on-quit` is non-`nil`) instead of the standard debugger. When this happens, Edebug displays the last stop point that it knows about. This may be the location of a call to a function which was not prepared for Edebug debugging, within which the error actually occurred.

Preparing Functions for Edebug

In order to use Edebug to debug a function, you must first prepare the function. Preparing a function inserts additional code into it which invokes Edebug at the proper places.

Any call to an Edebug-prepared function activates Edebug. This may or may not stop execution, depending on the Edebug execution mode in use. Some Edebug modes only update the display to indicate the progress of the evaluation without stopping execution. The default initial Edebug mode is `step` which does stop execution. See section Edebug Modes.

Once you have loaded Edebug, the command C-M-x is redefined so that when used on a function or macro definition, it prepares the function or macro if given a prefix argument. If the variable `edebug-all-defuns` is non-`nil`, that inverts the meaning of the prefix argument: then C-M-x prepares the function or macro *unless* it has a prefix argument. The default value of `edebug-all-defuns` is `nil`. The command M-x edebug-all-defuns toggles the value of the variable `edebug-all-defuns`.

If `edebug-all-defuns` is non-`nil`, then the commands `eval-region` and `eval-current-buffer` also prepare any functions and macros whose definitions they evaluate.

Loading a file does not prepare functions and macros for Edebug.

See section Evaluation for discussion of other evaluation functions available inside of Edebug.

Edebug Modes

Edebug supports several execution modes for running the program you are debugging. We call these alternatives Edebug modes; do not confuse them with major modes or minor modes. The current Edebug mode determines how Edebug displays the progress of the evaluation, whether it stops at each stop point, or continues to the next breakpoint, for example.

Normally, you specify the Edebug mode for execution by typing a command to continue the program in a certain mode. Here is a table of these commands. All except for S resume execution of the program, at least for a certain distance.

S

Stop: don't execute any more of the program for now, just wait for more Edebug commands.

SPC

Step: stop at the next stop point encountered.

t

Trace: pause one second at each Edebug stop point.

T

Rapid trace: mention each stop point, but don't actually pause.

g

Go: run until the next breakpoint. See section Breakpoints.

c

Continue: pause for one second at each breakpoint, but don't stop.

C

Continue: mention each breakpoint, but don't actually pause.

G

Non-stop: ignore breakpoints. You can still stop the program by typing S.

In general, the execution modes earlier in the above list run the program more slowly or stop sooner.

When you enter a new Edebug level, the mode comes from the value of the variable `edebug-initial-mode`. By default, this specifies step mode. If the mode thus specified is not stop mode, then the Edebug level executes the program (or part of it).

While executing or tracing, you can interrupt the execution by typing any Edebug command. Edebug stops the program at the next stop point and then executes the command that you typed. For example, typing t during execution switches to trace mode at the next stop point.

You can use the S command to stop execution without doing anything else.

If your function happens to read input, a character you hit intending to interrupt execution may be read by the function instead. You can avoid such unintended results by paying attention to when your program wants input.

Keyboard macros containing the commands in this section do not completely work: exiting from Edebug, to resume the program, loses track of the keyboard macro. This is not easy to fix.

Stepping

f

Run the program forward over one expression. More precisely, set a temporary breakpoint at the position that C-M-f would reach, then execute in go mode so that the program will stop at breakpoints. See section Breakpoints for the details on breakpoints.

With a prefix argument n, the temporary breakpoint is placed n sexps beyond point. If the containing list ends before n more elements, then the place to stop is after the containing expression.

Be careful that the position C-M-f finds is a place that the program will really get to; this may not

be true in a `condition-case`, for example.

This command does `forward-sexp` starting at point rather than the stop point, thus providing more flexibility. If you want to execute one expression from the current stop point, type w first, to move point there.

o

Run the program until the end of the containing sexp. If the containing sexp is the top level defun, run until just before the function returns. If that is where you are now, return from the function and then stop.

This command does not exit the currently executing function unless you are positioned after the last sexp of the function.

If the program does a non-local exit, it may fail to reach the temporary breakpoint that this command sets.

i

Step into the function about to be called. Use this command before any of the arguments of the function call are evaluated, since otherwise it is too late.

One undesirable side effect of using `edebug-step-in` is that the next time the stepped-into function is called, Edebug will be called there as well.

h

Proceed to the stop point near where point is. This uses a temporary breakpoint.

The f command runs the program forward over one expression. More precisely, set a temporary breakpoint at the position that C-M-f would reach, then execute in go mode so that the program will stop at breakpoints. See section Breakpoints for the details on breakpoints.

With a prefix argument n, the temporary breakpoint is placed n sexps beyond point. If the containing list ends before n more elements, then the place to stop is after the containing expression.

Be careful that the position C-M-f finds is a place that the program will really get to; this may not be true in a `condition-case`, for example.

The f command uses the existing value of point as the basis for setting the breakpoint, because that is more flexible. To execute one expression *from the current stop point*, type w and then f.

The o command continues "out of" an expression. It places a temporary breakpoints at the end of the containing sexp. If the containing sexp is the top level defun, it continues until just before the function returns. If that is where you are now, it returns from the function and then stops.

This command does not exit the currently executing function unless you are positioned after the last sexp of the function.

The i command steps into the function about to be called. Use this command before any of the arguments of the function call are evaluated, since otherwise it is too late.

One undesirable side effect of using i is that the next time the stepped-into function is called, Edebug will

be called there as well.

The h command proceeds to the stop point near where point is, using a temporary breakpoint.

All the commands in this section may fail to work as expected in case of nonlocal exit, because a nonlocal exit can bypass the temporary breakpoint where you expected the program to stop.

Miscellaneous

Some miscellaneous commands are described here.

C-]

Abort one level of Edebug activity.

q

Return to the top level editor command loop. This exits all recursive editing levels, including all levels of Edebug activity.

r

Redisplay the result of the previous expression in the echo area.

d

Display a backtrace, excluding Edebug's own functions for clarity.

You cannot use debugger commands in the backtrace buffer in Edebug as you would in the standard debugger.

The backtrace buffer is killed automatically when you continue execution.

Breakpoints

While using Edebug, you can specify breakpoints in the program you are testing: points where execution should stop. You can set a breakpoint at any stop point, as defined in section Using Edebug---even before a symbol. For setting and unsetting breakpoints, the stop point that is affected is the first one at or after point in the Edebug buffer. Here are the Edebug commands for breakpoints:

b

Set a breakpoint at the stop point at or after point. If you use a prefix argument, the breakpoint is temporary (it turns off the first time it stops the program).

u

Unset the breakpoint (if any) at the stop point at or after the current point.

x cond RET

Set a conditional breakpoint which stops the program only if cond evaluates to a non-`nil` value. If you use a prefix argument, the breakpoint is temporary (it turns off the first time it stops the program).

B

Move point to the next breakpoint in the current function definition.

While in Edebug, you can set a breakpoint with b (`edebug-set-breakpoint`) and unset one wi (`edebug-unset-breakpoint`). First you must move point to a position at or before the desired Edebug stop point, then hit the key to change the breakpoint. Unsetting a breakpoint that has not been set does nothing.

Reevaluating the function with `edebug-defun` clears all breakpoints in the function.

A conditional breakpoint tests a condition each time the program gets there, to decide whether to stop. To set a conditional breakpoint, use x, and specify the condition expression in the minibuffer.

You can make both conditional and unconditional breakpoints temporary by using a prefix arg to the command to set the breakpoint. After breaking at a temporary breakpoint, it is automatically cleared.

Edebug always stops or pauses at a breakpoint except when the Edebug mode is Go-nonstop. In that mode, it ignores breakpoints entirely.

To find out where your breakpoints are, use the B (`edebug-next-breakpoint`) command, which moves point to the next breakpoint in the function following point, or to the first breakpoint if there are no following breakpoints. This command does not continue execution--it just moves point in the buffer.

Views

These Edebug commands let you view aspects of the buffer and window status that obtained before entry to Edebug.

v
: View the outside window configuration.

p
: Temporarily display the outside current buffer with point at its outside position.

w
: Switch back to the buffer showing the currently executing function, and move point back to the current stop point.

W
: Forget the saved outside window configuration--so that the current window configuration will remain unchanged when you next exit Edebug (by continuing the program). Also toggle the `edebug-save-windows` variable.

Evaluation

While within Edebug, you can evaluate expressions "as if" Edebug were not running. Edebug tries to be invisible to the expression's evaluation.

e exp RET
: Evaluate expression exp in the context outside of Edebug. That is, Edebug tries to avoid altering the effect of exp.

M-ESC exp RET

Evaluate expression exp in the context of Edebug itself.

C-x C-e

Evaluate the expression in the buffer before point, in the context outside of Edebug.

Evaluation List Buffer

You can use the evaluation list buffer, called `*edebug*', to evaluate expressions interactively. You can also set up the evaluation list of expressions to be evaluated automatically each time Edebug is reentered.

E

Switch to the evaluation list buffer `*edebug*'.

In the `*edebug*' buffer you can use the commands of Lisp Interaction as well as these special commands:

LFD

Evaluate the expression before point, in the context outside of Edebug, and insert the value in the buffer.

C-x C-e

Evaluate the expression before point, in the context outside of Edebug.

C-c C-u

Build a new evaluation list from the first expression of each group, reevaluate and redisplay. Groups are separated by a line starting with a comment.

C-c C-d

Delete the evaluation list group that point is in.

C-c C-w

Switch back to the Edebug buffer at the current stop point.

You can evaluate expressions in the evaluation list window with LFD or C-x C-e, just as you would in `*scratch*'; but they are evaluated in the context outside of Edebug.

The expressions you enter interactively (and their results) are lost when you continue execution of your function unless you add them to the evaluation list with C-c C-u (`edebug-update-eval-list`). This command builds a new list from the first expression of each evaluation list group. Groups are separated by a line starting with a comment.

When the evaluation list is redisplayed, each expression is displayed followed by the result of evaluating it, and a comment line. If an error occurs during an evaluation, the error message is displayed in a string as if it were the result. Therefore expressions that use variables not currently valid do not interrupt your debugging.

Here is an example of what the evaluation list window looks like after several expressions have been added to it:

```
(current-buffer)
```


```
#<buffer *scratch*>
;-----------------------------------------------------------------
(point-min)
1
;-----------------------------------------------------------------
(point-max)
2
;-----------------------------------------------------------------
edebug-outside-point-max
"Symbol's value as variable is void: edebug-outside-point-max"
;-----------------------------------------------------------------
(recursion-depth)
0
;-----------------------------------------------------------------
this-command
eval-last-sexp
;-----------------------------------------------------------------
```

To delete a group, move point into it and type C-c C-d (`edebug-delete-eval-item`), or simply delete the text for it and update the evaluation list with C-c C-u. When you add a new group, be sure to add a comment at the beginning.

After selecting `*edebug*', you can return to the source code buffer (the Edebug buffer) with C-c C-w. The `*edebug*` buffer is killed when you continue execution of your function, and recreated next time it is needed.

Printing

If the results of your expressions contain circular references to other parts of the same structure, you can print them more usefully with the `` `custom-print' ``.

To load the package and activate custom printing only for Edebug, simply use the command `edebug-install-custom-print-funcs`. Then set the variable `print-circle` to enable special handling of circular structure. To restore the standard print functions, use `edebug-reset-print-funcs`.

The Outside Context

Edebug tries to be transparent to the program you are debugging, but it does not succeed completely. In addition, most evaluations you do within Edebug (see section Evaluation) occur in the same outside context which is temporarily restored for the evaluation. This section explains precisely how use Edebug fails to be completely transparent.

Just Checking

Whenever Edebug is entered just to think about whether to take some action, it needs to save and restore

certain data.

- `max-lisp-eval-depth` and `max-specpdl-size` are both incremented for each `edebug-enter` call so that your code should not be impacted by Edebug frames on the stack.
- The state of keyboard macro execution is saved and cleared out.

Outside Window Configuration

When Edebug needs to display something (e.g., in trace mode), it saves the current window configuration from "outside" Edebug (see section Window Configurations). When you exit Edebug (by continuing the program), it restores the previous window configuration.

Emacs redisplays only when it pauses. Usually, when you continue Edebug, the program comes back into Edebug at a breakpoint or after stepping, without pausing or reading input in between. In such cases, Emacs never gets a chance to redisplay the "outside" configuration. What you see is the window configuration for within Edebug, with no interruption.

The window configuration proper does not include which buffer is current or where point and mark are in the current buffer, but Edebug saves and restores these also.

Entry to Edebug for displaying something also saves and restores the following data. (Some of these variables are deliberately not restored if an error or quit signal occurs.)

- The position of point in the Edebug buffer is saved and restored if the outside current buffer is the same as the Edebug buffer.
- The outside window configuration, as described above, is saved and restored if `edebug-save-windows` is non-`nil`.
- The current buffer, and point and mark in the current buffer are normally saved and restored even if the current buffer is the same as the Edebug buffer.
- The value of point in each displayed buffers is saved and restored if `edebug-save-displayed-buffer-points` is non-`nil`.
- The variables `overlay-arrow-position` and `overlay-arrow-string` are saved and restored. This permits recursive use of Edebug, and use of Edebug while using GUD.
- `cursor-in-echo-area` is locally bound to `nil` so that the cursor shows up in the window.

Recursive Edit

When Edebug is entered and actually reads commands from the user, it saves (and later restores) these additional data:

- The current match data, for whichever buffer was current.
- `last-command`, `this-command`, `last-command-char`, and `last-input-char`. Commands used within Edebug do not affect these variables outside of Edebug.

  But note that it is not possible to preserve the status reported by `(this-command-keys)` and the variable `unread-command-char`.
- `standard-output` and `standard-input`.

Side Effects

Edebug operation unavoidably alters some data in Emacs, and this can interfere with debugging certain programs.

- Lisp stack usage is increased, but the limits, `max-lisp-eval-depth` and `max-specpdl-size`, are also increased proportionally.
- The key sequence returned by `this-command-keys` is changed by executing commands within Edebug and there appears to be no way to reset the key sequence from Lisp.
- Edebug cannot save and restore the value of `unread-command-char` or `unread-command-events`. Entering Edebug while these variables have nontrivial values can interfere with execution of the program you are debugging.
- Complex commands executed while in Edebug are added to the variable `command-history`. In rare cases this can alter execution.
- Within Edebug, the recursion depth appears one deeper than the recursion depth outside Edebug.
- Horizontal scrolling of the Edebug buffer is not recovered.

Macro Calls

When Edebug prepares for stepping through an expression that uses a Lisp macro, it needs additional advice to do the job properly. This is because there is no way to tell which parts of the macro call are forms to be evaluated. You must explain the format of calls to each macro to enable Edebug to handle it. To do this, use `def-edebug-form-spec` to define the format of calls to a given macro.

Macro: **def-edebug-form-spec** *macro argpattern*

Specify which parts of a call to macro macro are subexpressions to be evaluated. The second argument, argpattern, details what the argument list looks like.

Here is a table of the possibilities for argpattern and its subexpressions:

`t`
: A list of any number of evaluated arguments.

`0`
: A list of unevaluated arguments.

`sexp`
: A single unevaluated object.

`form`
: A single evaluated expression.

`symbolp`
: An unevaluated symbol.

`integerp`
: An unevaluated number.

`stringp`

An unevaluated string.

`vectorp`
: An unevaluated vector.

`atom`
: An unevaluated object that is not a cons cell.

`function`
: A function argument: a quoted symbol, a quoted lambda expression, or a form (that should evaluate to a function or lambda expression). Edebug treats the body of a lambda expression treated as evaluated.

`function`
: A function serves as a predicate--it designates the set of possible arguments for which it would return non-`nil`.

`'object`
: The precise object object, treated as unevaluated.

`(patterns)`
: A list whose elements are described by patterns. A sublist of the same format as the top level, processed recursively.

`[patterns]`
: A sequence of arguments that are described by patterns.

`&optional`
: This symbol serves as a flag saying that all following elements in the specification list at this level are optional. They may or may not match arguments; as soon as one does not match, processing of the specification list at this level terminates. To make just one item optional, use `[&optional pattern]`.

`&rest`
: This symbol serves as a flag saying that the following elements in the specification list at this level may be repeated, in order, zero or more times. Only one `&rest` may appear at the same level of a specification list, and `&rest` must not be followed by `&optional`.

  To specify repetition of certain types of arguments, followed by dissimilar arguments, use `[&rest patterns...]`.

`&or`
: This symbol serves as an operator saying that the following elements in the specification list at this level are alternatives. To group two or more list elements as one alternative, bracket them in `[...]`. Only one `&or` may appear in a list, and it may not be followed by `&optional` or `&rest`. One of the alternatives must match, unless the `&or` is preceded by `&optional` or `&rest`.

If the actual arguments of a macro call fail to match the specification, taking account of alternatives, optional arguments and repeated arguments, Edebug reports a syntax error in use of the macro.

The combination of backtracking, `&optional`, `&rest`, `&or`, and `[...]` for grouping provides the equivalent of regular expressions. The `(...)` lists require balanced parentheses, which is the only context free (finite state with stack) construct supported.

Here are some examples of using `def-edebug-form-spec`. First, for the `let` special form:

```
(def-edebug-form-spec let
  '((&rest
    &or symbolp (symbolp &optional form))
   &rest form))
```

Here's the spec for the `for` loop macro (see section Common Problems Using Macros) and for the `case` and `do` macros in `` `cl.el' ``:

```
(def-edebug-form-spec for
  '(symbolp 'from form 'to form 'do &rest form))

(def-edebug-form-spec case
  '(form &rest (sexp form)))

(def-edebug-form-spec do
  '((&rest &or symbolp (symbolp &optional form form))
    (form &rest form)
    &rest body))
```

Finally, the functions `mapcar`, `mapconcat`, `mapatoms`, `apply`, and `funcall` all take function arguments, and Edebug defines specifications for them. Here's one example:

```
(def-edebug-form-spec apply '(function &rest form))
```

The backquote (`) macro results in an expression that is not necessarily evaluated. Edebug cannot step through code generated by use of backquote.

Edebug Options

These options affect the behavior of Edebug:

User Option: **edebug-all-defuns**

If non-`nil`, normal evaluation of `defun` and `defmacro` forms prepares the functions and macros for stepping with Edebug. This applies to `eval-defun`, `eval-region` and `eval-current-buffer`.

The default value is `nil`.

User Option: **edebug-stop-before-symbols**

If non-`nil`, Edebug places stop points before symbols as well as after.

This option takes effect for a function when you prepare it for stepping with Edebug. Changing the option's value during execution of Edebug has no effect on the functions already set up for Edebug execution.

User Option: **edebug-save-windows**

If non-`nil`, save and restore window configuration on Edebug calls. It takes some time to save and restore, so if your program does not care what happens to the window configurations, it is better to set this variable to `nil`.

The default value is `t`.

User Option: **edebug-save-point**

If non-`nil`, Edebug saves and restores point and the mark in source code buffers. The default value is `t`.

User Option: **edebug-save-displayed-buffer-points**

If non-`nil`, save and restore point in all buffers when entering Edebug mode.

Saving and restoring point in other buffers is necessary if you are debugging code that changes the point of a buffer which is displayed in a non-selected window. If Edebug or the user then selects the window, the buffer's point will be changed to the window's point.

Saving and restoring is an expensive operation since it visits each window and each displayed buffer twice for each Edebug call, so it is best to avoid it if you can.

The default value is `nil`.

User Option: **edebug-initial-mode**

If this variable is non-`nil`, it specifies an Edebug mode to start in each time the program enters a new Edebug recursive-edit level. Possible values are `step`, `go`, `Go-nonstop`, `trace`, `Trace-fast`, `continue`, and `Continue-fast`.

The default value is `step`.

User Option: **edebug-trace**

Non-`nil` means display a trace of function entry and exit. Tracing output is displayed in a buffer named `*edebug-trace*', one function entry or exit per line, indented by the recursion level. You can customize this display by replacing the functions `edebug-print-trace-entry` and `edebug-print-trace-exit`.

The default value is `nil`.

Go to the previous, next section.

Go to the previous, next section.

Reading and Printing Lisp Objects

Printing and reading are the operations of converting Lisp objects to textual form and vice versa. They use the printed representations and read syntax described in section Lisp Data Types.

This chapter describes the Lisp functions for reading and printing. It also describes streams, which specify where to get the text (if reading) or where to put it (if printing).

Introduction to Reading and Printing

Reading a Lisp object means parsing a Lisp expression in textual form and producing a corresponding Lisp object. This is how Lisp programs get into Lisp from files of Lisp code. We call the text the read syntax of the object. For example, reading the text `(a . 5)' returns a cons cell whose CAR is `a` and whose CDR is the number 5.

Printing a Lisp object means producing text that represents that object--converting the object to its printed representation. Printing the cons cell described above produces the text `(a . 5)'.

Reading and printing are more or less inverse operations: printing the object that results from reading a given piece of text often produces the same text, and reading the text that results from printing an object usually produces a similar-looking object. For example, printing the symbol `foo` produces the text `foo', and reading that text returns the symbol `foo`. Printing a list whose elements are `a` and `b` produces the text `(a b)', and reading that text produces a list (but not the same list) with elements are `a` and `b`.

However, these two operations are not precisely inverses. There are two kinds of exceptions:

- Printing can produce text that cannot be read. For example, buffers, windows, subprocesses and markers print into text that starts with `#'; if you try to read this text, you get an error. There is no way to read those data types.
- One object can have multiple textual representations. For example, `1' and `01' represent the same integer, and `(a b)' and `(a . (b))' represent the same list. Reading will accept any of the alternatives, but printing must choose one of them.

Input Streams

Most of the Lisp functions for reading text take an input stream as an argument. The input stream specifies where or how to get the characters of the text to be read. Here are the possible types of input stream:

buffer

> The input characters are read from buffer, starting with the character directly after point. Point advances as characters are read.

marker

The input characters are read from the buffer that marker is in, starting with the character direc after the marker. The marker position advances as characters are read. The value of point in the buffer has no effect when the stream is a marker.

string

The input characters are taken from string, starting at the first character in the string and using as many characters as required.

function

The input characters are generated by function, one character per call. Normally function is called with no arguments, and should return a character.

Occasionally function is called with one argument (always a character). When that happens, function should save the argument and arrange to return it on the next call. This is called unreading the character; it happens when the Lisp reader reads one character too many and want to "put it back where it came from".

`t`

`t` used as a stream means that the input is read from the minibuffer. In fact, the minibuffer is invoked once and the text given by the user is made into a string that is then used as the input stream.

`nil`

`nil` used as a stream means that the value of `standard-input` should be used instead; that value is the default input stream, and must be a non-`nil` input stream.

symbol

A symbol as output stream is equivalent to the symbol's function definition (if any).

Here is an example of reading from a stream which is a buffer, showing where point is located before and after:

```
---------- Buffer: foo ----------
This-!- is the contents of foo.
---------- Buffer: foo ----------

(read (get-buffer "foo"))
     => is
(read (get-buffer "foo"))
     => the

---------- Buffer: foo ----------
This is the-!- contents of foo.
---------- Buffer: foo ----------
```

Note that the first read skips a space at the beginning of the buffer. Reading skips any amount of whitespace preceding the significant text.

In Emacs 18, reading a symbol discarded the delimiter terminating the symbol. Thus, point would end up

at the beginning of `contents' rather than after `the'. The Emacs 19 behavior is superior because it correctly handles input such as `bar(foo)' where the delimiter that ends one object is needed as the beginning of another object.

Here is an example of reading from a stream that is a marker, initialized to point at the beginning of the buffer shown. The value read is the symbol `This`.

```
---------- Buffer: foo ----------
This is the contents of foo.
---------- Buffer: foo ----------

(setq m (set-marker (make-marker) 1 (get-buffer "foo")))
     => #<marker at 1 in foo>
(read m)
     => This
m
     => #<marker at 6 in foo>   ;; After the first space.
```

Here we read from the contents of a string:

```
(read "(When in) the course")
     => (When in)
```

The following example reads from the minibuffer. The prompt is: `Lisp expression: '. (That is always the prompt used when you read from the stream `t`.) The user's input is shown following the prompt.

```
(read t)
     => 23
---------- Buffer: Minibuffer ----------
Lisp expression: 23 RET
---------- Buffer: Minibuffer ----------
```

Finally, here is an example of a stream that is a function, named `useless-stream`. Before we use the stream, we initialize the variable `useless-list` to a list of characters. Then each call to the function `useless-stream` obtains the next characters in the list or unreads a character by adding it to the front of the list.

```
(setq useless-list (append "XY()" nil))
     => (88 89 40 41)

(defun useless-stream (&optional unread)
  (if unread
      (setq useless-list (cons unread useless-list))
    (prog1 (car useless-list)
           (setq useless-list (cdr useless-list)))))
```


```
     => useless-stream
```

Now we read using the stream thus constructed:

```
(read 'useless-stream)
     => XY

useless-list
     => (41)
```

Note that the close parenthesis remains in the list. The reader has read it, discovered that it ended the input, and unread it. Another attempt to read from the stream at this point would get an error due to the unmatched close parenthesis.

Function: **get-file-char**

This function is used internally as an input stream to read from the input file opened by the function `load`. Don't use this function yourself.

Input Functions

This section describes the Lisp functions and variables that pertain to reading.

In the functions below, stream stands for an input stream (see the previous section). If stream is `nil` or omitted, it defaults to the value of `standard-input`.

An `end-of-file` error results if an unterminated list or vector is found.

Function: **read** *&optional stream*

This function reads one textual Lisp expression from stream, returning it as a Lisp object. This is the basic Lisp input function.

Function: **read-from-string** *string &optional start end*

This function reads the first textual Lisp expression from the text in string. It returns a cons cell whose CAR is that expression, and whose CDR is an integer giving the position of the next remaining character in the string (i.e., the first one not read).

If start is supplied, then reading begins at index start in the string (where the first character is at index 0). If end is also supplied, then reading stops at that index as if the rest of the string were not there.

For example:

```
(read-from-string "(setq x 55) (setq y 5)")
     => ((setq x 55) . 11)
(read-from-string "\"A short string\"")
     => ("A short string" . 16)
```


```
;; Read starting at the first character.
(read-from-string "(list 112)" 0)
     => ((list 112) . 10)
;; Read starting at the second character.
(read-from-string "(list 112)" 1)
     => (list . 6)
;; Read starting at the seventh character,
;;   and stopping at the ninth.
(read-from-string "(list 112)" 6 8)
     => (11 . 8)
```

Variable: **standard-input**

This variable holds the default input stream: the stream that `read` uses when the stream argument is `nil`.

Output Streams

An output stream specifies what to do with the characters produced by printing. Most print functions accept an output stream as an optional argument. Here are the possible types of output stream:

buffer

The output characters are inserted into buffer at point. Point advances as characters are inserted.

marker

The output characters are inserted into the buffer that marker is in at the marker position. The position advances as characters are inserted. The value of point in the buffer has no effect when the stream is a marker.

function

The output characters are passed to function, which is responsible for storing them away. It is called with a single character as argument, as many times as there are characters to be output, and is free to do anything at all with the characters it receives.

`t`

The output characters are displayed in the echo area.

`nil`

`nil` specified as an output stream means that the value of `standard-output` should be used as the output stream; that value is the default output stream, and must be a non-`nil` output stream.

symbol

A symbol as output stream is equivalent to the symbol's function definition (if any).

Here is an example of a buffer used as an output stream. Point is initially located as shown immediately before the `h' in `the'. At the end, point is located directly before that same `h'.

```
---------- Buffer: foo ----------
This is t-!-he contents of foo.
```


```
---------- Buffer: foo ----------

(print "This is the output" (get-buffer "foo"))
     => "This is the output"

---------- Buffer: foo ----------
This is t
"This is the output"
-!-he contents of foo.
---------- Buffer: foo ----------
```

Now we show a use of a marker as an output stream. Initially, the marker points in buffer `foo`, between the `t' and the `h' in the word `the'. At the end, the marker has been advanced over the inserted text so that it still points before the same `h'. Note that the location of point, shown in the usual fashion, has no effect.

```
---------- Buffer: foo ----------
"This is the -!-output"
---------- Buffer: foo ----------

m
     => #<marker at 11 in foo>

(print "More output for foo." m)
     => "More output for foo."

---------- Buffer: foo ----------
"This is t
"More output for foo."
he -!-output"
---------- Buffer: foo ----------

m
     => #<marker at 35 in foo>
```

The following example shows output to the echo area:

```
(print "Echo Area output" t)
     => "Echo Area output"
---------- Echo Area ----------
"Echo Area output"
---------- Echo Area ----------
```

Finally, we show an output stream which is a function. The function `eat-output` takes each character that it is given and conses it onto the front of the list `last-output` (see section Building Cons Cells and Lists). At the end, the list contains all the characters output, but in reverse order.


```
(setq last-output nil)
     => nil

(defun eat-output (c)
  (setq last-output (cons c last-output)))
     => eat-output

(print "This is the output" 'eat-output)
     => "This is the output"

last-output
     => (10 34 116 117 112 116 117 111 32 101 104
    116 32 115 105 32 115 105 104 84 34 10)
```

Now we can put the output in the proper order by reversing the list:

```
(concat (nreverse last-output))
     => "
\"This is the output\"
"
```

Output Functions

This section describes the Lisp functions for printing Lisp objects.

Some of the Emacs printing functions add quoting characters to the output when necessary so that it can be read properly. The quoting characters used are `\' and `"'; they are used to distinguish strings from symbols, and to prevent punctuation characters in strings and symbols from being taken as delimiters. See section Printed Representation and Read Syntax, for full details. You specify quoting or no quoting by the choice of printing function.

If the text is to be read back into Lisp, then it is best to print with quoting characters to avoid ambiguity. Likewise, if the purpose is to describe a Lisp object clearly for a Lisp programmer. However, if the purpose of the output is to look nice for humans, then it is better to print without quoting.

Printing a self-referent Lisp object requires an infinite amount of text. In certain cases, trying to produce this text leads to a stack overflow. Emacs detects such recursion and prints `#level' instead of recursively printing an object already being printed. For example, here `#0' indicates a recursive reference to the object at level 0 of the current print operation:

```
(setq foo (list nil))
     => (nil)
(setcar foo foo)
     => (#0)
```

In the functions below, stream stands for an output stream. (See the previous section for a description of output streams.) If stream is `nil` or omitted, it defaults to the value of `standard-output`.

Function: **print** *object &optional stream*

The `print` is a convenient way of printing. It outputs the printed representation of object to stream, printing in addition one newline before object and another after it. Quoting characters are used. `print` returns object. For example:

```
(progn (print 'The\ cat\ in)
       (print "the hat")
       (print " came back"))
     -|
     -| The\ cat\ in
     -|
     -| "the hat"
     -|
     -| " came back"
     -|
     => " came back"
```

Function: **prin1** *object &optional stream*

This function outputs the printed representation of object to stream. It does not print any spaces or newlines to separate output as `print` does, but it does use quoting characters just like `print`. It returns object.

```
(progn (prin1 'The\ cat\ in)
       (prin1 "the hat")
       (prin1 " came back"))
     -| The\ cat\ in"the hat"" came back"
     => " came back"
```

Function: **princ** *object &optional stream*

This function outputs the printed representation of object to stream. It returns object.

This function is intended to produce output that is readable by people, not by `read`, so quoting characters are not used and double-quotes are not printed around the contents of strings. It does not add any spacing between calls.

```
(progn
  (princ 'The\ cat)
  (princ " in the \"hat\""))
     -| The cat in the "hat"
     => " in the \"hat\""
```

Function: **terpri** *&optional stream*

This function outputs a newline to stream. The name stands for "terminate print".

Function: **write-char** *character &optional stream*

This function outputs character to stream. It returns character.

Function: **prin1-to-string** *object &optional noescape*

This function returns a string containing the text that `prin1` would have printed for the same argument.

```
(prin1-to-string 'foo)
     => "foo"
(prin1-to-string (mark-marker))
     => "#<marker at 2773 in strings.texi>"
```

If noescape is non-`nil`, that inhibits use of quoting characters in the output. (This argument is supported in Emacs versions 19 and later.)

```
(prin1-to-string "foo")
     => "\"foo\""
(prin1-to-string "foo" t)
     => "foo"
```

See `format`, in section Conversion of Characters and Strings, for other ways to obtain the printed representation of a Lisp object as a string.

Variables Affecting Output

Variable: **standard-output**

The value of this variable is the default output stream, used when the stream argument is omitted or `nil`.

Variable: **print-escape-newlines**

If this variable is non-`nil`, then newline characters in strings are printed as `\n'. Normally they are printed as actual newlines.

This variable affects the print functions `prin1` and `print`, as well as everything that uses them. It does not affect `princ`. Here is an example using `prin1`:

```
(prin1 "a\nb")
     -| "a
     -| b"
     => "a
     => b"

(let ((print-escape-newlines t))
```


```
(prin1 "a\nb"))
   -| "a\nb"
   => "a
   => b"
```

In the second expression, the local binding of `print-escape-newlines` is in effect during the call to `prin1`, but not during the printing of the result.

Variable: **print-length**

The value of this variable is the maximum number of elements of a list that will be printed. If the list being printed has more than this many elements, then it is abbreviated with an ellipsis.

If the value is `nil` (the default), then there is no limit.

```
(setq print-length 2)
     => 2
(print '(1 2 3 4 5))
     -| (1 2 ...)
     => (1 2 ...)
```

Variable: **print-level**

The value of this variable is the maximum depth of nesting of parentheses that will be printed. Any list or vector at a depth exceeding this limit is abbreviated with an ellipsis. A value of `nil` (which is the default) means no limit.

This variable exists in version 19 and later versions.

Go to the previous, next section.

Go to the previous, next section.

Minibuffers

A minibuffer is a special buffer that Emacs commands use to read arguments more complicated than the single numeric prefix argument. These arguments include file names, buffer names, and command names (as in M-x). The minibuffer is displayed on the bottom line of the screen, in the same place as the echo area, but only while it is in use for reading an argument.

Introduction to Minibuffers

In most ways, a minibuffer is a normal Emacs buffer. Most operations *within* a buffer, such as editing commands, work normally in a minibuffer. However, many operations for managing buffers do not apply to minibuffers. The name of a minibuffer always has the form ` *Minibuf-number', and it cannot be changed. Minibuffers are displayed only in special windows used only for minibuffers; these windows always appear at the bottom of a frame. (Sometime frames have no minibuffer window, and sometimes a special kind of frame contains nothing but a minibuffer window; see section Minibuffers and Frames.)

The minibuffers window is normally a single line; you can resize it temporarily with the window sizing commands, but reverts to its normal size when the minibuffer is exited.

A recursive minibuffer may be created when there is an active minibuffer and a command is invoked that requires input from a minibuffer. The first minibuffer is named ` *Minibuf-0*'. Recursive minibuffers are named by incrementing the number at the end of the name. (The names begin with a space so that they won't show up in normal buffer lists.) Of several recursive minibuffers, the innermost (or most recently entered) is the active minibuffer. We usually call this "the" minibuffer. You can permit or forbid recursive minibuffers by setting the variable `enable-recursive-minibuffers` or by putting properties of that name on command symbols (see section Minibuffer Miscellany).

Like other buffers, a minibuffer may use any of several local keymaps (see section Keymaps); these contain various exit commands and in some cases completion commands. See section Completion.

- `minibuffer-local-map` is for ordinary input (no completion).
- `minibuffer-local-ns-map` is similar, except that SPC exits just like RET. This is used mainly for Mocklisp compatibility.
- `minibuffer-local-completion-map` is for permissive completion.
- `minibuffer-local-must-match-map` is for strict completion and for cautious completion.

Reading Text Strings with the Minibuffer

The minibuffer is usually used to read text which is returned as a string, but can also be used to read a Lisp object in textual form. The most basic primitive for minibuffer input is `read-from-minibuffer`.

Function: **read-from-minibuffer** *prompt-string &optional initial keymap read hist*

This function is the most general way to get input through the minibuffer. By default, it accepts arbitrary text and returns it as a string; however, if read is non-`nil`, then it uses `read` to convert the text into a Lisp object (see section Input Functions).

The first thing this function does is to activate a minibuffer and display it with prompt-string as the prompt. This value must be a string.

Then, if initial is a string; its contents are inserted into the minibuffer as initial contents. The text thus inserted is treated as if the user had inserted it; the user can alter it with Emacs editing commands.

The value of initial may also be a cons cell of the form `(string . position)`. This means to insert string in the minibuffer but put the cursor position characters from the beginning, rather than at the end.

If keymap is non-`nil`, that keymap is the local keymap to use while reading. If keymap is omitted or `nil`, the value of `minibuffer-local-map` is used as the keymap. Specifying a keymap is the most important way to customize minibuffer input for various applications including completion.

The argument hist specifies which history list variable to use for saving the input and for history commands used in the minibuffer. It defaults to `minibuffer-history`. See section Minibuffer History.

When the user types a command to exit the minibuffer, the current minibuffer contents are usually made into a string which becomes the value of `read-from-minibuffer`. However, if read is non-`nil`, `read-from-minibuffer` converts the result to a Lisp object, and returns that object, unevaluated.

Suppose, for example, you are writing a search command and want to record the last search string and provide it as a default for the next search. Suppose that the previous search string is stored in the variable `last-search-string`. Here is how you can read a search string while providing the previous string as initial input to be edited:

```
(read-from-minibuffer "Find string: " last-search-string)
```

Assuming the value of `last-search-string` is `No', and the user wants to search for `Nope', the interaction looks like this:

```
(setq last-search-string "No")

(read-from-minibuffer "Find string: " last-search-string)
---------- Buffer: Minibuffer ----------
Find string: No-!-
---------- Buffer: Minibuffer ----------
;; The user now types pe RET:
     => "Nope"
```

This technique is no longer preferred for most applications; it is usually better to use a history list.

Function: **read-string** *prompt &optional initial*

This function reads a string from the minibuffer and returns it. The arguments prompt and initial are as in `read-from-minibuffer`.

This is a simplified interface to the `read-from-minibuffer` function:

```
(read-string prompt initial)
==
(read-from-minibuffer prompt initial nil nil)
```

Variable: **minibuffer-local-map**

This is the default local keymap for reading from the minibuffer. It is the keymap used by the minibuffer for local bindings in the function `read-string`. By default, it makes the following bindings:

LFD
: `exit-minibuffer`

RET
: `exit-minibuffer`

C-g
: `abort-recursive-edit`

M-n and M-p
: `next-history-element` and `previous-history-element`

M-r
: `next-matching-history-element`

M-s
: `previous-matching-history-element`

Function: **read-no-blanks-input** *prompt &optional initial*

This function reads a string from the minibuffer, but does not allow whitespace characters as part of the input: instead, those characters terminate the input. The arguments prompt and initial are used as in `read-from-minibuffer`.

This is a simplified interface to the `read-from-minibuffer` function, and passes the value of the `minibuffer-local-ns-map` keymap as the keymap argument for that function. Since the keymap `minibuffer-local-ns-map` does not rebind C-q, it *is* possible to put a space into the string, by quoting it.

```
(read-no-blanks-input prompt initial)
==
(read-from-minibuffer prompt initial minibuffer-local-ns-map)
```

Variable: **minibuffer-local-ns-map**

This built-in variable is the keymap used as the minibuffer local keymap in the function

`read-no-blanks-input`. By default, it makes the following bindings:

LFD
: `exit-minibuffer`

SPC
: `exit-minibuffer`

TAB
: `exit-minibuffer`

RET
: `exit-minibuffer`

C-g
: `abort-recursive-edit`

?
: `self-insert-and-exit`

M-n and M-p
: `next-history-element` and `previous-history-element`

M-r
: `next-matching-history-element`

M-s
: `previous-matching-history-element`

Reading Lisp Objects with the Minibuffer

This section describes functions for reading Lisp objects with the minibuffer.

Function: **read-minibuffer** *prompt &optional initial*

This function reads a Lisp object in the minibuffer and returns it, without evaluating it. The arguments prompt and initial are used as in `read-from-minibuffer`; in particular, initial must be a string or `nil`.

This is a simplified interface to the `read-from-minibuffer` function:

```
(read-minibuffer prompt initial)
==
(read-from-minibuffer prompt initial nil t)
```

Here is an example in which we supply the string `"(testing)"` as initial input:

```
(read-minibuffer
 "Enter an expression: " (format "%s" '(testing)))
```


```
;; Here is how the minibuffer is displayed:

---------- Buffer: Minibuffer ----------
Enter an expression: (testing)-!-
---------- Buffer: Minibuffer ----------
```

The user can type RET immediately to use the initial input as a default, or can edit the input.

Function: **eval-minibuffer** *prompt &optional initial*

This function reads a Lisp expression in the minibuffer, evaluates it, then returns the result. The arguments prompt and initial are used as in `read-from-minibuffer`.

This function simply evaluates the result of a call to `read-minibuffer`:

```
(eval-minibuffer prompt initial)
==
(eval (read-minibuffer prompt initial))
```

Function: **edit-and-eval-command** *prompt form*

This function reads a Lisp expression in the minibuffer, and then evaluates it. The difference between this command and `eval-minibuffer` is that here the initial form is not optional and it is treated as a Lisp object to be converted to printed representation rather than as a string of text. It is printed with `prin1`, so if it is a string, double-quote characters (`"') appear in the initial text. See section Output Functions.

The first thing `edit-and-eval-command` does is to activate the minibuffer with prompt as the prompt. Then it inserts the printed representation of form in the minibuffer, and lets the user edit. When the user exits the minibuffer, the edited text is read with `read` and then evaluated. The resulting value becomes the value of `edit-and-eval-command`.

In the following example, we offer the user an expression with initial text which is a valid form already:

```
(edit-and-eval-command "Please edit: " '(forward-word 1))

;; After evaluating the preceding expression,
;;   the following appears in the minibuffer:

---------- Buffer: Minibuffer ----------
Please edit: (forward-word 1)-!-
---------- Buffer: Minibuffer ----------
```

Typing RET right away would exit the minibuffer and evaluate the expression, thus moving point forward one word. `edit-and-eval-command` returns `nil` in this example.

Minibuffer History

A minibuffer history list records previous minibuffer inputs so the user can reuse them conveniently. There are many separate history lists which contain different kinds of inputs. The Lisp programmer's job is to specify the right history list for each use of the minibuffer.

The basic minibuffer input functions `read-from-minibuffer` and `completing-read` both accept an optional argument named hist which is how you specify the history list. Here are the possible values:

variable

If you specify a variable (a symbol), that variable is the history list.

(variable . startpos)

If you specify a cons cell of this form, then variable is the history list variable, and startpos specifies the initial history position (an integer, counting from zero which specifies the most recent element of the history).

If you specify startpos, then you should also specify that element of the history as initial, for consistency.

If you don't specify hist, then the default history list `minibuffer-history` is used. For other standard history lists, see below. You can also create your own history list variable; just initialize it to `nil` before the first use. The value of the history list variable is a list of strings, most recent first.

Both `read-from-minibuffer` and `completing-read` add new elements to the history list automatically, and provide commands to allow the user to reuse items on the list. The only thing your program needs to do to use a history list is to initialize it and to pass its name to the input functions when you wish. But it is safe to modify the list by hand when the minibuffer input functions are not using it.

Variable: **minibuffer-history**

The default history list for minibuffer history input.

Variable: **query-replace-history**

A history list for arguments to `query-replace` (and similar arguments to other commands).

Variable: **file-name-history**

A history list for file name arguments.

Completion

Completion is a feature that fills in the rest of a name starting from an abbreviation for it. Completion works by comparing the user's input against a list of valid names and determining how much of the name is determined uniquely by what the user has typed.

For example, when you type C-x b (`switch-to-buffer`) and then type the first few letters of the

name of the buffer to which you wish to switch, and then type TAB (`minibuffer-complete`), Emacs extends the name as far as it can. Standard Emacs commands offer completion for names of symbols, files, buffers, and processes; with the functions in this section, you can implement completion for other kinds of names.

The `try-completion` function is the basic primitive for completion: it returns the longest determined completion of a given initial string, with a given set of strings to match against.

The function `completing-read` provides a higher-level interface for completion. A call to `completing-read` specifies how to determine the list of valid names. The function then activates the minibuffer with a local keymap that binds a few keys to commands useful for completion. Other functions provide convenient simple interfaces for reading certain kinds of names with completion.

Basic Completion Functions

Function: **try-completion** *string collection &optional predicate*

This function returns the longest common substring of all possible completions of string in collection. The value of collection must be an alist, an obarray, or a function which implements a virtual set of strings.

If collection is an alist (see section Association Lists), completion compares the CAR of each cons cell in it against string; if the beginning of the CAR equals string, the cons cell matches. If no cons cells match, `try-completion` returns `nil`. If only one cons cell matches, and the match is exact, then `try-completion` returns `t`. Otherwise, the value is the longest initial sequence common to all the matching strings in the alist.

If collection is an obarray (see section Creating and Interning Symbols), the names of all symbols in the obarray form the space of possible completions. They are tested and used just like the CARs of the elements of an association list. (The global variable `obarray` holds an obarray containing the names of all interned Lisp symbols.)

Note that the only valid way to make a new obarray is to create it empty and then add symbols to it one by one using `intern`. Also, you cannot intern a given symbol in more than one obarray.

If the argument predicate is non-`nil`, then it must be a function of one argument. It is used to test each possible match, and the match is accepted only if predicate returns non-`nil`. The argument given to predicate is either a cons cell from the alist (the CAR of which is a string) or else it is a symbol (*not* a symbol name) from the obarray.

It is also possible to use a function symbol as collection. Then the function is solely responsible for performing completion; `try-completion` returns whatever this function returns. The function is called with three arguments: string, predicate and `nil`. (The reason for the third argument is so that the same function can be used in `all-completions` and do the appropriate thing in either case.) See section Programmed Completion.

In the first of the following examples, the string `foo' is matched by three of the alist CARs. All of the matches begin with the characters `fooba', so that is the result. In the second example, there is only one

possible match, and it is exact, so the value is `t`.

```
(try-completion
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4)))
     => "fooba"

(try-completion "foo" '(("barfoo" 2) ("foo" 3)))
     => t
```

In the following example, numerous symbols begin with the characters `forw', and all of them begin with the word `forward'. In most of the symbols, this is followed with a `-', but not in all, so no more than `forward' can be completed.

```
(try-completion "forw" obarray)
     => "forward"
```

Finally, in the following example, only two of the three possible matches pass the predicate `test` (the string `foobaz' is too short). Both of those begin with the string `foobar'.

```
(defun test (s)
  (> (length (car s)) 6))
     => test
(try-completion
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
     'test)
     => "foobar"
```

Function: **all-completions** *string collection &optional predicate*

This function returns a list of all possible completions, instead of the longest substring they share. The parameters to this function are the same as to `try-completion`.

If collection is a function, it is called with three arguments: string, predicate and `t`, and `all-completions` returns whatever the function returns. See section Programmed Completion.

Here is an example, using the function `test` shown in the example for `try-completion`:

```
(defun test (s)
  (> (length (car s)) 6))
     => test

(all-completions
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 (function test))
```


```
=> ("foobar1" "foobar2")
```

Variable: **completion-ignore-case**

If the value of this variable is non-`nil`, Emacs does not consider case significant in completion.

The two functions `try-completion` and `all-completions` have nothing in themselves to do with minibuffers. However, completion is most often used there, which is why it is described in this chapter.

Programmed Completion

Sometimes it is not possible to create an alist or an obarray containing all the intended possible completions. In such a case, you can supply your own function to compute the completion of a given string. This is called programmed completion.

To use this feature, pass a symbol with a function definition as the collection argument to `completing-read`. This command arranges to pass the function along to `try-completion` and `all-completions`, which will then let your function do all the work.

The completion function should accept three arguments:

- The string to be completed.
- The predicate function to filter possible matches, or `nil` if none. Your function should call the predicate for each possible match and ignore the possible match if the predicate returns `nil`.
- A flag specifying the type of operation.

There are three flag values for three operations:

- `nil` specifies `try-completion`. The completion function should return the completion of the specified string, or `t` if the string is an exact match already, or `nil` if the string matches no possibility.
- `t` specifies `all-completions`. The completion function should return a list of all possible completions of the specified string.
- `lambda` specifies a test for an exact match. The completion function should return `t` if the specified string is an exact match for some possibility; `nil` otherwise.

It would be consistent and clean for completion functions to allow lambda expressions (lists which are functions) as well as function symbols as collection, but this is impossible. Lists as completion tables are already assigned another meaning--as alists. It would be unreliable to fail to handle an alist normally because it is also a possible function. So you must arrange for any function you wish to use for completion to be encapsulated in a symbol.

Emacs uses programmed completion when completing file names. See section File Name Completion.

Completion and the Minibuffer

This section describes the basic interface for reading from the minibuffer with completion.

Function: **completing-read** *prompt collection &optional predicate require-match initial hist*

This function reads a string in the minibuffer, assisting the user by providing completion. It activates the minibuffer with prompt prompt, which must be a string. If initial is non-`nil`, `completing-read` inserts it into the minibuffer as part of the input. Then it allows the user to edit the input, providing several commands to attempt completion.

The actual completion is done by passing collection and predicate to the function `try-completion`. This happens in certain commands bound in the local keymaps used for completion.

If require-match is `t`, the user is not allowed to exit unless the input completes to an element of collection. If require-match is neither `nil` nor `t`, then `completing-read` does not exit unless the input typed is itself an element of collection. To accomplish this, `completing-read` calls `read-minibuffer`. It uses the value of `minibuffer-local-completion-map` as the keymap if require-match is `nil`, and uses `minibuffer-local-must-match-map` if require-match is non-`nil`.

The argument hist specifies which history list variable to use for saving the input and for minibuffer history commands. It defaults to `minibuffer-history`. See section Minibuffer History.

Case is ignored when comparing the input against the possible matches if the built-in variable `completion-ignore-case` is non-`nil`. See section Basic Completion Functions.

For example:

```
(completing-read
 "Complete a foo: "
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 nil t "fo")

;; After evaluating the preceding expression,
;;   the following appears in the minibuffer:

---------- Buffer: Minibuffer ----------
Complete a foo: fo-!-
---------- Buffer: Minibuffer ----------
```

If the user then types DEL DEL b RET, `completing-read` returns `barfoo`.

The `completing-read` function binds three variables to pass information to the commands which actually do completion. Here they are:

`minibuffer-completion-table`

This variable is bound to the collection argument. It is passed to the `try-completion` function.

`minibuffer-completion-predicate`

This variable is bound to the predicate argument. It is passed to the `try-completion` function.

`minibuffer-completion-confirm`

This variable is bound to the require-match argument. It is used in the `minibuffer-complete-and-exit` function.

Minibuffer Commands That Do Completion

This section describes the keymaps, commands and user options used in the minibuffer to do completion.

Variable: **minibuffer-local-completion-map**

`completing-read` uses this value as the local keymap when an exact match of one of the completions is not required. By default, this keymap makes the following bindings:

?

`minibuffer-completion-help`

SPC

`minibuffer-complete-word`

TAB

`minibuffer-complete`

with other characters bound as in `minibuffer-local-map`.

Variable: **minibuffer-local-must-match-map**

`completing-read` uses this value as the local keymap when an exact match of one of the completions is required. Therefore, no keys are bound to `exit-minibuffer`, the command which exits the minibuffer unconditionally. By default, this keymap makes the following bindings:

?

`minibuffer-completion-help`

SPC

`minibuffer-complete-word`

TAB

`minibuffer-complete`

LFD

`minibuffer-complete-and-exit`

RET

`minibuffer-complete-and-exit`

with other characters bound as in `minibuffer-local-map`.

Variable: **minibuffer-completion-table**

The value of this variable is the alist or obarray used for completion in the minibuffer. This is the global

variable that contains what `completing-read` passes to `try-completion`. It is used by all the minibuffer completion functions, such as `minibuffer-complete-word`.

Variable: **minibuffer-completion-predicate**

This variable's value is the predicate that `completing-read` passes to `try-completion`. The variable is also used by the other minibuffer completion functions.

Command: **minibuffer-complete-word**

This function completes the minibuffer contents by at most a single word. Even if the minibuffer contents have only one completion, `minibuffer-complete-word` does not add any characters beyond the first character that is not a word constituent. See section Syntax Tables.

Command: **minibuffer-complete**

This function completes the minibuffer contents as far as possible.

Command: **minibuffer-complete-and-exit**

This function completes the minibuffer contents, and exits if confirmation is not required, i.e., if `minibuffer-completion-confirm` is non-`nil`. If confirmation *is* required, it is given by repeating this command immediately.

Variable: **minibuffer-completion-confirm**

When the value of this variable is non-`nil`, Emacs asks for confirmation of a completion before exiting the minibuffer. The function `minibuffer-complete-and-exit` checks the value of this variable before it exits.

Command: **minibuffer-completion-help**

This function creates a list of the possible completions of the current minibuffer contents. It works by calling `all-completions` using the value of the variable `minibuffer-completion-table` as the collection argument, and the value of `minibuffer-completion-predicate` as the predicate argument. The list of completions is displayed as text in a buffer named `*Completions*'.

Function: **display-completion-list** *completions*

This function displays completions to the stream in `standard-output`, usually a buffer. (See section Reading and Printing Lisp Objects, for more information about streams.) The argument completions is normally a list of completions just returned by `all-completions`, but it does not have to be. Each element may be a symbol or a string, either of which is simply printed, or a list of two strings, which is printed as if the strings were concatenated.

This function is called by `minibuffer-completion-help`. The most common way to use it is together with `with-output-to-temp-buffer`, like this:

```
(with-output-to-temp-buffer " *Completions*"
  (display-completion-list
    (all-completions (buffer-string) my-alist)))
```

User Option: **completion-auto-help**

If this variable is non-`nil`, the completion commands automatically display a list of possible completions whenever nothing can be completed because the next character is not uniquely determined.

High-Level Completion Functions

This section describes the higher-level convenient functions for reading certain sorts of names with completion.

Function: **read-buffer** *prompt &optional default existing*

This function reads the name of a buffer and returns it as a string. The argument default is the default name to use, the value to return if the user exits with an empty minibuffer. If non-`nil`, it should be a string. It is mentioned in the prompt, but is not inserted in the minibuffer as initial input.

If existing is non-`nil`, then the name specified must be that of an existing buffer. The usual commands to exit the minibuffer do not exit if the text is not valid, and RET does completion to attempt to find a valid name. (However, default is not checked for this; it is returned, whatever it is, if the user exits with the minibuffer empty.)

In the following example, the user enters `minibuffer.t', and then types RET. The argument existing is `t`, and the only buffer name starting with the given input is `minibuffer.texi', so that name is the value.

```
(read-buffer "Buffer name? " "foo" t)

;; After evaluating the preceding expression,
;;   the following prompt appears,
;;   with an empty minibuffer:

---------- Buffer: Minibuffer ----------
Buffer name? (default foo) -!-
---------- Buffer: Minibuffer ----------

;; The user types minibuffer.t RET.

     => "minibuffer.texi"
```

Function: **read-command** *prompt*

This function reads the name of a command and returns it as a Lisp symbol. The argument prompt is used as in `read-from-minibuffer`. Recall that a command is anything for which `commandp` returns `t`, and a command name is a symbol for which `commandp` returns `t`. See section Interactive Call.

```
(read-command "Command name? ")
```


```
;; After evaluating the preceding expression,
;;   the following appears in the minibuffer:

---------- Buffer: Minibuffer ----------
Command name?
---------- Buffer: Minibuffer ----------
```

If the user types forward-c RET, then this function returns `forward-char`.

The `read-command` function is a simplified interface to the `completing-read` function. It uses the `commandp` predicate to allow only commands to be entered, and it uses the variable `obarray` so as to be able to complete all extant Lisp symbols:

```
(read-command prompt)
==
(intern (completing-read prompt obarray 'commandp t nil))
```

Function: **read-variable** *prompt*

This function reads the name of a user variable and returns it as a symbol.

```
(read-variable "Variable name? ")

;; After evaluating the preceding expression,
;;   the following prompt appears,
;;   with an empty minibuffer:

---------- Buffer: Minibuffer ----------
Variable name? -!-
---------- Buffer: Minibuffer ----------
```

If the user then types fill-p RET, `read-variable` will return `fill-prefix`.

This function is similar to `read-command`, but uses the predicate `user-variable-p` instead of `commandp`:

```
(read-variable prompt)
==
(intern
 (completing-read prompt obarray 'user-variable-p t nil))
```

Reading File Names

Here is another high-level completion function, designed for reading a file name. It provides special features including automatic insertion of the default directory.

Function: **read-file-name** *prompt &optional directory default existing initial*

This function reads a file name in the minibuffer, prompting with prompt and providing completion. If default is non-`nil`, then the function returns default if the user just types RET.

If existing is non-`nil`, then the name must refer to an existing file; then RET performs completion to make the name valid if possible, and then refuses to exit if it is not valid. If the value of existing is neither `nil` nor `t`, then RET also requires confirmation after completion.

The argument directory specifies the directory to use for completion of relative file names. Usually it is inserted in the minibuffer as initial input as well. It defaults to the current buffer's default directory.

If you specify initial, that is an initial file name to insert in the buffer along with directory. In this case, point goes after directory, before initial. The default for initial is `nil`---don't insert any file name. To see what initial does, try the command C-x C-v.

Here is an example:

```
(read-file-name "The file is ")

;; After evaluating the preceding expression,
;;   the following appears in the minibuffer:

---------- Buffer: Minibuffer ----------
The file is /gp/gnu/elisp/-!-
---------- Buffer: Minibuffer ----------
```

Typing manual TAB results in the following:

```
---------- Buffer: Minibuffer ----------
The file is /gp/gnu/elisp/manual.texi-!-
---------- Buffer: Minibuffer ----------
```

If the user types RET, `read-file-name` returns `"/gp/gnu/elisp/manual.texi"`.

User Option: **insert-default-directory**

This variable is used by `read-file-name`. Its value controls whether `read-file-name` starts by placing the name of the default directory in the minibuffer, plus the initial file name if any. If the value of this variable is `nil`, then `read-file-name` does not place any initial input in the minibuffer. In that case, the default directory is still used for completion of relative file names, but is not displayed.

For example:

```
;; Here the minibuffer starts out containing the default directory.

(let ((insert-default-directory t))
  (read-file-name "The file is "))

---------- Buffer: Minibuffer ----------
```


```
The file is ~lewis/manual/-!-
---------- Buffer: Minibuffer ----------

;; Here the minibuffer is empty and only the prompt
;;   appears on its line.

(let ((insert-default-directory nil))
  (read-file-name "The file is "))

---------- Buffer: Minibuffer ----------
The file is -!-
---------- Buffer: Minibuffer ----------
```

Lisp Symbol Completion

If you type a part of a symbol, and then type M-TAB (`lisp-complete-symbol`), this command attempts to fill in as much more of the symbol name as it can. Not only does this save typing, but it can help you with the name of a symbol that you have partially forgotten.

Command: **lisp-complete-symbol**

This function performs completion on the symbol name preceding point. The name is completed against the symbols in the global variable `obarray`, and characters from the completion are inserted into the buffer, making the name longer. If there is more than one completion, a list of all possible completions is placed in the `*Help*' buffer. The bell rings if there is no possible completion in `obarray`.

If an open parenthesis immediately precedes the name, only symbols with function definitions are considered. (By reducing the number of alternatives, this may succeed in completing more characters.) Otherwise, symbols with either a function definition, a value, or at least one property are considered.

`lisp-complete-symbol` returns `t` if the symbol had an exact, and unique, match; otherwise, it returns `nil`.

In the following example, the user has already inserted `(forwa' into the buffer `` `foo.el' ``. The command `lisp-complete-symbol` then completes the name to `(forward-'.

```
---------- Buffer: foo.el ----------
(forwa-!-
---------- Buffer: foo.el ----------

(lisp-complete-symbol)
     => nil

---------- Buffer: foo.el ----------
(forward--!-
---------- Buffer: foo.el ----------
```

Yes-or-No Queries

This section describes functions used to ask the user a yes-or-no question. The function `y-or-n-p` can be answered with a single character; it is useful for questions where an inadvertent wrong answer will not have serious consequences. `yes-or-no-p` is suitable for more momentous questions, since it requires three or four characters to answer.

Strictly speaking, `yes-or-no-p` uses the minibuffer and `y-or-n-p` does not; but it seems best to describe them together.

Function: **y-or-n-p** *prompt*

This function asks the user a question, expecting input in the echo area. It returns `t` if the user types y, `nil` if the user types n. This function also accepts SPC to mean yes and DEL to mean no. It accepts C-] to mean "quit", like C-g, because the question might look like a minibuffer and for that reason the user might try to use C-] to get out. The answer is a single character, with no RET needed to terminate it. Upper and lower case are equivalent.

"Asking the question" means printing prompt in the echo area, followed by the string `(y or n) '. If the input is not one of the expected answers (y, n, SPC, DEL, or something that quits), the function responds `Please answer y or n.', and repeats the request.

This function does not actually use the minibuffer, since it does not allow editing of the answer. It actually uses the echo area (see section The Echo Area), which uses the same screen space as the minibuffer. The cursor moves to the echo area while the question is being asked.

The meanings of answers, even `y' and `n', are not hardwired. They are controlled by the keymap `query-replace-map`. See section Replacement.

In the following example, the user first types q, which is invalid. At the next prompt the user types n.

```
(y-or-n-p "Do you need a lift? ")

;; After evaluating the preceding expression,
;;   the following prompt appears in the echo area:

---------- Echo area ----------
Do you need a lift? (y or n)
---------- Echo area ----------

;; If the user then types q, the following appears:

---------- Echo area ----------
Please answer y or n.  Do you need a lift? (y or n)
---------- Echo area ----------

;; When the user types a valid answer,
```


```
;;   it is displayed after the question:

---------- Echo area ----------
Do you need a lift? (y or n) y
---------- Echo area ----------
```

Note that we show successive lines of echo area messages here. Only one actually appears on the screen at a time.

Function: **yes-or-no-p** *prompt*

This function asks the user a question, expecting input in minibuffer. It returns `t` if the user enters `yes', `nil` if the user types `no'. The user must type RET to finalize the response. Upper and lower case are equivalent.

`yes-or-no-p` starts by displaying prompt in the echo area, followed by `(yes or no) '. The user must type one of the expected responses; otherwise, the function responds `Please answer yes or no.', waits about two seconds and repeats the request.

`yes-or-no-p` requires more work from the user than `y-or-n-p` and is appropriate for more crucial decisions.

Here is an example:

```
(yes-or-no-p "Do you really want to remove everything? ")

;; After evaluating the preceding expression,
;;   the following prompt appears,
;;   with an empty minibuffer:

---------- Buffer: minibuffer ----------
Do you really want to remove everything? (yes or no)
---------- Buffer: minibuffer ----------
```

If the user first types y RET, which is invalid because this function demands the entire word `yes', it responds by displaying these prompts, with a brief pause between them:

```
---------- Buffer: minibuffer ----------
Please answer yes or no.
Do you really want to remove everything? (yes or no)
---------- Buffer: minibuffer ----------
```

Asking Multiple Y-or-N Queries

Function: **map-y-or-n-p** *prompter actor list &optional help action-alist*

This function, new in Emacs 19, asks the user a series of questions, reading a single-character answer in

the echo area for each one.

The value of list specifies what varies from question to question within the series. It should be either a list of objects or a generator function. If it is a function, it should expect no arguments, and should return either the next object or `nil` meaning there are no more questions.

The argument prompter specifies how to ask each question. If prompter is a string, the question text is computed like this:

```
(format prompter object)
```

where object is the next object to ask about (as obtained from list).

If not a string, prompter should be a function of one argument (the next object to ask about) and should return the question text.

The argument actor says how to act on the answers that the user gives. It should be a function of one argument, and it is called with each object that the user says yes for. Its argument is always an object obtained from list.

If the argument help is given, it should be a list of this form:

```
(singular plural action)
```

where singular is a string containing a singular noun that describes the objects conceptually being acted on, plural is the corresponding plural noun, and action is a transitive verb describing what actor does.

If you don't specify help, the default is `("object" "objects" "act on")`.

Each time a question is asked, the user may enter y, Y, or SPC to act on that object; n, N, or DEL to skip that object; ! to act on all following objects; ESC or q to exit (skip all following objects); . (period) to act on the current object and then exit; or C-h to get help. These are the same answers that `query-replace` accepts. The keymap `query-replace-map` defines their meaning for `map-y-or-n-p` as well as for `query-replace`; see section Replacement.

You can use action-alist to specify additional possible answers and what they mean. It is an alist of elements of the form `(char function help)`, each of which defines one additional answer. In this element, char is a character (the answer); function is a function of one argument (an object from list); help is a string.

When the user responds with char, `map-y-or-n-p` calls function. If it returns non-`nil`, the object is considered "acted upon", and `map-y-or-n-p` advances to the next object in list. If it returns `nil`, the prompt is repeated for the same object.

The return value of `map-y-or-n-p` is the number of objects acted on.

Minibuffer Miscellany

This section describes some basic functions and variables related to minibuffers.

Command: **exit-minibuffer**

This command exits the active minibuffer. It is normally bound to keys in minibuffer local keymaps.

Command: **self-insert-and-exit**

This command exits the active minibuffer after inserting the last character typed on the keyboard (found in `last-command-char`; see section Information from the Command Loop).

Command: **previous-history-element** *n*

This command replaces the minibuffer contents with the value of the nth previous (older) history element.

Command: **next-history-element** *n*

This command replaces the minibuffer contents with the value of the nth more recent history element.

Command: **previous-matching-history-element** *pattern*

This command replaces the minibuffer contents with the value of the previous (older) history element that matches pattern. At the time of printing, we have not made a final decision about how to get the pattern interactively or how to match it against history elements.

Command: **next-matching-history-element** *pattern*

This command replaces the minibuffer contents with the value of the next (newer) history element that matches pattern.

Variable: **minibuffer-help-form**

The current value of this variable is used to rebind `help-form` locally inside the minibuffer (see section Help Functions).

Function: **minibuffer-window** *&optional frame*

This function returns the window that is used for the minibuffer. In Emacs 18, there is one and only one minibuffer window; this window always exists and cannot be deleted. In Emacs 19, each frame can have its own minibuffer, and this function returns the minibuffer window used for frame frame (which defaults to the currently selected frame).

Function: **window-minibuffer-p** *window*

This function returns non-`nil` if window is a minibuffer window.

It is not correct to determine whether a given window is a minibuffer by comparing it with the result of `(minibuffer-window)`, because there can be more than one minibuffer window there is more than one frame.

Variable: **minibuffer-scroll-window**

If the value of this variable is non-`nil`, it should be a window object. When the function `scroll-other-window` is called in the minibuffer, it scrolls this window.

Finally, some functions and variables deal with recursive minibuffers (see section Recursive Editing):

Function: **minibuffer-depth**

This function returns the current depth of activations of the minibuffer, a nonnegative integer. If no minibuffers are active, it returns zero.

User Option: **enable-recursive-minibuffers**

If this variable is non-`nil`, you can invoke commands (such as `find-file`) which use minibuffers even while in the minibuffer window. Such invocation produces a recursive editing level for a new minibuffer. The outer-level minibuffer is invisible while you are editing the inner one.

This variable only affects invoking the minibuffer while the minibuffer window is selected. If you switch windows while in the minibuffer, you can always invoke minibuffer commands while some other window is selected.

If a command name has a property `enable-recursive-minibuffers` which is non-`nil`, then the command can use the minibuffer to read arguments even if it is invoked from the minibuffer. The minibuffer command `next-matching-history-element` (normally bound to M-s in the minibuffer) uses this feature.

Go to the previous, next section.

Go to the previous, next section.

Command Loop

When you run Emacs, it enters the editor command loop almost immediately. This loop reads key sequences, executes their definitions, and displays the results. In this chapter, we describe how these things are done, and the subroutines that allow Lisp programs to do them.

Command Loop Overview

The first thing the command loop must do is read a key sequence, which is a sequence of events that translates into a command. It does this by calling the function `read-key-sequence`. Your Lisp code can also call this function (see section Key Sequence Input). Lisp programs can also do input at a lower level with `read-event` (see section Reading One Event) or discard pending input with `discard-input` (see section Peeking and Discarding).

The key sequence is translated into a command through the currently active keymaps. See section Key Lookup, for information on how this is done. The result should be a keyboard macro or an interactively callable function. If the key is M-x, then it reads the name of another command, which is used instead. This is done by the command `execute-extended-command` (see section Interactive Call).

Once the command is chosen, it must be executed, which includes reading arguments to be given to it. This is done by calling `command-execute` (see section Interactive Call). For commands written in Lisp, the `interactive` specification says how to read the arguments. This may use the prefix argument (see section Prefix Command Arguments) or may read with prompting in the minibuffer (see section Minibuffers). For example, the command `find-file` has an `interactive` specification which says to read a file name using the minibuffer. The command's function body does not use the minibuffer; if you call this command from Lisp code as a function, you must supply the file name string as an ordinary Lisp function argument.

If the command is a string or vector (i.e., a keyboard macro) then `execute-kbd-macro` is used to execute it. You can call this function yourself (see section Keyboard Macros).

If a command runs away, typing C-g terminates its execution immediately. This is called quitting (see section Quitting).

Variable: **pre-command-hook**

The editor command loop runs this normal hook before each command.

Variable: **post-command-hook**

The editor command loop runs this normal hook after each command.

Defining Commands

A Lisp function becomes a command when its body contains, at top level, a form which calls the special form `interactive`. This form does nothing when actually executed, but its presence serves as a flag to indicate that interactive calling is permitted. Its argument controls the reading of arguments for an interactive call.

Using `interactive`

This section describes how to write the `interactive` form that makes a Lisp function an interactively-callable command.

Special Form: **interactive** *arg-descriptor*

This special form declares that the function in which it appears is a command, and that it may therefore be called interactively (via M-x or by entering a key sequence bound to it). The argument arg-descriptor declares the way the arguments to the command are to be computed when the command is called interactively.

A command may be called from Lisp programs like any other function, but then the arguments are supplied by the caller and arg-descriptor has no effect.

The `interactive` form has its effect because the command loop (actually, its subroutine `call-interactively`) scans through the function definition looking for it, before calling the function. Once the function is called, all its body forms including the `interactive` form are executed, but at this time `interactive` simply returns `nil` without even evaluating its argument.

There are three possibilities for the argument arg-descriptor:

- It may be omitted or `nil`; then the command is called with no arguments. This leads quickly to an error if the command requires one or more arguments.
- It may be a Lisp expression that is not a string; then it should be a form that is evaluated to get a list of arguments to pass to the command.
- It may be a string; then its contents should consist of a code character followed by a prompt (which some code characters use and some ignore). The prompt ends either with the end of the string or with a newline. Here is a simple example:

  ```
  (interactive "bFrobnicate buffer: ")
  ```

  The code letter `b' says to read the name of an existing buffer, with completion. The buffer name is the sole argument passed to the command. The rest of the string is a prompt.

  If there is a newline character in the string, it terminates the prompt. If the string does not end there, then the rest of the string should contain another code character and prompt, specifying another argument. You can specify any number of arguments in this way.

  The prompt string can use `%' to include previous argument values in the prompt. This is done

using `format` (see section Formatting Strings). For example, here is how you could read the name of an existing buffer followed by a new name to give to that buffer:

```
(interactive "bBuffer to rename: \nsRename buffer %s to: ")
```

If the first character in the string is `*', then an error is signaled if the buffer is read-only.

If the first character in the string is `@', and if the key sequence used to invoke the command includes any mouse events, then the window associated with the first of those events is selected before the command is run.

You can use `*' and `@' together; the order does not matter. Actual reading of arguments is controlled by the rest of the prompt string (starting with the first character that is not `*' or `@').

Code Characters for `interactive`

The code character descriptions below contain a number of key words, defined here as follows:

Completion

Provide completion. TAB, SPC, and RET perform name completion because the argument is read using `completing-read` (see section Completion). ? displays a list of possible completions.

Existing

Require the name of an existing object. An invalid name is not accepted; the commands to exit the minibuffer do not exit if the current input is not valid.

Default

A default value of some sort is used if the user enters no text in the minibuffer. The default depends on the code character.

No I/O

This code letter computes an argument without reading any input. Therefore, it does not use a prompt string, and any prompt string you supply is ignored.

Prompt

A prompt immediately follows the code character. The prompt ends either with the end of the string or with a newline.

Special

This code character is meaningful only at the beginning of the interactive string, and it does not look for a prompt or a newline. It is a single, isolated character.

Here are the code character descriptions for use with `interactive`:

`*'

Signal an error if the current buffer is read-only. Special.

`@'

Select the window mentioned in the first mouse event in the key sequence that invoked this command. Special.

`a'

A function name (i.e., a symbol which is `fboundp`). Existing, Completion, Prompt.

`b'

The name of an existing buffer. By default, uses the name of the current buffer (see section Buffers). Existing, Completion, Default, Prompt.

`B'

A buffer name. The buffer need not exist. By default, uses the name of a recently used buffer other than the current buffer. Completion, Prompt.

`c'

A character. The cursor does not move into the echo area. Prompt.

`C'

A command name (i.e., a symbol satisfying `commandp`). Existing, Completion, Prompt.

`d'

The position of point as a number (see section Point). No I/O.

`D'

A directory name. The default is the current default directory of the current buffer, `default-directory` (see section Operating System Environment). Existing, Completion, Default, Prompt.

`e'

The first or next mouse event in the key sequence that invoked the command. More precisely, `e' gets events which are lists, so you can look at the data in the lists. See section Input Events. No I/O.

You can use `e' more than once in a single command's interactive specification. If the key sequence which invoked the command has n events with parameters, the nth `e' provides the nth list event. Events which are not lists, such as function keys and ASCII characters, do not count where `e' is concerned.

Even though `e' does not use a prompt string, you must follow it with a newline if it is not the last code character.

`f'

A file name of an existing file (see section File Names). The default directory is `default-directory`. Existing, Completion, Default, Prompt.

`F'

A file name. The file need not exist. Completion, Default, Prompt.

`k'

A key sequence (see section Keymap Terminology). This keeps reading events until a command (or undefined command) is found in the current key maps. The key sequence argument is represented as a string or vector. The cursor does not move into the echo area. Prompt.

This kind of input is used by commands such as `describe-key` and `global-set-key`.

`m'

The position of the mark as a number. No I/O.

`n'

A number read with the minibuffer. If the input is not a number, the user is asked to try again. The prefix argument, if any, is not used. Prompt.

`N'

The raw prefix argument. If the prefix argument is `nil`, then a number is read as with n. Requires a number. Prompt.

`p'

The numeric prefix argument. (Note that this `p' is lower case.) No I/O.

`P'

The raw prefix argument. (Note that this `P' is upper case.) See section Prefix Command Arguments. No I/O.

`r'

Point and the mark, as two numeric arguments, smallest first. This is the only code letter that specifies two successive arguments rather than one. No I/O.

`s'

Arbitrary text, read in the minibuffer and returned as a string (see section Reading Text Strings with the Minibuffer). Terminate the input with either LFD or RET. (C-q may be used to include either of these characters in the input.) Prompt.

`S'

An interned symbol whose name is read in the minibuffer. Any whitespace character terminates the input. (Use C-q to include whitespace in the string.) Other characters that normally terminate a symbol (e.g., parentheses and brackets) do not do so here. Prompt.

`v'

A variable declared to be a user option (i.e., satisfying the predicate `user-variable-p`). See section High-Level Completion Functions. Existing, Completion, Prompt.

`x'

A Lisp object specified in printed representation, terminated with a LFD or RET. The object is not evaluated. See section Reading Lisp Objects with the Minibuffer. Prompt.

`X'

A Lisp form is read as with x, but then evaluated so that its value becomes the argument for the command. Prompt.

Examples of Using `interactive`

Here are some examples of `interactive`:

```
(defun foo1 ()                  ; foo1 takes no arguments,
```


```
    (interactive)           ;   just moves forward two words.
    (forward-word 2))
     => foo1

(defun foo2 (n)             ; foo2 takes one argument,
    (interactive "p")       ;   which is the numeric prefix.
    (forward-word (* 2 n)))
     => foo2

(defun foo3 (n)             ; foo3 takes one argument,
    (interactive "nCount:") ;   which is read with the Minibuffer.
    (forward-word (* 2 n)))
     => foo3

(defun three-b (b1 b2 b3)
  "Select three existing buffers.
Put them into three windows, selecting the last one."
    (interactive "bBuffer1:\nbBuffer2:\nbBuffer3:")
    (delete-other-windows)
    (split-window (selected-window) 8)
    (switch-to-buffer b1)
    (other-window 1)
    (split-window (selected-window) 8)
    (switch-to-buffer b2)
    (other-window 1)
    (switch-to-buffer b3))
     => three-b
(three-b "*scratch*" "declarations.texi" "*mail*")
     => nil
```

Interactive Call

After the command loop has translated a key sequence into a definition, it invokes that definition using the function `command-execute`. If the definition is a function that is a command, `command-execute` calls `call-interactively`, which reads the arguments and calls the command. You can also call these functions yourself.

Function: **commandp** *object*

Returns `t` if object is suitable for calling interactively; that is, if object is a command. Otherwise, returns `nil`.

The interactively callable objects include strings and vectors (treated as keyboard macros), lambda expressions that contain a top-level call to `interactive`, byte-code function objects, autoload objects that are declared as interactive (non-`nil` fourth argument to `autoload`), and some of the primitive functions.

A symbol is `commandp` if its function definition is `commandp`.

Keys and keymaps are not commands. Rather, they are used to look up commands (see section Keymaps).

See `documentation` in section Access to Documentation Strings, for a realistic example of using `commandp`.

Function: **call-interactively** *command &optional record-flag*

This function calls the interactively callable function command, reading arguments according to its interactive calling specifications. An error is signaled if command cannot be called interactively (i.e., it is not a command). Note that keyboard macros (strings and vectors) are not accepted, even though they are considered commands.

If record-flag is non-`nil`, then this command and its arguments are unconditionally added to the list `command-history`. Otherwise, the command is added only if it uses the minibuffer to read an argument. See section Command History.

Function: **command-execute** *command &optional record-flag*

This function executes command as an editing command. The argument command must satisfy the `commandp` predicate; i.e., it must be an interactively callable function or a string.

A string or vector as command is executed with `execute-kbd-macro`. A function is passed to `call-interactively`, along with the optional record-flag.

A symbol is handled by using its function definition in its place. A symbol with an `autoload` definition counts as a command if it was declared to stand for an interactively callable function. Such a definition is handled by loading the specified library and then rechecking the definition of the symbol.

Command: **execute-extended-command** *prefix-argument*

This function reads a command name from the minibuffer using `completing-read` (see section Completion). Then it uses `command-execute` to call the specified command. Whatever that command returns becomes the value of `execute-extended-command`.

If the command asks for a prefix argument, the value prefix-argument is supplied. If `execute-extended-command` is called interactively, the current raw prefix argument is used for prefix-argument, and thus passed on to whatever command is run.

`execute-extended-command` is the normal definition of M-x, so it uses the string `M-x ' as a prompt. (It would be better to take the prompt from the events used to invoke `execute-extended-command`, but that is painful to implement.) A description of the value of the prefix argument, if any, also becomes part of the prompt.

```
(execute-extended-command 1)
---------- Buffer: Minibuffer ----------
M-x forward-word RET
```


```
---------- Buffer: Minibuffer ----------
     => t
```

Function: **interactive-p**

This function returns `t` if the containing function (the one that called `interactive-p`) was called interactively, with the function `call-interactively`. (It makes no difference whether `call-interactively` was called from Lisp or directly from the editor command loop.) Note that if the containing function was called by Lisp evaluation (or with `apply` or `funcall`), then it was not called interactively.

The usual application of `interactive-p` is for deciding whether to print an informative message. As a special exception, `interactive-p` returns `nil` whenever a keyboard macro is being run. This is to suppress the informative messages and speed execution of the macro.

For example:

```
(defun foo ()
  (interactive)
  (and (interactive-p)
       (message "foo")))
     => foo

(defun bar ()
  (interactive)
  (setq foobar (list (foo) (interactive-p))))
     => bar

;; Type M-x foo.
     -| foo

;; Type M-x bar.
;; This does not print anything.

foobar
     => (nil t)
```

Information from the Command Loop

The editor command loop sets several Lisp variables to keep status records for itself and for commands that are run.

Variable: **last-command**

This variable records the name of the previous command executed by the command loop (the one before the current command). Normally the value is a symbol with a function definition, but this is not guaranteed.

The value is set by copying the value of `this-command` when a command returns to the command loop, except when the command specifies a prefix argument for the following command.

Variable: **this-command**

This variable records the name of the command now being executed by the editor command loop. Like `last-command`, it is normally a symbol with a function definition.

This variable is set by the command loop just before the command is run, and its value is copied into `last-command` when the command finishes (unless the command specifies a prefix argument for the following command).

Some commands change the value of this variable during their execution, simply as a flag for whatever command runs next. In particular, the functions that kill text set `this-command` to `kill-region` so that any kill commands immediately following will know to append the killed text to the previous kill.

Function: **this-command-keys**

This function returns a string or vector containing the key sequence that invoked the present command, plus any previous commands that generated the prefix argument for this command. The value is a string if all those events were characters. See section Input Events.

```
(this-command-keys)
;; Now type C-u C-x C-e.
     => "^U^X^E"
```

Variable: **last-nonmenu-event**

This variable holds the last input event read as part of a key sequence, aside from events resulting from mouse menus.

One use of this variable is to figure out a good default location to pop up another menu.

Variable: **last-command-event**

Variable: **last-command-char**

This variable is set to the last input event that was read by the command loop as part of a command. The principal use of this variable is in `self-insert-command`, which uses it to decide which character to insert.

```
last-command-char
;; Now type C-u C-x C-e.
     => 5
```

The value is 5 because that is the ASCII code for C-e.

The alias `last-command-char` exists for compatibility with Emacs version 18.

Variable: **last-event-frame**

This variable records which frame the last input event was directed to. Usually this is the frame that was selected when the event was generated, but if that frame has redirected input focus to another frame, the value is the frame to which the event was redirected. See section Input Focus.

Variable: **echo-keystrokes**

This variable determines how much time should elapse before command characters echo. Its value must be an integer, which specifies the number of seconds to wait before echoing. If the user types a prefix key (say C-x) and then delays this many seconds before continuing, the key C-x is echoed in the echo area. Any subsequent characters in the same command will be echoed as well.

If the value is zero, then command input is not echoed.

Input Events

The Emacs command loop reads a sequence of input events that represent keyboard or mouse activity. The events for keyboard activity are characters or symbols; mouse events are always lists. This section describes the representation and meaning of input events in detail.

A command invoked using events that are lists can get the full values of these events using the `e' interactive code. See section Code Characters for `interactive`.

A key sequence that starts with a mouse event is read using the keymaps of the buffer in the window that the mouse was in, not the current buffer. This does not imply that clicking in a window selects that window or its buffer--that is entirely under the control of the command binding of the key sequence.

Function: **eventp** *object*

This function returns non-`nil` if event is an input event.

Keyboard Events

There are two kinds of input you can get from the keyboard: ordinary keys, and function keys. Ordinary keys correspond to characters; the events they generate are represented in Lisp as characters. In Emacs versions 18 and earlier, characters were the only events.

An input character event consists of a basic code between 0 and 255, plus any or all of these modifier bits:

meta

The 2**23 bit in the character code indicates a character typed with the meta key held down.

control

The 2**22 bit in the character code indicates a non-ASCII control character.

ASCII control characters such as C-a have special basic codes of their own, so Emacs needs no special bit to indicate them. Thus, the code for C-a is just 1.

But if you type a control combination not in ASCII, such as % with the control key, the numeric

value you get is the code for % plus 2**22 (assuming the terminal supports non-ASCII control characters).

shift

The 2**21 bit in the character code indicates an ASCII control character typed with the shift key held down.

For letters, the basic code indicates upper versus lower case; for digits and punctuation, the shift key selects an entirely different character with a different basic code. In order to keep within the ASCII character set whenever possible, Emacs avoids using the 2**21 bit for those characters.

However, ASCII provides no way to distinguish C-A from C-A, so Emacs uses the 2**21 bit in C-A and not in C-a.

hyper

The 2**20 bit in the character code indicates a character typed with the hyper key held down.

super

The 2**19 bit in the character code indicates a character typed with the super key held down.

alt

The 2**18 bit in the character code indicates a character typed with the alt key held down. (On some terminals, the key labeled ALT is actually the meta key.)

In the future, Emacs may support a larger range of basic codes. We may also move the modifier bits to larger bit numbers. Therefore, you should avoid mentioning specific bit numbers in your program. Instead, the way to test the modifier bits of a character is with the function `event-modifiers` (see section Classifying Events).

Function Keys

Most keyboards also have function keys---keys which have names or symbols that are not characters. Function keys are represented in Lisp as symbols; the symbol's name is the function key's label. For example, pressing a key labeled F1 places the symbol `f1` in the input stream.

For all keyboard events, the event type (which classifies the event for key lookup purposes) is identical to the event--it is the character or the symbol. See section Classifying Events.

Here are a few special cases in the symbol naming convention for function keys:

`backspace`, `tab`, `newline`, `return`, `delete`

These keys correspond to common ASCII control characters that have special keys on most keyboards.

In ASCII, C-i and TAB are the same character. Emacs lets you distinguish them if you wish, by returning the former as the integer 9, and the latter as the symbol `tab`.

Most of the time, it's not useful to distinguish the two. So normally `function-key-map` is set up to map `tab` into 9. Thus, a key binding for character code 9 also applies to `tab`. Likewise for the other symbols in this group. The function `read-char` also converts these events into

characters.

In ASCII, BS is really C-h. But `backspace` converts into the character code 127 (DEL), not into code 8 (BS). This is what most users prefer.

`kp-add`, `kp-decimal`, `kp-divide`, ...

Keypad keys (to the right of the regular keyboard).

`kp-0`, `kp-1`, ...

Keypad keys with digits.

`kp-f1`, `kp-f2`, `kp-f3`, `kp-f4`

Keypad PF keys.

`left`, `up`, `right`, `down`

Cursor arrow keys

You can use the modifier keys CTRL, META, HYPER, SUPER, ALT and SHIFT with function keys. The way to represent them is with prefixes in the symbol name:

`A-'

The alt modifier.

`C-'

The control modifier.

`H-'

The hyper modifier.

`M-'

The meta modifier.

`S-'

The shift modifier.

`s-'

The super modifier.

Thus, the symbol for the key F3 with META held down is M-F3. When you use more than one prefix, we recommend you write them in alphabetical order (though the order does not matter in arguments to the key-binding lookup and modification functions).

Click Events

When the user presses a mouse button and releases it at the same location, that generates a click event. Mouse click events have this form:

```
(event-type
 (window buffer-pos
  (column . row) timestamp))
```

Here is what the elements normally mean:

event-type

This is a symbol that indicates which mouse button was used. It is one of the symbols `mouse-1`, `mouse-2`, ..., where the buttons are numbered numbered left to right.

You can also use prefixes `A-', `C-', `H-', `M-', `S-' and `s-' for modifiers alt, control, hyper, meta, shift and super, just as you would with function keys.

This symbol also serves as the event type of the event. Key bindings describe events by their types; thus, if there is a key binding for `mouse-1`, that binding would apply to all events whose event-type is `mouse-1`.

window

This is the window in which the click occurred.

column

row

These are the column and row of the click, relative to the top left corner of window, which is `(0 . 0)`.

buffer-pos

This is the buffer position of the character clicked on.

timestamp

This is the time at which the event occurred, in milliseconds. (Since this value wraps around the entire range of Emacs Lisp integers in about five hours, it is useful only for relating the times of nearby events.)

The meanings of buffer-pos, row and column are somewhat different when the event location is in a special part of the screen, such as the mode line or a scroll bar.

If the location is in a scroll bar, then buffer-pos is the symbol `vertical-scroll-bar` or `horizontal-scroll-bar`, and the pair `(column . row)` is replaced with a pair `(portion . whole)`, where portion is the distance of the click from the top or left end of the scroll bar, and whole is the length of the entire scroll bar.

If the position is on a mode line or the vertical line separating window from its neighbor to the right, then buffer-pos is the symbol `mode-line` or `vertical-line`. For the mode line, row does not have meaningful data. For the vertical line, column does not have meaningful data.

Drag Events

With Emacs, you can have a drag event without even changing your clothes. A drag event happens every time the user presses a mouse button and then moves the mouse to a different character position before releasing the button. Like all mouse events, drag events are represented in Lisp as lists. The lists record both the starting mouse position and the final position, like this:

```
(event-type
 (window1 buffer-pos1
```


```
 (column1 . row1) timestamp1)
(window2 buffer-pos2
 (column2 . row2) timestamp2))
```

For a drag event, the name of the symbol event-type contains the prefix `drag-'. The second and third elements of the event give the starting and ending position of the drag. Aside from that, the data have the same meanings as in a click event (see section Click Events). You can access the second element of any mouse event in the same way, with no need to distinguish drag events from others.

The `drag-' prefix follows the modifier key prefixes such as `C-' and `M-'.

If `read-key-sequence` receives a drag event which has no key binding, and the corresponding click event does have a binding, it changes the drag event into a click event at the drag's starting position. This means that you don't have to distinguish between click and drag events unless you want to.

Button-Down Events

Click and drag events happen when the user releases a mouse button. They cannot happen earlier, because there is no way to distinguish a click from a drag until the button is released.

If you want to take action as soon as a button is pressed, you need to handle button-down events.(2). These occur as soon as a button is pressed. They are represented by lists which look exactly like click events (see section Click Events), except that the name of event-type contains the prefix `down-'. The `down-' prefix follows the modifier key prefixes such as `C-' and `M-'.

The function `read-key-sequence`, and the Emacs command loop, ignore any button-down events that don't have command bindings. This means that you need not worry about defining button-down events unless you want them to do something. The usual reason to define a button-down event is so that you can track mouse motion (by reading motion events) until the button is released.

Motion Events

Emacs sometimes generates mouse motion events to describe motion of the mouse without any button activity. Mouse motion events are represented by lists that look like this:

```
(mouse-movement
 (window buffer-pos
  (column . row) timestamp))
```

The second element of the list describes the current position of the mouse, just as in a click event (see section Click Events).

The special form `track-mouse` enables generation of motion events within its body. Outside of `track-mouse` forms, Emacs does not generate events for mere motion of the mouse, and these events do not appear.

Special Form: **track-mouse** *body...*

This special form executes body, with generation of mouse motion events enabled. Typically body would use `read-event` to read the motion events and modify the display accordingly.

When the user releases the button, that generates a click event. Normally body should return when it sees the click event, and discard the event.

Focus Events

Window systems provide general ways for the user to control which window gets keyboard input. This choice of window is called the focus. When the user does something to switch between Emacs frames, that generates a focus event. The normal definition of a focus event, in the global keymap, is to select a new frame within Emacs, as the user would expect. See section Input Focus.

Focus events are represented in Lisp as lists that look like this:

```
(switch-frame new-frame)
```

where new-frame is the frame switched to.

In X windows, most window managers are set up so that just moving the mouse into a window is enough to set the focus there. Emacs appears to do this, because it changes the cursor to solid in the new frame. However, there is no need for the Lisp program to know about the focus change until some other kind of input arrives. So Emacs generates the focus event only when the user actually types a keyboard key or presses a mouse button in the new frame; just moving the mouse between frames does not generate a focus event.

A focus event in the middle of a key sequence would garble the sequence. So Emacs never generates a focus event in the middle of a key sequence. If the user changes focus in the middle of a key sequence--that is, after a prefix key--then Emacs reorders the events so that the focus event comes either before or after the multi-event key sequence, and not within it.

Event Examples

If the user presses and releases the left mouse button over the same location, that generates a sequence of events like this:

```
(down-mouse-1 (#<window 18 on NEWS> 2613 (0 . 38) -864320))
(mouse-1      (#<window 18 on NEWS> 2613 (0 . 38) -864180))
```

Or, while holding the control key down, the user might hold down the second mouse button, and drag the mouse from one line to the next. That produces two events, as shown here:

```
(C-down-mouse-2 (#<window 18 on NEWS> 3440 (0 . 27) -731219))
(C-drag-mouse-2 (#<window 18 on NEWS> 3440 (0 . 27) -731219)
                (#<window 18 on NEWS> 3510 (0 . 28) -729648))
```

Or, while holding down the meta and shift keys, the user might press the second mouse button on the

window's mode line, and then drag the mouse into another window. That produces the following pair of events:

```
(M-S-down-mouse-2 (#<window 18 on NEWS> mode-line (33 . 31) -457844))
(M-S-drag-mouse-2 (#<window 18 on NEWS> mode-line (33 . 31) -457844)
                  (#<window 20 on carlton-sanskrit.tex> 161 (33 . 3)
                   -453816))
```

Classifying Events

Every event has an event type which classifies the event for key binding purposes. For a keyboard event, the event type equals the event value; thus, the event type for a character is the character, and the event type for a function key symbol is the symbol itself. For events which are lists, the event type is the symbol in the CAR of the list. Thus, the event type is always a symbol or a character.

Two events of the same type are equivalent where key bindings are concerned; thus, they always run the same command. That does not necessarily mean they do the same things, however, as some commands look at the whole event to decide what to do. For example, some commands use the location of a mouse event to decide what text to act on.

Sometimes broader classifications of events are useful. For example, you might want to ask whether an event involved the META key, regardless of which other key or mouse button was used.

To get such information conveniently, call the functions `event-modifiers` and `event-basic-type`.

Function: **event-modifiers** *event*

This function returns a list of the modifiers that event has. The modifiers are symbols; they include `shift`, `control`, `meta`, `alt`, `hyper` and `super`. In addition, the property of a mouse event symbol always has one of `click`, `drag`, and `down` among the modifiers. For example:

```
(event-modifiers ?a)
     => nil
(event-modifiers ?\C-a)
     => (control)
(event-modifiers ?\C-%)
     => (control)
(event-modifiers ?\C-\S-a)
     => (control shift)
(event-modifiers 'f5)
     => nil
(event-modifiers 's-f5)
     => (super)
(event-modifiers 'M-S-f5)
     => (meta shift)
(event-modifiers 'mouse-1)
```


```
     => (click)
(event-modifiers 'down-mouse-1)
     => (down)
```

The modifiers list for a click event explicitly contains `click`, but the event symbol name itself does not contain `click'.

Function: **event-basic-type** *event*

This function returns the key or mouse button that event describes, with all modifiers removed. For example:

```
(event-basic-type ?a)
     => 97
(event-basic-type ?A)
     => 97
(event-basic-type ?\C-a)
     => 97
(event-basic-type ?\C-\S-a)
     => 97
(event-basic-type 'f5)
     => f5
(event-basic-type 's-f5)
     => f5
(event-basic-type 'M-S-f5)
     => f5
(event-basic-type 'down-mouse-1)
     => mouse-1
```

Function: **mouse-movement-p** *object*

This function returns non-`nil` if object is a mouse movement event.

Accessing Events

This section describes convenient functions for accessing the data in an event which is a list.

The following functions return the starting or ending position of a mouse-button event. The position is a list of this form:

```
(window buffer-position (col . row) timestamp)
```

Function: **event-start** *event*

This returns the starting position of event.

If event is a click or button-down event, this returns the location of the event. If event is a drag event, this returns the drag's starting position.

Function: **event-end** *event*

This returns the ending position of event.

If event is a drag event, this returns the position where the user released the mouse button. If event is a click or button-down event, the value is actually the starting position, which is the only position such events have.

These four functions take a position-list as described above, and return various parts of it.

Function: **posn-window** *position*

Return the window that position is in.

Function: **posn-point** *position*

Return the buffer location in position.

Function: **posn-col-row** *position*

Return the row and column in position, as a list `(col . row)`.

Function: **posn-timestamp** *position*

Return the timestamp of position.

Function: **scroll-bar-scale** *ratio total*

This function multiples (in effect) ratio by total, rounding the result to an integer. ratio is not a number, but rather a pair `(num . denom)`.

This is handy for scaling a position on a scroll bar into a buffer position. Here's how to do that:

```
(scroll-bar-scale (posn-col-row (event-start event))
                  (buffer-size))
```

Putting Keyboard Events in Strings

In most of the places where strings are used, we conceptualize the string as containing text characters--the same kind of characters found in buffers or files. Occasionally Lisp programs use strings which conceptually contain keyboard characters; for example, they may be key sequences or keyboard macro definitions. There are special rules for how to put keyboard characters into a string, because they are not limited to the range of 0 to 255 as text characters are.

A keyboard character typed using the META key is called a meta character. The numeric code for such an event includes the 2**23 bit; it does not even come close to fitting in a string. However, earlier Emacs versions used a different representation for these characters, which gave them codes in the range of 128 to 255. That did fit in a string, and many Lisp programs contain string constants that use `\M-' to express meta characters, especially as the argument to `define-key` and similar functions.

We provide backward compatibility to run those programs with special rules for how to put a keyboard

character event in a string. Here are the rules:

- If the keyboard event value is in the range of 0 to 127, it can go in the string unchanged.
- The meta variants of those events, with codes in the range of 2**23 to 2**23+127, can also go in the string, but you must change their numeric values. You must set the 2**7 bit instead of the 2**23 bit, resulting in a value between 128 and 255.
- Other keyboard character events cannot fit in a string. This includes keyboard events in the range of 128 to 255.

Functions such as `read-key-sequence` that can construct strings containing events follow these rules.

When you use the read syntax `\M-' in a string, it produces a code in the range of 128 to 255--the same code that you get if you modify the corresponding keyboard event to put it in the string. Thus, meta events in strings work consistently regardless of how they get into the strings.

New programs can avoid dealing with these rules by using vectors instead of strings for key sequences when there is any possibility that these issues might arise.

The reason we changed the representation of meta characters as keyboard events is to make room for basic character codes beyond 127, and support meta variants of such larger character codes.

Reading Input

The editor command loop reads keyboard input using the function `read-key-sequence`, which uses `read-event`. These and other functions for keyboard input are also available for use in Lisp programs. See also `momentary-string-display` in section Temporary Displays, and `sit-for` in section Waiting for Elapsed Time or Input. See section Terminal Input, for functions and variables for controlling terminal input modes and debugging terminal input.

For higher-level input facilities, see section Minibuffers.

Key Sequence Input

The command loop reads input a key sequence at a time, by calling `read-key-sequence`. Lisp programs can also call this function; for example, `describe-key` uses it to read the key to describe.

Function: **read-key-sequence** *prompt*

This function reads a key sequence and returns it as a string or vector. It keeps reading events until it has accumulated a full key sequence; that is, enough to specify a non-prefix command using the currently active keymaps.

If the events are all characters and all can fit in a string, then `read-key-sequence` returns a string (see section Putting Keyboard Events in Strings). Otherwise, it returns a vector, since a vector can hold all kinds of events--characters, symbols, and lists. The elements of the string or vector are the events in the key sequence.

Quitting is suppressed inside `read-key-sequence`. In other words, a C-g typed while reading with this function is treated like any other character, and does not set `quit-flag`. See section Quitting.

The argument prompt is either a string to be displayed in the echo area as a prompt, or `nil`, meaning not to display a prompt.

In the example below, the prompt `?' is displayed in the echo area, and the user types C-x C-f.

```
(read-key-sequence "?")

---------- Echo Area ----------
?C-x C-f
---------- Echo Area ----------

     => "^X^F"
```

Variable: **num-input-keys**

This variable's value is the number of key sequences processed so far in this Emacs session. This includes key sequences read from the terminal and key sequences read from keyboard macros being executed.

If an input character is an upper case letter and has no key binding, but the lower case equivalent has one, then `read-key-sequence` converts the character to lower case. Note that `lookup-key` does not perform case conversion in this way.

The function `read-key-sequence` also transforms some mouse events. It converts unbound drag events into click events, and discards unbound button-down events entirely. It also reshuffles focus events so that they never appear in a key sequence with any other events.

When mouse events occur in special parts of a window, such as a mode line or a scroll bar, the event itself shows nothing special--only the symbol that would normally represent that mouse button and modifier keys. The information about the screen region is kept elsewhere in the event--in the coordinates. But `read-key-sequence` translates this information into imaginary prefix keys, all of which are symbols: `mode-line`, `vertical-line`, `horizontal-scroll-bar` and `vertical-scroll-bar`.

For example, if you call `read-key-sequence` and then click the mouse on the window's mode line, this is what happens:

```
(read-key-sequence "Click on the mode line: ")
     => [mode-line
         (mouse-1
          (#<window 6 on NEWS> mode-line
           (40 . 63) 5959987))]
```

You can define meanings for mouse clicks in special window regions by defining key sequences using these imaginary prefix keys.

Reading One Event

The lowest level functions for command input are those which read a single event.

Function: **read-event**

This function reads and returns the next event of command input, waiting if necessary until an event is available. Events can come directly from the user or from a keyboard macro.

The function `read-event` does not display any message to indicate it is waiting for input; use `message` first, if you wish to display one. If you have not displayed a message, `read-event` does prompting: it displays descriptions of the events that led to or were read by the current command. See section The Echo Area.

If `cursor-in-echo-area` is non-`nil`, then `read-event` moves the cursor temporarily to the echo area, to the end of any message displayed there. Otherwise `read-event` does not move the cursor.

Here is what happens if you call `read-event` and then press the right-arrow function key:

```
(read-event)
     => right
```

Function: **read-char**

This function reads and returns a character of command input. It discards any events that are not characters until it gets a character.

In the first example, the user types 1 (which is ASCII code 49). The second example shows a keyboard macro definition that calls `read-char` from the minibuffer. `read-char` reads the keyboard macro's very next character, which is 1. The value of this function is displayed in the echo area by the command `eval-expression`.

```
(read-char)
     => 49

(symbol-function 'foo)
     => "^[^[(read-char)^M1"
(execute-kbd-macro foo)
     -| 49
     => nil
```

Quoted Character Input

You can use the function `read-quoted-char` when you want the user to specify a character, and allow the user to specify a control or meta character conveniently with quoting or as an octal character code. The command `quoted-insert` calls this function.

Function: **read-quoted-char** *&optional prompt*

This function is like `read-char`, except that if the first character read is an octal digit (0-7), it reads to two more octal digits (but stopping if a non-octal digit is found) and returns the character represented by those digits as an octal number.

Quitting is suppressed when the first character is read, so that the user can enter a C-g. See section Quitting.

If prompt is supplied, it specifies a string for prompting the user. The prompt string is always printed in the echo area and followed by a single `-'.

In the following example, the user types in the octal number 177 (which is 127 in decimal).

```
(read-quoted-char "What character")

---------- Echo Area ----------
What character-177
---------- Echo Area ----------

     => 127
```

Peeking and Discarding

Variable: **unread-command-events**

This variable holds a list of events waiting to be read as command input. The events are used in the order they appear in the list.

The variable is used because in some cases a function reads a event and then decides not to use it. Storing the event in this variable causes it to be processed normally by the command loop or when the functions to read command input are called.

For example, the function that implements numeric prefix arguments reads any number of digits. When it finds a non-digit event, it must unread the event so that it can be read normally by the command loop. Likewise, incremental search uses this feature to unread events it does not recognize.

Variable: **unread-command-char**

This variable holds a character to be read as command input. A value of -1 means "empty".

This variable is pretty much obsolete now that you can use `unread-command-events` instead; it exists only to support programs written for Emacs versions 18 and earlier.

Function: **listify-key-sequence** *key*

This function converts the string or vector key to a list of events which you can put in `unread-command-events`. Converting a vector is simple, but converting a string is tricky because of the special representation used for meta characters in a string (see section Putting Keyboard Events in Strings).

Function: **input-pending-p**

This function determines whether any command input is currently available to be read. It returns immediately, with value `t` if there is input, `nil` otherwise. On rare occasions it may return `t` when no input is available.

Variable: **last-input-event**

Variable: **last-input-char**

This variable records the last terminal input event read, whether as part of a command or explicitly by a Lisp program.

In the example below, a character is read (the character 1, ASCII code 49). It becomes the value of `last-input-char`, while C-e (from the C-x C-e command used to evaluate this expression) remains the value of `last-command-char`.

```
(progn (print (read-char))
       (print last-command-char)
       last-input-char)
     -| 49
     -| 5
     => 49
```

The alias `last-input-char` exists for compatibility with Emacs version 18.

Function: **discard-input**

This function discards the contents of the terminal input buffer and cancels any keyboard macro that might be in the process of definition. It returns `nil`.

In the following example, the user may type a number of characters right after starting the evaluation of the form. After the `sleep-for` finishes sleeping, any characters that have been typed are discarded.

```
(progn (sleep-for 2)
  (discard-input))
     => nil
```

Waiting for Elapsed Time or Input

The waiting commands are designed to make Emacs wait for a certain amount of time to pass or until there is input. For example, you may wish to pause in the middle of a computation to allow the user time to view the display. `sit-for` pauses and updates the screen, and returns immediately if input comes in, while `sleep-for` pauses without updating the screen.

Function: **sit-for** *seconds &optional millisec nodisp*

This function performs redisplay (provided there is no pending input from the user), then waits seconds

seconds, or until input is available. The result is `t` if `sit-for` waited the full time with no input arriving (see `input-pending-p` in section Peeking and Discarding). Otherwise, the value is `nil`.

The optional argument millisec specifies an additional waiting period measured in milliseconds. This adds to the period specified by seconds. Not all operating systems support waiting periods other than multiples of a second; on those that do not, you get an error if you specify nonzero millisec.

Redisplay is always preempted if input arrives, and does not happen at all if input is available before it starts. Thus, there is no way to force screen updating if there is pending input; however, if there is no input pending, you can force an update with no delay by using `(sit-for 0)`.

If nodisp is non-`nil`, then `sit-for` does not redisplay, but it still returns as soon as input is available (or when the timeout elapses).

The usual purpose of `sit-for` is to give the user time to read text that you display.

Function: **sleep-for** *seconds &optional millisec*

This function simply pauses for seconds seconds without updating the display. It pays no attention to available input. It returns `nil`.

The optional argument millisec specifies an additional waiting period measured in milliseconds. This adds to the period specified by seconds. Not all operating systems support waiting periods other than multiples of a second; on those that do not, you get an error if you specify nonzero millisec.

Use `sleep-for` when you wish to guarantee a delay.

See section Time of Day, for functions to get the current time.

Quitting

Typing C-g while the command loop has run a Lisp function causes Emacs to quit whatever it is doing. This means that control returns to the innermost active command loop.

Typing C-g while the command loop is waiting for keyboard input does not cause a quit; it acts as an ordinary input character. In the simplest case, you cannot tell the difference, because C-g normally runs the command `keyboard-quit`, whose effect is to quit. However, when C-g follows a prefix key, the result is an undefined key. The effect is to cancel the prefix key as well as any prefix argument.

In the minibuffer, C-g has a different definition: it aborts out of the minibuffer. This means, in effect, that it exits the minibuffer and then quits. (Simply quitting would return to the command loop *within* the minibuffer.) The reason why C-g does not quit directly when the command reader is reading input is so that its meaning can be redefined in the minibuffer in this way. C-g following a prefix key is not redefined in the minibuffer, and it has its normal effect of canceling the prefix key and prefix argument. This too would not be possible if C-g quit directly.

C-g causes a quit by setting the variable `quit-flag` to a non-`nil` value. Emacs checks this variable at appropriate times and quits if it is not `nil`. Setting `quit-flag` non-`nil` in any way thus causes a quit.

At the level of C code, quits cannot happen just anywhere; only at the special places which check `quit-flag`. The reason for this is that quitting at other places might leave an inconsistency in Ema internal state. Because quitting is delayed until a safe place, quitting cannot make Emacs crash.

Certain functions such as `read-key-sequence` or `read-quoted-char` prevent quitting entirely even though they wait for input. Instead of quitting, C-g serves as the requested input. In the case of `read-key-sequence`, this serves to bring about the special behavior of C-g in the command loop. In the case of `read-quoted-char`, this is so that C-q can be used to quote a C-g.

You can prevent quitting for a portion of a Lisp function by binding the variable `inhibit-quit` to a non-`nil` value. Then, although C-g still sets `quit-flag` to `t` as usual, the usual result of this--a quit--is prevented. Eventually, `inhibit-quit` will become `nil` again, such as when its binding is unwound at the end of a `let` form. At that time, if `quit-flag` is still non-`nil`, the requested quit happens immediately. This behavior is ideal for a "critical section", where you wish to make sure that quitting does not happen within that part of the program.

In some functions (such as `read-quoted-char`), C-g is handled in a special way which does not involve quitting. This is done by reading the input with `inhibit-quit` bound to `t` and setting `quit-flag` to `nil` before `inhibit-quit` becomes `nil` again. This excerpt from the definition of `read-quoted-char` shows how this is done; it also shows that normal quitting is permitted after the first character of input.

```
(defun read-quoted-char (&optional prompt)
  "...documentation..."
  (let ((count 0) (code 0) char)
    (while (< count 3)
      (let ((inhibit-quit (zerop count))
            (help-form nil))
        (and prompt (message "%s-" prompt))
        (setq char (read-char))
        (if inhibit-quit (setq quit-flag nil)))
      ...)
    (logand 255 code)))
```

Variable: **quit-flag**

If this variable is non-`nil`, then Emacs quits immediately, unless `inhibit-quit` is non-`nil`. Typing C-g sets `quit-flag` non-`nil`, regardless of `inhibit-quit`.

Variable: **inhibit-quit**

This variable determines whether Emacs should quit when `quit-flag` is set to a value other than `nil`. If `inhibit-quit` is non-`nil`, then `quit-flag` has no special effect.

Command: **keyboard-quit**

This function signals the `quit` condition with `(signal 'quit nil)`. This is the same thing that quitting does. (See `signal` in section Errors.)

You can specify a character other than C-g to use for quitting. See the function `set-input-mode` section Terminal Input.

Prefix Command Arguments

Most Emacs commands can use a prefix argument, a number specified before the command itself. (Don't confuse prefix arguments with prefix keys.) The prefix argument is represented by a value that is always available (though it may be `nil`, meaning there is no prefix argument). Each command may use the prefix argument or ignore it.

There are two representations of the prefix argument: raw and numeric. The editor command loop uses the raw representation internally, and so do the Lisp variables that store the information, but commands can request either representation.

Here are the possible values of a raw prefix argument:

- `nil`, meaning there is no prefix argument. Its numeric value is 1, but numerous commands make a distinction between `nil` and the integer 1.
- An integer, which stands for itself.
- A list of one element, which is an integer. This form of prefix argument results from one or a succession of C-u's with no digits. The numeric value is the integer in the list, but some commands make a distinction between such a list and an integer alone.
- The symbol `-`. This indicates that M-- or C-u - was typed, without following digits. The equivalent numeric value is -1, but some commands make a distinction between the integer -1 and the symbol `-`.

The various possibilities may be illustrated by calling the following function with various prefixes:

```
(defun display-prefix (arg)
  "Display the value of the raw prefix arg."
  (interactive "P")
  (message "%s" arg))
```

Here are the results of calling `print-prefix` with various raw prefix arguments:

```
        M-x print-prefix  -| nil

C-u     M-x print-prefix  -| (4)

C-u C-u M-x print-prefix  -| (16)

C-u 3   M-x print-prefix  -| 3

M-3     M-x print-prefix  -| 3      ; (Same as C-u 3.)

C-u -   M-x print-prefix  -| -
```


```
M- -    M-x print-prefix  -| -      ; (Same as C-u -.)

C-u -7  M-x print-prefix  -| -7

M- -7   M-x print-prefix  -| -7     ; (Same as C-u -7.)
```

Emacs uses two variables to store the prefix argument: `prefix-arg` and `current-prefix-arg`. Commands such as `universal-argument` that set up prefix arguments for other commands store them in `prefix-arg`. In contrast, `current-prefix-arg` conveys the prefix argument to the current command, so setting it has no effect on the prefix arguments for future commands.

Normally, commands specify which representation to use for the prefix argument, either numeric or raw, in the `interactive` declaration. (See section Interactive Call.) Alternatively, functions may look at the value of the prefix argument directly in the variable `current-prefix-arg`, but this is less clean.

Do not call the functions `universal-argument`, `digit-argument`, or `negative-argument` unless you intend to let the user enter the prefix argument for the *next* command.

Command: **universal-argument**

This command reads input and specifies a prefix argument for the following command. Don't call this command yourself unless you know what you are doing.

Command: **digit-argument** *arg*

This command adds to the prefix argument for the following command. The argument arg is the raw prefix argument as it was before this command; it is used to compute the updated prefix argument. Don't call this command yourself unless you know what you are doing.

Command: **negative-argument** *arg*

This command adds to the numeric argument for the next command. The argument arg is the raw prefix argument as it was before this command; its value is negated to form the new prefix argument. Don't call this command yourself unless you know what you are doing.

Function: **prefix-numeric-value** *arg*

This function returns the numeric meaning of a valid raw prefix argument value, arg. The argument may be a symbol, a number, or a list. If it is `nil`, the value 1 is returned; if it is any other symbol, the value -1 is returned. If it is a number, that number is returned; if it is a list, the CAR of that list (which should be a number) is returned.

Variable: **current-prefix-arg**

This variable is the value of the raw prefix argument for the *current* command. Commands may examine it directly, but the usual way to access it is with `(interactive "P")`.

Variable: **prefix-arg**

The value of this variable is the raw prefix argument for the *next* editing command. Commands that

specify prefix arguments for the following command work by setting this variable.

Recursive Editing

The Emacs command loop is entered automatically when Emacs starts up. This top-level invocation of the command loop is never exited until the Emacs is killed. Lisp programs can also invoke the command loop. Since this makes more than one activation of the command loop, we call it recursive editing. A recursive editing level has the effect of suspending whatever command invoked it and permitting the user to do arbitrary editing before resuming that command.

The commands available during recursive editing are the same ones available in the top-level editing loop and defined in the keymaps. Only a few special commands exit the recursive editing level; the others return to the recursive editing level when finished. (The special commands for exiting are always available, but do nothing when recursive editing is not in progress.)

All command loops, including recursive ones, set up all-purpose error handlers so that an error in a command run from the command loop will not exit the loop.

Minibuffer input is a special kind of recursive editing. It has a few special wrinkles, such as enabling display of the minibuffer and the minibuffer window, but fewer than you might suppose. Certain keys behave differently in the minibuffer, but that is only because of the minibuffer's local map; if you switch windows, you get the usual Emacs commands.

To invoke a recursive editing level, call the function `recursive-edit`. This function contains the command loop; it also contains a call to `catch` with tag `exit`, which makes it possible to exit the recursive editing level by throwing to `exit` (see section Explicit Nonlocal Exits: `catch` and `throw`). If you throw a value other than `t`, then `recursive-edit` returns normally to the function that called it. The command C-M-c (`exit-recursive-edit`) does this. Throwing a `t` value causes `recursive-edit` to quit, so that control returns to the command loop one level up. This is called aborting, and is done by C-] (`abort-recursive-edit`).

Most applications should not use recursive editing, except as part of using the minibuffer. Usually it is more convenient for the user if you change the major mode of the current buffer temporarily to a special major mode, which has a command to go back to the previous mode. (This technique is used by the w command in Rmail.) Or, if you wish to give the user different text to edit "recursively", create and select a new buffer in a special mode. In this mode, define a command to complete the processing and go back to the previous buffer. (The m command in Rmail does this.)

Recursive edits are useful in debugging. You can insert a call to `debug` into a function definition as a sort of breakpoint, so that you can look around when the function gets there. `debug` invokes a recursive edit but also provides the other features of the debugger.

Recursive editing levels are also used when you type C-r in `query-replace` or use C-x q (`kbd-macro-query`).

Function: **recursive-edit**

This function invokes the editor command loop. It is called automatically by the initialization of Emacs,

to let the user begin editing. When called from a Lisp program, it enters a recursive editing level.

In the following example, the function `simple-rec` first advances point one word, then enters a recursive edit, printing out a message in the echo area. The user can then do any editing desired, and then type C-M-c to exit and continue executing `simple-rec`.

```
(defun simple-rec ()
  (forward-word 1)
  (message "Recursive edit in progress.")
  (recursive-edit)
  (forward-word 1))
     => simple-rec
(simple-rec)
     => nil
```

Command: **exit-recursive-edit**

This function exits from the innermost recursive edit (including minibuffer input). Its definition is effectively `(throw 'exit nil)`.

Command: **abort-recursive-edit**

This function aborts the command that requested the innermost recursive edit (including minibuffer input), by signaling `quit` after exiting the recursive edit. Its definition is effectively `(throw 'exit t)`. See section Quitting.

Command: **top-level**

This function exits all recursive editing levels; it does not return a value, as it jumps completely out of any computation directly back to the main command loop.

Function: **recursion-depth**

This function returns the current depth of recursive edits. When no recursive edit is active, it returns 0.

Disabling Commands

Disabling a command marks the command as requiring user confirmation before it can be executed. Disabling is used for commands which might be confusing to beginning users, to prevent them from using the commands by accident.

The low-level mechanism for disabling a command is to put a non-`nil` `disabled` property on the Lisp symbol for the command. These properties are normally set up by the user's `` `.emacs' `` file with Lisp expressions such as this:

```
(put 'upcase-region 'disabled t)
```

For a few commands, these properties are present by default and may be removed by the `` `.emacs' ``

file.

If the value of the `disabled` property is a string, that string is included in the message printed when the command is used:

```
(put 'delete-region 'disabled
     "Text deleted this way cannot be yanked back!\n")
```

See section 'Disabling' in The GNU Emacs Manual, for the details on what happens when a disabled command is invoked interactively. Disabling a command has no effect on calling it as a function from Lisp programs.

Command: **enable-command** *command*

Allow command to be executed without special confirmation from now on. The user's `` `.emacs' `` file is optionally altered so that this will apply to future sessions.

Command: **disable-command** *command*

Require special confirmation to execute command from now on. The user's `` `.emacs' `` file is optionally altered so that this will apply to future sessions.

Variable: **disabled-command-hook**

This variable is a normal hook that is run instead of a disabled command, when the user runs the disabled command interactively. The hook functions can use `this-command-keys` to determine what the user typed to run the command, and thus find the command itself.

By default, `disabled-command-hook` contains a function that asks the user whether to proceed.

Command History

The command loop keeps a history of the complex commands that have been executed, to make it convenient to repeat these commands. A complex command is one for which the interactive argument reading uses the minibuffer. This includes any M-x command, any M-ESC command, and any command whose `interactive` specification reads an argument from the minibuffer. Explicit use of the minibuffer during the execution of the command itself does not cause the command to be considered complex.

Variable: **command-history**

This variable's value is a list of recent complex commands, each represented as a form to evaluate. It continues to accumulate all complex commands for the duration of the editing session, but all but the first (most recent) thirty elements are deleted when a garbage collection takes place (see section Garbage Collection).

```
command-history
=> ((switch-to-buffer "chistory.texi")
```


```
(describe-key "^X^[")
(visit-tags-table "~/emacs/src/")
(find-tag "repeat-complex-command"))
```

This history list is actually a special case of minibuffer history (see section Minibuffer History), with one special twist: the elements are expressions rather than strings.

There are a number of commands devoted to the editing and recall of previous commands. The commands `repeat-complex-command`, and `list-command-history` are described in the user manual (see section 'Repetition' in The GNU Emacs Manual). Within the minibuffer, the history commands used are the same ones available in any minibuffer.

Keyboard Macros

A keyboard macro is a canned sequence of input events that can be considered a command and made the definition of a key. Don't confuse keyboard macros with Lisp macros (see section Macros).

Function: **execute-kbd-macro** *macro &optional count*

This function executes macro as a sequence of events. If macro is a string or vector, then the events in it are executed exactly as if they had been input by the user. The sequence is *not* expected to be a single key sequence; normally a keyboard macro definition consists of several key sequences concatenated.

If macro is a symbol, then its function definition is used in place of macro. If that is another symbol, this process repeats. Eventually the result should be a string or vector. If the result is not a symbol, string, or vector, an error is signaled.

The argument count is a repeat count; macro is executed that many times. If count is omitted or `nil`, macro is executed once. If it is 0, macro is executed over and over until it encounters an error or a failing search.

Variable: **last-kbd-macro**

This variable is the definition of the most recently defined keyboard macro. Its value is a string or vector, or `nil`.

Variable: **executing-macro**

This variable contains the string or vector that defines the keyboard macro that is currently executing. It is `nil` if no macro is currently executing.

Variable: **defining-kbd-macro**

This variable indicates whether a keyboard macro is being defined. It is set to `t` by `start-kbd-macro`, and `nil` by `end-kbd-macro`. You can use this variable to make a command behave differently when run from a keyboard macro (perhaps indirectly by calling `interactive-p`). However, do not set this variable yourself.

The commands are described in the user's manual (see section 'Keyboard Macros' in The GNU Emacs

TOC

Manual).

Go to the previous, next section.

Go to the previous, next section.

Keymaps

The bindings between input events and commands are recorded in data structures called keymaps. Each binding in a keymap associates (or binds) an individual event type either with another keymap or with a command. When an event is bound to a keymap, that keymap is used to look up the next character typed; this continues until a command is found. The whole process is called key lookup.

Keymap Terminology

A keymap is a table mapping event types to definitions (which can be any Lisp objects, though only certain types are meaningful for execution by the command loop). Given an event (or an event type) and a keymap, Emacs can get the event's definition. Events include ordinary ASCII characters, function keys, and mouse actions (see section Input Events).

A sequence of input events that form a unit is called a key sequence, or key for short. A sequence of one event is always a key sequence, and so are some multi-event sequences.

A keymap determines a binding or definition for any key sequence. If the key sequence is a single event, its binding is the definition of the event in the keymap. The binding of a key sequence of more than one event is found by an iterative process: the binding of the first event is found, and must be a keymap; then the second event's binding is found in that keymap, and so on until all the events in the key sequence are used up.

If the binding of a key sequence is a keymap, we call the key sequence a prefix key. Otherwise, we call it a complete key (because no more characters can be added to it). If the binding is `nil`, we call the key undefined. Examples of prefix keys are C-c, C-x, and C-x 4. Examples of defined complete keys are X, RET, and C-x 4 C-f. Examples of undefined complete keys are C-x C-g, and C-c 3. See section Prefix Keys, for more details.

The rule for finding the binding of a key sequence assumes that the intermediate bindings (found for the events before the last) are all keymaps; if this is not so, the sequence of events does not form a unit--it is not really a key sequence. In other words, removing one or more events from the end of any valid key must always yield a prefix key. For example, C-f C-f is not a key; C-f is not a prefix key, so a longer sequence starting with C-f cannot be a key.

Note that the set of possible multi-event key sequences depends on the bindings for prefix keys; therefore, it can be different for different keymaps, and can change when bindings are changed. However, a one-event sequence is always a key sequence, because it does not depend on any prefix keys for its well-formedness.

At any time, several primary keymaps are active---that is, in use for finding key bindings. These are the global map, which is shared by all buffers; the local keymap, which is usually associated with a specific major mode; and zero or more minor mode keymaps which belong to currently enabled minor modes.

(Not all minor modes have keymaps.) The local keymap bindings shadow (i.e., take precedence over) corresponding global bindings. The minor mode keymaps shadow both local and global keymaps. See section Active Keymaps, for details.

Format of Keymaps

A keymap is a list whose CAR is the symbol `keymap`. The remaining elements of the list define the key bindings of the keymap. Use the function `keymapp` (see below) to test whether an object is a keymap.

An ordinary element is a cons cell of the form `(type . binding)`. This specifies one binding which applies to events of type type. Each ordinary binding applies to events of a particular event type, which is always a character or a symbol. See section Classifying Events.

A cons cell whose CAR is `t` is a default key binding; any event not bound by other elements of the keymap is given binding as its binding. Default bindings allow a keymap to bind all possible event types without having to enumerate all of them. A keymap that has a default binding completely masks any lower-precedence keymap.

If an element of a keymap is a vector, the vector counts as bindings for all the ASCII characters; vector element n is the binding for the character with code n. This is a more compact way to record lots of bindings. A keymap with such a vector is called a full keymap. Other keymaps are called sparse keymaps.

When a keymap contains a vector, it always defines a binding for every ASCII character even if the vector element is `nil`. Such a binding of `nil` overrides any default binding in the keymap. However, default bindings are still meaningful for events that are not ASCII characters. A binding of `nil` does *not* override lower-precedence keymaps; thus, if the local map gives a binding of `nil`, Emacs uses the binding from the global map.

Aside from bindings, a keymap can also have a string as an element. This is called the overall prompt string and makes it possible to use the keymap as a menu. See section Menu Keymaps.

Keymaps do not directly record bindings for the meta characters, whose codes are from 128 to 255. Instead, meta characters are regarded for purposes of key lookup as sequences of two characters, the first of which is ESC (or whatever is currently the value of `meta-prefix-char`). Thus, the key M-a is really represented as ESC a, and its global binding is found at the slot for a in `esc-map`.

Here as an example is the local keymap for Lisp mode, a sparse keymap. It defines bindings for DEL and TAB, plus C-c C-l, M-C-q, and M-C-x.

```
lisp-mode-map
=>
(keymap
 ;; TAB
 (9 . lisp-indent-line)
 ;; DEL
 (127 . backward-delete-char-untabify)
```


```
(3 keymap
   ;; C-c C-l
   (12 . run-lisp))
(27 keymap
    ;; M-C-q, treated as ESC C-q
    (17 . indent-sexp)
    ;; M-C-x, treated as ESC C-x
    (24 . lisp-send-defun)))
```

Function: **keymapp** *object*

This function returns `t` if object is a keymap, `nil` otherwise. Practically speaking, this function tests for a list whose CAR is `keymap`.

```
(keymapp '(keymap))
    => t
(keymapp (current-global-map))
    => t
```

Creating Keymaps

Here we describe the functions for creating keymaps.

Function: **make-keymap** *&optional prompt*

This function creates and returns a new full keymap (i.e., one which contains a vector of length 128 for defining all the ASCII characters). The new keymap initially binds all ASCII characters to `nil`, and does not bind any other kind of event.

```
(make-keymap)
    => (keymap [nil nil nil ... nil nil])
```

If you specify prompt, that becomes the overall prompt string for the keymap. The prompt string is useful for menu keymaps (see section Menu Keymaps).

Function: **make-sparse-keymap** *&optional prompt*

This function creates and returns a new sparse keymap with no entries. The new keymap does not bind any events. The argument prompt specifies a prompt string, as in `make-keymap`.

```
(make-sparse-keymap)
    => (keymap)
```

Function: **copy-keymap** *keymap*

This function returns a copy of keymap. Any keymaps which appear directly as bindings in keymap are also copied recursively, and so on to any number of levels. However, recursive copying does not take

place when the definition of a character is a symbol whose function definition is a keymap; the same symbol appears in the new copy.

```
(setq map (copy-keymap (current-local-map)))
=> (keymap
     ;; (This implements meta characters.)
     (27 keymap
         (83 . center-paragraph)
         (115 . center-line))
     (9 . tab-to-tab-stop))

(eq map (current-local-map))
    => nil
(equal map (current-local-map))
    => t
```

Inheritance and Keymaps

A keymap can inherit the bindings of another keymap. Do do this, make a keymap whose "tail" is another existing keymap to inherit from. Such a keymap looks like this:

```
(keymap bindings... . other-keymap)
```

The effect is that this keymap inherits all the bindings of other-keymap, whatever they may be at the time a key is looked up, but can add to them or override them with bindings.

If you change the bindings in other-keymap using `define-key` or other key-binding functions, these changes are visible in the inheriting keymap unless shadowed by bindings. The converse is not true: if you use `define-key` to change the inheriting keymap, that affects bindings, but has no effect on other-keymap.

Here is an example showing how to make a keymap that inherits from `text-mode-map`:

```
(setq my-mode-map (cons 'keymap text-mode-map))
```

Prefix Keys

A prefix key has an associated keymap which defines what to do with key sequences that start with the prefix key. For example, C-x is a prefix key, and it uses a keymap which is also stored in the variable `ctl-x-map`. Here is a list of the standard prefix keys of Emacs and their keymaps:

- `esc-map` is used for events that follow ESC. Thus, the global definitions of all meta characters are actually found here. This map is also the function definition of `ESC-prefix`.
- `help-map` is used for events that follow C-h.
- `mode-specific-map` is for events that follow C-c. This map is not actually mode specific; its

name was chosen to be informative for the user in C-h b (`display-bindings`), where it describes the main use of the C-c prefix key.

- `ctl-x-map` is the variable name for the map used for events that follow C-x. This map is also the function definition of `Control-X-prefix`.
- `ctl-x-4-map` is used for events that follow C-x 4.
- `ctl-x-5-map` used is for events that follow C-x 5.
- A nameless keymap is used for events that follow C-x n. Others are used for events following C-x r and C-x a.

The binding of a prefix key is the keymap to use for looking up the events that follow the prefix key. (It may instead be a symbol whose function definition is a keymap. The effect is the same, but the symbol serves as a name for the prefix key.) Thus, the binding of C-x is the symbol `Control-X-prefix`, whose function definition is the keymap for C-x commands. (The same keymap is also the value of `ctl-x-map`.)

Prefix key definitions of this sort can appear in any active keymap. The definitions of C-c, C-x, C-h and ESC as prefix keys appear in the global map, so these prefix keys are always available. Major and minor modes can redefine a key as a prefix by putting a prefix key definition for it in the local map or the minor mode's map. See section Active Keymaps.

If a key is defined as a prefix in more than one active map, then the various definitions are in effect merged: the commands defined in the minor mode keymaps come first, followed by those in the local map's prefix definition, and then by those from the global map.

In the following example, we make C-p a prefix key in the local keymap, in such a way that C-p is identical to C-x. Then the binding for C-p C-f is the function `find-file`, just like C-x C-f. The key sequence C-p 6 is not found in any active keymap.

```
(use-local-map (make-sparse-keymap))
    => nil
(local-set-key "\C-p" ctl-x-map)
    => nil
(key-binding "\C-p\C-f")
    => find-file

(key-binding "\C-p6")
    => nil
```

Function: **define-prefix-command** *symbol*

This function defines symbol as a prefix command: it creates a full keymap and stores it as symbol's function definition. Storing the symbol as the binding of a key makes the key a prefix key which has a name. It also sets symbol as a variable, to have the keymap as its value. The function returns symbol.

In Emacs version 18, only the function definition of symbol was set, not the value as a variable.

Menu Keymaps

A keymap can define a menu as well as ordinary keys and mouse button meanings. Menus are normally actuated with the mouse, but they can work with the keyboard also.

Defining Menus

A keymap is suitable for menu use if it has an overall prompt string, which is a string that appears as an element of the keymap. (See section Format of Keymaps.) The string should describe the purpose of the menu. The easiest way to construct a keymap with a prompt string is to specify the string as an argument when you call `make-keymap` or `make-sparse-keymap` (see section Creating Keymaps).

The individual bindings in the menu keymap should also have prompt strings; these strings become the items displayed in the menu. A binding with a prompt string looks like this:

```
(string . real-binding)
```

As far as `define-key` and `lookup-key` are concerned, the string is part of the event's binding. However, only real-binding is used for executing the key.

You can also supply a second string, called the help string, as follows:

```
(string help-string . real-binding)
```

Currently Emacs does not actually use help-string; it knows only how to ignore help-string in order to extract real-binding. In the future we hope to make help-string serve as extended documentation for the menu item, available on request.

The prompt string for a binding should be short--one or two words. It should describe the action of the command it corresponds to.

If real-binding is `nil`, then string appears in the menu but cannot be selected.

If real-binding is a symbol, and has a non-`nil` `menu-enable` property, that property is an expression which controls whether the menu item is enabled. Every time the keymap is used to display a menu, Emacs evaluates the expression, and it enables the menu item only if the expression's value is non-`nil`. When a menu item is disabled, it is displayed in a "fuzzy" fashion, and cannot be selected with the mouse.

The order of items in the menu is the same as the order of bindings in the keymap. Since `define-key` puts new bindings at the front, you should define the menu items starting at the bottom of the menu and moving to the top, if you care about the order.

Menus and the Mouse

The way to make a menu keymap produce a menu is to make it the definition of a prefix key.

When the prefix key ends with a mouse event, Emacs handles the menu keymap by popping up a visible menu, so that the user can select a choice with the mouse. When the user clicks on a menu item, the event generated is whatever character or symbol has the binding which brought about that menu item.

It's often best to use a button-down event to trigger the menu. Then the user can select a menu item by releasing the button.

A single keymap can appear as multiple menu panes, if you explicitly arrange for this. The way to do this is to make a keymap for each pane, then create a binding for each of those maps in the main keymap of the menu. Give each of these bindings a prompt string that starts with `@'. The rest of the prompt string becomes the name of the pane. See the file `lisp/mouse.el' for an example of this. Any ordinary bindings with `@'-less prompt strings are grouped into one pane, which appears along with the other panes explicitly created for the submaps.

You can also get multiple panes from separate keymaps. The full definition of a prefix key always comes from merging the definitions supplied by the various active keymaps (minor mode, local, and global). When more than one of these keymaps is a menu, each of them makes a separate pane or panes. See section Active Keymaps.

A Lisp program can explicitly pop up a menu and receive the user's choice. You can use keymaps for this also. See section Pop-Up Menus.

Menus and the Keyboard

When a prefix key ending with a keyboard event (a character or function key) has a definition that is a menu keymap, the user can use the keyboard to choose a menu item.

Emacs displays the menu alternatives (the prompt strings of the bindings) in the echo area. If they don't all fit at once, the user can type SPC to see the next line of alternatives. Successive uses of SPC eventually get to the end of the menu and then cycle around to the beginning.

When the user has found the desired alternative from the menu, he or she should type the corresponding character--the one whose binding is that alternative.

In a menu intended for keyboard use, each menu item must clearly indicate what character to type. The best convention to use is to make the character the first letter of the menu item prompt string. That is something users will understand without being told.

This way of using menus in an Emacs-like editor was inspired by the Hierarkey system.

Variable: **menu-prompt-more-char**

This variable specifies the character to use to ask to see the next line of a menu. Its initial value is 32, the code for SPC.

Menu Example

Here is a simple example of how to set up a menu for mouse use.

```
(defvar my-menu-map
  (make-sparse-keymap "Key Commands <==> Functions"))
(fset 'help-for-keys my-menu-map)

(define-key my-menu-map [bindings]
  '("List all keystroke commands" . describe-bindings))
(define-key my-menu-map [key]
  '("Describe key briefly" . describe-key-briefly))
(define-key my-menu-map [key-verbose]
  '("Describe key verbose" . describe-key))
(define-key my-menu-map [function]
  '("Describe Lisp function" . describe-function))
(define-key my-menu-map [where-is]
  '("Where is this command" . where-is))

(define-key global-map [C-S-down-mouse-1] 'help-for-keys)
```

The symbols used in the key sequences bound in the menu are fictitious "function keys"; they don't appear on the keyboard, but that doesn't stop you from using them in the menu. Their names were chosen to be mnemonic, because they show up in the output of `where-is` and `apropos` to identify the corresponding menu items.

However, if you want the menu to be usable from the keyboard as well, you must use real ASCII characters instead of fictitious function keys.

The Menu Bar

Under X Windows, each frame can have a menu bar---a permanently displayed menu stretching horizontally across the top of the frame. The items of the menu bar are the subcommands of the fake "function key" `menu-bar`, as defined by all the active keymaps.

To add an item to the menu bar, invent a fake "function key" of your own (let's call it key), and make a binding for the key sequence `[menu-bar key]`. Most often, the binding is a menu keymap, so that pressing a button on the menu bar item leads to another menu.

When more than one active keymap defines the same fake function key for the menu bar, the item appears just once. If the user clicks on that menu bar item, it brings up a single, combined submenu containing all the subcommands of that item--the global subcommands, the local subcommands, and the minor mode subcommands, all together.

In order for a frame to display a menu bar, its `menu-bar-lines` property must be greater than zero. Emacs uses just one line for the menu bar itself; if you specify more than one line, the other lines serve to separate the menu bar from the windows in the frame. We recommend you try one or two as the value of

`menu-bar-lines`. See section X Window Frame Parameters.

Here's an example of setting up a menu bar item:

```
(modify-frame-parameters (selected-frame) '((menu-bar-lines . 2)))

;; Make a menu keymap (with a prompt string)
;; to be the menu bar item's definition.
(define-key global-map [menu-bar words]
  (cons "Words" (make-sparse-keymap "Words")))

;; Make specific subcommands in the item's submenu.
(define-key global-map
  [menu-bar words forward]
  '("Forward word" . forward-word))
(define-key global-map
  [menu-bar words backward]
  '("Backward word" . backward-word))
```

Active Keymaps

Emacs normally contains many keymaps; at any given time, just a few of them are active in that they participate in the interpretation of user input. These are the global keymap, the current buffer's local keymap, and the keymaps of any enabled minor modes.

The global keymap holds the bindings of keys that are defined regardless of the current buffer, such as C-f. The variable `global-map` holds this keymap, which is always active.

Each buffer may have another keymap, its local keymap, which may contain new or overriding definitions for keys. At all times, the current buffer's local keymap is active. Text properties can specify an alternative local map for certain parts of the buffer; see section Special Properties.

Each minor mode may have a keymap; if it does, the keymap is active whenever the minor mode is enabled.

All the active keymaps are used together to determine what command to execute when a key is entered. The key lookup proceeds as described earlier (see section Key Lookup), but Emacs *first* searches for the key in the minor mode maps (one map at a time); if they do not supply a binding for the key, Emacs searches the local map; if that too has no binding, Emacs then searches the global map.

Since every buffer that uses the same major mode normally uses the very same local keymap, it may appear as if the keymap is local to the mode. A change to the local keymap of a buffer (using `local-set-key`, for example) will be seen also in the other buffers that share that keymap.

The local keymaps that are used for Lisp mode, C mode, and several other major modes exist even if they have not yet been used. These local maps are the values of the variables `lisp-mode-map`, `c-mode-map`, and so on. For most other modes, which are less frequently used, the local keymap is

constructed only when the mode is used for the first time in a session.

The minibuffer has local keymaps, too; they contain various completion and exit commands. See section Minibuffers.

See section Standard Keymaps, for a list of standard keymaps.

Variable: **global-map**

This variable contains the default global keymap that maps Emacs keyboard input to commands. Normally this keymap is the global keymap. The default global keymap is a full keymap that binds `self-insert-command` to all of the printing characters.

Function: **current-global-map**

This function returns the current global keymap. This is always the same as the value of `global-map` unless you change one or the other.

```
(current-global-map)
=> (keymap [set-mark-command beginning-of-line ...
            delete-backward-char])
```

Function: **current-local-map**

This function returns the current buffer's local keymap, or `nil` if it has none. In the following example, the keymap for the `*scratch*' buffer (using Lisp Interaction mode) is a sparse keymap in which the entry for ESC, ASCII code 27, is another sparse keymap.

```
(current-local-map)
=> (keymap
    (10 . eval-print-last-sexp)
    (9 . lisp-indent-line)
    (127 . backward-delete-char-untabify)
    (27 keymap
        (24 . eval-defun)
        (17 . indent-sexp)))
```

Function: **current-minor-mode-maps**

This function returns a list of the keymaps of currently enabled minor modes.

Function: **use-global-map** *keymap*

This function makes keymap the new current global keymap. It returns `nil`.

It is very unusual to change the global keymap.

Function: **use-local-map** *keymap*

This function makes keymap the new current local keymap of the current buffer. If keymap is `nil`, then

there will be no local keymap. It returns `nil`. Most major modes use this function.

Variable: **minor-mode-map-alist**

This variable is an alist describing keymaps that may or may not be active according to the values of certain variables. Its elements look like this:

```
(variable . keymap)
```

The keymap keymap is active whenever variable has a non-`nil` value. Typically variable is the variable which enables or disables a minor mode. See section Keymaps and Minor Modes.

When more than one minor mode keymap is active, their order of priority is the order of `minor-mode-map-alist`.

See also `minor-mode-key-binding` in section Functions for Key Lookup.

Key Lookup

Key lookup is the process of finding the binding of a key sequence from a given keymap. Actual execution of the binding is not part of key lookup.

Key lookup uses just the event types of each event in the key sequence; the rest of the event is ignored. In fact, a key sequence used for key lookup may designate mouse events with just their types (symbols) instead of with entire mouse events (lists). See section Input Events. Such a pseudo-key-sequence is insufficient for `command-execute`, but it is sufficient for looking up or rebinding a key.

When the key sequence consists of multiple events, key lookup processes the events sequentially: the binding of the first event is found, and must be a keymap; then the second event's binding is found in that keymap, and so on until all the events in the key sequence are used up. (The binding thus found for the last event may or may not be a keymap.) Thus, the process of key lookup is defined in terms of a simpler process for looking up a single event in a keymap. How that is done depends on the type of object associated with the event in that keymap.

Let's use the term keymap entry to describe the value directly associated with an event type in a keymap. While any Lisp object may be stored as a keymap entry, not all make sense for key lookup. Here is a list of the meaningful kinds of keymap entries:

`nil`
: `nil` means that the events used so far in the lookup form an undefined key. When a keymap fails to mention an event type at all, that is equivalent to an entry of `nil` for that type.

keymap
: The events used so far in the lookup form a prefix key. The next event of the key sequence is looked up in keymap.

command
: The events used so far in the lookup form a complete key, and command is its binding.

string

vector

The events used so far in the lookup form a complete key, whose binding is a keyboard macro. See section Keyboard Macros, for more information.

list

The meaning of a list depends on the types of the elements of the list.

If the CAR of list is the symbol `keymap`, then the list is a keymap, and is treated as a keymap (see above).

If the CAR of list is `lambda`, then the list is a lambda expression. This is presumed to be a command, and is treated as such (see above).

If the CAR of list is a keymap and the CDR is an event type, then this is an indirect entry:

```
(othermap . othertype)
```

When key lookup encounters an indirect entry, it looks up instead the binding of othertype in othermap and uses that.

This feature permits you to define one key as an alias for another key. For example, an entry whose CAR is the keymap called `esc-map` and whose CDR is 32 (the code for space) means, "Use the global binding of Meta-SPC, whatever that may be."

If the CAR of list is a string, it serves as a menu item name if the keymap is used as a menu. For executing the key, the string is discarded and the CDR of list is used instead. (Any number of strings can be discarded from the front of the list in this way.) See section Menu Keymaps.

symbol

The function definition of symbol is used in place of symbol. If that too is a symbol, then this process is repeated, any number of times. Ultimately this should lead to an object which is a keymap, a command or a keyboard macro. A list is allowed if it is a keymap or a command, but indirect entries are not understood when found via symbols.

Note that keymaps and keyboard macros (strings and vectors) are not valid functions, so a symbol with a keymap, string or vector as its function definition is also invalid as a function. It is, however, valid as a key binding. If the definition is a keyboard macro, then the symbol is also valid as an argument to `command-execute` (see section Interactive Call).

The symbol `undefined` is worth special mention: it means to treat the key as undefined. Strictly speaking, the key is defined, and its binding is the command `undefined`; but that command does the same thing that is done automatically for an undefined key: it rings the bell (by calling `ding`) but does not signal an error.

`undefined` is used in local keymaps to override a global key binding and make the key "undefined" locally. A local binding of `nil` would fail to do this because it would not override the global binding.

anything else

If any other type of object is found, the events used so far in the lookup form a complete key, and the object is its binding, but the binding is not executable as a command.

In short, a keymap entry may be a keymap, a command, a keyboard macro, a symbol which leads to one of them, or an indirection or `nil`. Here is an example of a sparse keymap with two characters bound to commands and one bound to another keymap. This map is the normal value of `emacs-lisp-mode-map`. Note that 9 is the code for TAB, 127 for DEL, 27 for ESC, 17 for C-q and 24 for C-x.

```
(keymap (9 . lisp-indent-line)
        (127 . backward-delete-char-untabify)
        (27 keymap (17 . indent-sexp) (24 . eval-defun)))
```

Functions for Key Lookup

Here are the functions and variables pertaining to key lookup.

Function: **lookup-key** *keymap key &optional accept-defaults*

This function returns the definition of key in keymap. If the string or vector key is not a valid key sequence according to the prefix keys specified in keymap (which means it is "too long" and has extra events at the end), then the value is a number, the number of events at the front of key that compose a complete key.

If accept-defaults is non-`nil`, then `lookup-key` considers default bindings as well as bindings for the specific events in key. Otherwise, `lookup-key` reports only bindings for the specific sequence key, ignoring default bindings except when an element of key is `t`.

All the other functions described in this chapter that look up keys use `lookup-key`.

```
(lookup-key (current-global-map) "\C-x\C-f")
    => find-file
(lookup-key (current-global-map) "\C-x\C-f12345")
    => 2
```

If key contains a meta character, that character is implicitly replaced by a two-character sequence: the value of `meta-prefix-char`, followed by the corresponding non-meta character. Thus, the first example below is handled by conversion into the second example.

```
(lookup-key (current-global-map) "\M-f")
    => forward-word
(lookup-key (current-global-map) "\ef")
    => forward-word
```

This function does not modify the specified events in ways that discard information as

`read-key-sequence` does (see section Key Sequence Input). In particular, it does not convert le to lower case and it does not change drag events to clicks.

Command: **undefined**

Used in keymaps to undefine keys. It calls `ding`, but does not cause an error.

Function: **key-binding** *key &optional accept-defaults*

This function returns the binding for key in the current keymaps, trying all the active keymaps. The result is `nil` if key is undefined in the keymaps.

The argument accept-defaults controls checking for default bindings, as in `lookup-key`.

An error is signaled if key is not a string or a vector.

```
(key-binding "\C-x\C-f")
    => find-file
```

Function: **local-key-binding** *key &optional accept-defaults*

This function returns the binding for key in the current local keymap, or `nil` if it is undefined there.

The argument accept-defaults controls checking for default bindings, as in `lookup-key` (above).

Function: **global-key-binding** *key &optional accept-defaults*

This function returns the binding for command key in the current global keymap, or `nil` if it is undefined there.

The argument accept-defaults controls checking for default bindings, as in `lookup-key` (above).

Function: **minor-mode-key-binding** *key &optional accept-defaults*

This function returns a list of all the active minor mode bindings of key. More precisely, it returns an alist of pairs `(modename . binding)`, where modename is the the variable which enables the minor mode, and binding is key's binding in that mode. If key has no minor-mode bindings, the value is `nil`.

If the first binding is a non-prefix, all subsequent bindings from other minor modes are omitted, since they would be completely shadowed. Similarly, the list omits non-prefix bindings that follow prefix bindings.

The argument accept-defaults controls checking for default bindings, as in `lookup-key` (above).

Variable: **meta-prefix-char**

This variable is the meta-prefix character code. It is used when translating a meta character to a two-character sequence so it can be looked up in a keymap. For useful results, the value should be a prefix event (see section Prefix Keys). The default value is 27, which is the ASCII code for ESC.

As long as the value of `meta-prefix-char` remains 27, key lookup translates M-b into ESC b, which is normally defined as the `backward-word` command. However, if you set `meta-prefix-char` to

24, the code for C-x, then Emacs will translate M-b into C-x b, whose standard binding is the `switch-to-buffer` command.

```
meta-prefix-char                        ; The default value.
     => 27
(key-binding "\M-b")
     => backward-word
?\C-x                                   ; The print representation
     => 24                              ;   of a character.
(setq meta-prefix-char 24)
     => 24
(key-binding "\M-b")
     => switch-to-buffer                ; Now, typing M-b is
                                        ;   like typing C-x b.

(setq meta-prefix-char 27)              ; Avoid confusion!
     => 27                              ; Restore the default value!
```

Changing Key Bindings

The way to rebind a key is to change its entry in a keymap. You can change the global keymap, so that the change is effective in all buffers (except those that override the global binding with a local one). Or you can change the current buffer's local map, which usually affects all buffers using the same major mode. The `global-set-key` and `local-set-key` functions are convenient interfaces for these operations. Or you can use `define-key` and specify explicitly which map to change.

People often use `global-set-key` in their `.emacs' file for simple customization. For example,

```
(global-set-key "\C-x\C-\\" 'next-line)
```

or

```
(global-set-key [?\C-x ?\C-\\] 'next-line)
```

redefines C-x C-\ to move down a line.

```
(global-set-key [M-mouse-1] 'mouse-set-point)
```

redefines the first (leftmost) mouse button, typed with the Meta key, to set point where you click.

In writing the key sequence to rebind, it is useful to use the special escape sequences for control and meta characters (see section String Type). The syntax `\C-' means that the following character is a control character and `\M-' means that the following character is a meta character. Thus, the string `"\M-x"` is read as containing a single M-x, `"\C-f"` is read as containing a single C-f, and `"\M-\C-x"` and `"\C-\M-x"` are both read as containing a single C-M-x.

For the functions below, an error is signaled if keymap is not a keymap or if key is not a string or ve representing a key sequence. However, you can use event types (symbols) as shorthand for events th are lists.

Function: **define-key** *keymap key binding*

This function sets the binding for key in keymap. (If key is more than one event long, the change is actually made in another keymap reached from keymap.) The argument binding can be any Lisp object, but only certain types are meaningful. (For a list of meaningful types, see section Key Lookup.) The value returned by `define-key` is binding.

Every prefix of key must be a prefix key (i.e., bound to a keymap) or undefined; otherwise an error is signaled.

If some prefix of key is undefined, then `define-key` defines it as a prefix key so that the rest of key may be defined as specified.

The following example creates a sparse keymap and makes a number of bindings:

```
(setq map (make-sparse-keymap))
    => (keymap)
(define-key map "\C-f" 'forward-char)
    => forward-char
map
    => (keymap (6 . forward-char))

;; Build sparse submap for C-x and bind f in that.
(define-key map "\C-xf" 'forward-word)
    => forward-word
map
=> (keymap
    (24 keymap                ; C-x
        (102 . forward-word)) ;      f
    (6 . forward-char))       ; C-f

;; Bind C-p to the ctl-x-map.
(define-key map "\C-p" ctl-x-map)
;; ctl-x-map
=> [nil ... find-file ... backward-kill-sentence]

;; Bind C-f to foo in the ctl-x-map.
(define-key map "\C-p\C-f" 'foo)
=> 'foo
map
=> (keymap     ; Note foo in ctl-x-map.
    (16 keymap [nil ... foo ... backward-kill-sentence])
    (24 keymap
```


```
        (102 . forward-word))
    (6 . forward-char))
```

Note that storing a new binding for C-p C-f actually works by changing an entry in `ctl-x-map`, and this has the effect of changing the bindings of both C-p C-f and C-x C-f in the default global map.

Function: **substitute-key-definition** *olddef newdef keymap &optional oldmap*

This function replaces olddef with newdef for any keys in keymap that were bound to olddef. In other words, olddef is replaced with newdef wherever it appears. The function returns `nil`.

For example, this redefines C-x C-f, if you do it in an Emacs with standard bindings:

```
(substitute-key-definition
 'find-file 'find-file-read-only (current-global-map))
```

If oldmap is non-`nil`, then its bindings determine which keys to rebind. The rebindings still happen in newmap, not in oldmap. Thus, you can change one map under the control of the bindings in another. For example,

```
(substitute-key-definition
  'delete-backward-char 'my-funny-delete
  my-map global-map)
```

puts the special deletion command in `my-map` for whichever keys are globally bound to the standard deletion command.

Here is an example showing a keymap before and after substitution:

```
(setq map '(keymap
            (?1 . olddef-1)
            (?2 . olddef-2)
            (?3 . olddef-1)))
=> (keymap (49 . olddef-1) (50 . olddef-2) (51 . olddef-1))

(substitute-key-definition 'olddef-1 'newdef map)
=> nil
map
=> (keymap (49 . newdef) (50 . olddef-2) (51 . newdef))
```

Function: **suppress-keymap** *keymap &optional nodigits*

This function changes the contents of the full keymap keymap by replacing the self-insertion commands for numbers with the `digit-argument` function, unless nodigits is non-`nil`, and by replacing the functions for the rest of the printing characters with `undefined`. This means that ordinary insertion of text is impossible in a buffer with a local keymap on which `suppress-keymap` has been called.

`suppress-keymap` returns `nil`.

The `suppress-keymap` function does not make it impossible to modify a buffer, as it does not suppress commands such as `yank` and `quoted-insert`. To prevent any modification of a buffer, make it read-only (see section Read-Only Buffers).

Since this function modifies keymap, you would normally use it on a newly created keymap. Operating on an existing keymap that is used for some other purpose is likely to cause trouble; for example, suppressing `global-map` would make it impossible to use most of Emacs.

Most often, `suppress-keymap` is used to initialize local keymaps of modes such as Rmail and Dired where insertion of text is not desirable and the buffer is read-only. Here is an example taken from the file `` `emacs/lisp/dired.el' ``, showing how the local keymap for Dired mode is set up:

```
...
(setq dired-mode-map (make-keymap))
(suppress-keymap dired-mode-map)
(define-key dired-mode-map "r" 'dired-rename-file)
(define-key dired-mode-map "\C-d" 'dired-flag-file-deleted)
(define-key dired-mode-map "d" 'dired-flag-file-deleted)
(define-key dired-mode-map "v" 'dired-view-file)
(define-key dired-mode-map "e" 'dired-find-file)
(define-key dired-mode-map "f" 'dired-find-file)
...
```

Commands for Binding Keys

This section describes some convenient interactive interfaces for changing key bindings. They work by calling `define-key`.

Command: **global-set-key** *key definition*

This function sets the binding of key in the current global map to definition.

```
(global-set-key key definition)
==
(define-key (current-global-map) key definition)
```

Command: **global-unset-key** *key*

This function removes the binding of key from the current global map.

One use of this function is in preparation for defining a longer key which uses it implicitly as a prefix--which would not be allowed if key has a non-prefix binding. For example:

```
(global-unset-key "\C-l")
    => nil
(global-set-key "\C-l\C-l" 'redraw-display)
```


```
    => nil
```

This function is implemented simply using `define-key`:

```
(global-unset-key key)
==
(define-key (current-global-map) key nil)
```

Command: **local-set-key** *key definition*

This function sets the binding of key in the current local keymap to definition.

```
(local-set-key key definition)
==
(define-key (current-local-map) key definition)
```

Command: **local-unset-key** *key*

This function removes the binding of key from the current local map.

```
(local-unset-key key)
==
(define-key (current-local-map) key nil)
```

Scanning Keymaps

This section describes functions used to scan all the current keymaps for the sake of printing help information.

Function: **accessible-keymaps** *keymap*

This function returns a list of all the keymaps that can be accessed (via prefix keys) from keymap. The value is an association list with elements of the form `(key . map)`, where key is a prefix key whose definition in keymap is map.

The elements of the alist are ordered so that the key increases in length. The first element is always `("" . keymap)`, because the specified keymap is accessible from itself with a prefix of no events.

In the example below, the returned alist indicates that the key ESC, which is displayed as `^[', is a prefix key whose definition is the sparse keymap `(keymap (83 . center-paragraph) (115 . foo))`.

```
(accessible-keymaps (current-local-map))
=>(("" keymap
      (27 keymap   ; Note this keymap for ESC is repeated below.
          (83 . center-paragraph)
          (115 . center-line))
```


```
     (9 . tab-to-tab-stop))

   ("^[" keymap
    (83 . center-paragraph)
    (115 . foo)))
```

In the following example, C-h is a prefix key that uses a sparse keymap starting `(keymap (118 . describe-variable)...)`. Another prefix, C-x 4, uses a keymap which happens to be `ctl-x-4-map`. The event `mode-line` is one of several dummy events used as prefixes for mouse actions in special parts of a window.

```
(accessible-keymaps (current-global-map))
=> (("" keymap [set-mark-command beginning-of-line ...
                delete-backward-char])
    ("^H" keymap (118 . describe-variable) ...
     (8 . help-for-help))
    ("^X" keymap [x-flush-mouse-queue ... backward-kill-sentence])
    ("^[" keymap [mark-sexp backward-sexp ... backward-kill-word])
    ("^X4" keymap (15 . display-buffer) ...)
    ([mode-line] keymap
     (S-mouse-2 . mouse-split-window-horizontally) ...))
```

These are not all the keymaps you would see in an actual case.

Function: **where-is-internal** *command &optional keymap firstonly*

This function returns a list of key sequences (of any length) that are bound to command in keymap and the global keymap. The argument command can be any object; it is compared with all keymap entries using `eq`. If keymap is not supplied, then the global map alone is used.

If firstonly is non-`nil`, then the value is a single string representing the first key sequence found, rather than a list of all possible key sequences.

This function is used by `where-is` (see section 'Help' in The GNU Emacs Manual).

```
(where-is-internal 'describe-function)
    => ("\^hf" "\^hd")
```

Command: **describe-bindings**

This function creates a listing of all defined keys, and their definitions. The listing is put in a buffer named `*Help*', which is then displayed in a window.

A meta character is shown as ESC followed by the corresponding non-meta character. Control characters are indicated with C-.

When several characters with consecutive ASCII codes have the same definition, they are shown together, as `firstchar..lastchar'. In this instance, you need to know the ASCII codes to understand which characters this means. For example, in the default global map, the characters `SPC .. ~' are described by a

single line. SPC is ASCII 32, ~ is ASCII 126, and the characters between them include all the normal printing characters, (e.g., letters, digits, punctuation, etc.); all these characters are bound to `self-insert-command`.

Go to the previous, next section.

Go to the previous, next section.

Major and Minor Modes

A mode is a set of definitions that customize Emacs and can be turned on and off while you edit. There are two varieties of modes: major modes, which are mutually exclusive and used for editing particular kinds of text, and minor modes, which provide features that may be enabled individually.

This chapter covers both major and minor modes, the way they are indicated in the mode line, and how they run hooks supplied by the user. Related topics such as keymaps and syntax tables are covered in separate chapters. (See section Keymaps, and section Syntax Tables.)

Major Modes

Major modes specialize Emacs for editing particular kinds of text. Each buffer has only one major mode at a time.

The least specialized major mode is called Fundamental mode. This mode has no mode-specific definitions or variable settings, so each Emacs command behaves in its default manner, and each option is in its default state. All other major modes redefine various keys and options. For example, Lisp Interaction mode provides special key bindings for LFD (`eval-print-last-sexp`), TAB (`lisp-indent-line`), and other keys.

When you need to write several editing commands to help you perform a specialized editing task, creating a new major mode is usually a good idea. In practice, writing a major mode is easy (in contrast to writing a minor mode, which is often difficult).

If the new mode is similar to an old one, it is often unwise to modify the old one to serve two purposes, since it may become harder to use and maintain. Instead, copy and rename an existing major mode definition and alter it for its new function. For example, Rmail Edit mode, which is in `` `emacs/lisp/rmailedit.el' ``, is a major mode that is very similar to Text mode except that it provides three additional commands. Its definition is distinct from that of Text mode, but was derived from it.

Rmail Edit mode is an example of a case where one piece of text is put temporarily into a different major mode so it can be edited in a different way (with ordinary Emacs commands rather than Rmail). In such cases, the temporary major mode usually has a command to switch back to the buffer's usual mode (Rmail mode, in this case). You might be tempted to present the temporary redefinitions inside a recursive edit and restore the usual ones when the user exits; but this is a bad idea because it constrains the user's options when it is done in more than one buffer: recursive edits must be exited most-recently-entered first. Using alternative major modes avoids this limitation. See section Recursive Editing.

The standard GNU Emacs Lisp library directory contains the code for several major modes, in files including `` `text-mode.el' ``, `` `texinfo.el' ``, `` `lisp-mode.el' ``, `` `c-mode.el' ``, and `` `rmail.el' ``. You can look at these libraries to see how modes are written. Text mode is perhaps the simplest major mode aside from Fundamental mode. Rmail mode is a rather complicated, full-featured mode.

Major Mode Conventions

The code for existing major modes follows various coding conventions, including conventions for local keymap and syntax table initialization, global names, and hooks. Please keep these conventions in mind when you create a new major mode:

- Define a command whose name ends in `-mode', with no arguments, that switches to the new mode in the current buffer. This command should set up the keymap, syntax table, and local variables in an existing buffer without changing the buffer's text.
- Write a documentation string for this command which describes the special commands available in this mode. C-h m (`describe-mode`) will print this.

  The documentation string may include the special documentation substrings, `\[command]', `\{keymap}', and `\<keymap>', that enable the documentation to adapt automatically to the user's own key bindings. See section Substituting Key Bindings in Documentation. The `describe-mode` function replaces these special documentation substrings with their current meanings. See section Access to Documentation Strings.
- The major mode command should set the variable `major-mode` to the major mode command symbol. This is how `describe-mode` discovers which documentation to print.
- The major mode command should set the variable `mode-name` to the "pretty" name of the mode, as a string. This appears in the mode line.
- Since all global names are in the same name space, all the global variables, constants, and functions that are part of the mode should have names that start with the major mode name (or with an abbreviation of it if the name is long). See section Writing Clean Lisp Programs.
- The major mode should usually have its own keymap, which is used as the local keymap in all buffers in that mode. The major mode function should call `use-local-map` to install this local map. See section Active Keymaps, for more information.

  This keymap should be kept in a global variable named `modename-mode-map`. Normally the library that defines the mode sets this variable. Use `defvar` to set the variable, so that it is not reinitialized if it already has a value. (Such reinitialization could discard customizations made by the user.)
- The mode may have its own syntax table or may share one with other related modes. If it has its own syntax table, it should store this in a variable named `modename-mode-syntax-table`. The reasons for this are the same as for using a keymap variable. See section Syntax Tables.
- The mode may have its own abbrev table or may share one with other related modes. If it has its own abbrev table, it should store this in a variable named `modename-mode-abbrev-table`. See section Abbrev Tables.
- To give a variable a buffer-local binding, use `make-local-variable` in the major mode command, not `make-variable-buffer-local`. The latter function would make the variable local to every buffer in which it is subsequently set, which would affect buffers that do not use this mode. It is undesirable for a mode to have such global effects. See section Buffer-Local Variables.
- If hooks are appropriate for the mode, the major mode command should run the hooks after completing all other initialization so the user may further customize any of the settings. See section Hooks.
- If this mode is appropriate only for specially-prepared text, then the major mode command symbol should have a property named `mode-class` with value `special`, put on as follows:


```
(put 'funny-mode 'mode-class 'special)
```

This tells Emacs that new buffers created while the current buffer has Funny mode should not inherit Funny mode. Modes such as Dired, Rmail, and Buffer List use this feature.

- If it is desirable that Emacs use the new mode by default after visiting files with certain recognizable names, add an element to `auto-mode-alist` to select the mode for those file names. If you define the mode command to autoload, you should add this element in the same file that calls `autoload`. Otherwise, it is sufficient to add the element in the file that contains the mode definition. See section How Emacs Chooses a Major Mode.
- In the documentation, you should provide a sample `autoload` form and an example of how to add to `auto-mode-alist`, that users can include in their `.emacs' files.
- The top level forms in the file defining the mode should be written so that they may be evaluated more than once without adverse consequences. Even if you never load the file more than once, someone else will.

Major Mode Examples

Text mode is perhaps the simplest mode besides Fundamental mode. Here are excerpts from `text-mode.el' that illustrate many of the conventions listed above:

```
;; Create mode-specific tables.
(defvar text-mode-syntax-table nil
  "Syntax table used while in text mode.")

(if text-mode-syntax-table
    ()              ; Do not change the table if it is already set up.
  (setq text-mode-syntax-table (make-syntax-table))
  (modify-syntax-entry ?\" ".   " text-mode-syntax-table)
  (modify-syntax-entry ?\\ ".   " text-mode-syntax-table)
  (modify-syntax-entry ?' "w   " text-mode-syntax-table))

(defvar text-mode-abbrev-table nil
  "Abbrev table used while in text mode.")
(define-abbrev-table 'text-mode-abbrev-table ())

(defvar text-mode-map nil)   ; Create a mode-specific keymap.

(if text-mode-map
    ()              ; Do not change the keymap if it is already set up.
  (setq text-mode-map (make-sparse-keymap))
  (define-key text-mode-map "\t" 'tab-to-tab-stop)
  (define-key text-mode-map "\es" 'center-line)
  (define-key text-mode-map "\eS" 'center-paragraph))
```

Here is the complete major mode function definition for Text mode:


```
(defun text-mode ()
  "Major mode for editing text intended for humans to read.
 Special commands: \\{text-mode-map}
Turning on text-mode runs the hook `text-mode-hook'."
  (interactive)
  (kill-all-local-variables)
  (use-local-map text-mode-map)     ; This provides the local keymap.
  (setq mode-name "Text")           ; This name goes into the mode line.
  (setq major-mode 'text-mode)      ; This is how describe-mode
                                    ;   finds the doc string to print.
  (setq local-abbrev-table text-mode-abbrev-table)
  (set-syntax-table text-mode-syntax-table)
  (run-hooks 'text-mode-hook))      ; Finally, this permits the user to
                                    ;   customize the mode with a hook.
```

The three Lisp modes (Lisp mode, Emacs Lisp mode, and Lisp Interaction mode) have more features than Text mode and the code is correspondingly more complicated. Here are excerpts from `lisp-mode.el' that illustrate how these modes are written.

```
;; Create mode-specific table variables.
(defvar lisp-mode-syntax-table nil "")
(defvar emacs-lisp-mode-syntax-table nil "")
(defvar lisp-mode-abbrev-table nil "")

(if (not emacs-lisp-mode-syntax-table) ; Do not change the table
                                       ;   if it is already set.
    (let ((i 0))
      (setq emacs-lisp-mode-syntax-table (make-syntax-table))

      ;; Set syntax of chars up to 0 to class of chars that are
      ;;   part of symbol names but not words.
      ;;   (The number 0 is 48 in the ASCII character set.)
      (while (< i ?0)
        (modify-syntax-entry i "_   " emacs-lisp-mode-syntax-table)
        (setq i (1+ i)))
      ...
      ;; Set the syntax for other characters.
      (modify-syntax-entry ?  "    " emacs-lisp-mode-syntax-table)
      (modify-syntax-entry ?\t "    " emacs-lisp-mode-syntax-table)
      ...
      (modify-syntax-entry ?\( "()  " emacs-lisp-mode-syntax-table)
      (modify-syntax-entry ?\) ")(  " emacs-lisp-mode-syntax-table)
      ...))
;; Create an abbrev table for lisp-mode.
(define-abbrev-table 'lisp-mode-abbrev-table ())
```

Much code is shared among the three Lisp modes. The following function sets various variables; it is called by each of the major Lisp mode functions:


```
(defun lisp-mode-variables (lisp-syntax)
  ;; The lisp-syntax argument is nil in Emacs Lisp mode,
  ;;   and t in the other two Lisp modes.
  (cond (lisp-syntax
         (if (not lisp-mode-syntax-table)
             ;; The Emacs Lisp mode syntax table always exists, but
             ;;   the Lisp Mode syntax table is created the first time a
             ;;   mode that needs it is called.  This is to save space.
             (progn (setq lisp-mode-syntax-table
                          (copy-syntax-table emacs-lisp-mode-syntax-table))
                    ;; Change some entries for Lisp mode.
                    (modify-syntax-entry ?\| "\"   "
                                         lisp-mode-syntax-table)
                    (modify-syntax-entry ?\[ "_   "
                                         lisp-mode-syntax-table)
                    (modify-syntax-entry ?\] "_   "
                                         lisp-mode-syntax-table)))
          (set-syntax-table lisp-mode-syntax-table)))
  (setq local-abbrev-table lisp-mode-abbrev-table)
  ...)
```

Functions such as `forward-paragraph` use the value of the `paragraph-start` variable. Since Lisp code is different from ordinary text, the `paragraph-start` variable needs to be set specially to handle Lisp. Also, comments are indented in a special fashion in Lisp and the Lisp modes need their own mode-specific `comment-indent-function`. The code to set these variables is the rest of `lisp-mode-variables`.

```
  (make-local-variable 'paragraph-start)
  (setq paragraph-start (concat "^$\\|" page-delimiter))
  ...
  (make-local-variable 'comment-indent-function)
  (setq comment-indent-function 'lisp-comment-indent))
```

Each of the different Lisp modes has a slightly different keymap. For example, Lisp mode binds C-c C-l to `run-lisp`, but the other Lisp modes do not. However, all Lisp modes have some commands in common. The following function adds these common commands to a given keymap.

```
(defun lisp-mode-commands (map)
  (define-key map "\e\C-q" 'indent-sexp)
  (define-key map "\177" 'backward-delete-char-untabify)
  (define-key map "\t" 'lisp-indent-line))
```

Here is an example of using `lisp-mode-commands` to initialize a keymap, as part of the code for Emacs Lisp mode. First we declare a variable with `defvar` to hold the mode-specific keymap. When this `defvar` executes, it sets the variable to `nil` if it was void. Then we set up the keymap if the variable is `nil`.

This code avoids changing the keymap or the variable if it is already set up. This lets the user customize the keymap if he or she so wishes.


```
(defvar emacs-lisp-mode-map () "")

(if emacs-lisp-mode-map
    ()
  (setq emacs-lisp-mode-map (make-sparse-keymap))
  (define-key emacs-lisp-mode-map "\e\C-x" 'eval-defun)
  (lisp-mode-commands emacs-lisp-mode-map))
```

Finally, here is the complete major mode function definition for Emacs Lisp mode.

```
(defun emacs-lisp-mode ()
  "Major mode for editing Lisp code to run in Emacs.
Commands:
Delete converts tabs to spaces as it moves back.
Blank lines separate paragraphs.  Semicolons start comments.
\\{emacs-lisp-mode-map}
Entry to this mode runs the hook `emacs-lisp-mode-hook'."
  (interactive)
  (kill-all-local-variables)
  (use-local-map emacs-lisp-mode-map)    ; This provides the local keymap.
  (set-syntax-table emacs-lisp-mode-syntax-table)
  (setq major-mode 'emacs-lisp-mode)     ; This is how describe-mode
                                         ;   finds out what to describe.
  (setq mode-name "Emacs-Lisp")          ; This goes into the mode line.
  (lisp-mode-variables nil)              ; This define various variables.
  (run-hooks 'emacs-lisp-mode-hook))     ; This permits the user to use a
                                         ;   hook to customize the mode.
```

How Emacs Chooses a Major Mode

Based on information in the file name or in the file itself, Emacs automatically selects a major mode for the new buffer when a file is visited.

Command: **fundamental-mode**

Fundamental mode is a major mode that is not specialized for anything in particular. Other major modes are defined in effect by comparison with this one--their definitions say what to change, starting from Fundamental mode. The `fundamental-mode` function does *not* run any hooks, so it is not readily customizable.

Command: **normal-mode** *&optional find-file*

This function establishes the proper major mode and local variable bindings for the current buffer. First it calls `set-auto-mode`, then it runs `hack-local-variables` to parse, and bind or evaluate as appropriate, any local variables.

If the find-file argument to `normal-mode` is non-`nil`, `normal-mode` assumes that the `find-file` function is calling it. In this case, it may process a local variables list at the end of the file. The variable `enable-local-variables` controls whether to do so.

If you run `normal-mode` yourself, the argument find-file is normally `nil`. In this case, `normal-mode` unconditionally processes any local variables list. See section 'Local Variables in Files' in The GNU Emacs Manual, for the syntax of the local variables section of a file.

`normal-mode` uses `condition-case` around the call to the major mode function, so errors are caught and reported as a `File mode specification error', followed by the original error message.

User Option: **enable-local-variables**

This variable controls processing of local variables lists in files being visited. A value of `t` means process the local variables lists unconditionally; `nil` means ignore them; anything else means ask the user what to do for each file. The default value is `t`.

User Option: **enable-local-eval**

This variable controls processing of `Eval:' in local variables lists in files being visited. A value of `t` means process them unconditionally; `nil` means ignore them; anything else means ask the user what to do for each file. The default value is `maybe`.

Function: **set-auto-mode**

This function selects the major mode that is appropriate for the current buffer. It may base its decision on the value of the `-*-' line, on the visited file name (using `auto-mode-alist`), or on the value of a local variable). However, this function does not look for the `mode:' local variable near the end of a file; the `hack-local-variables` function does that. See section 'How Major Modes are Chosen' in The GNU Emacs Manual.

User Option: **default-major-mode**

This variable holds the default major mode for new buffers. The standard value is `fundamental-mode`.

If the value of `default-major-mode` is `nil`, Emacs uses the (previously) current buffer's major mode for the major mode of a new buffer. However, if the major mode symbol has a `mode-class` property with value `special`, then it is not used for new buffers; Fundamental mode is used instead. The modes that have this property are those such as Dired and Rmail that are useful only with text that has been specially prepared.

Variable: **initial-major-mode**

The value of this variable determines the major mode of the initial `*scratch*' buffer. The value should be a symbol that is a major mode command name. The default value is `lisp-interaction-mode`.

Variable: **auto-mode-alist**

This variable contains an association list of file name patterns (regular expressions; see section Regular Expressions) and corresponding major mode functions. Usually, the file name patterns test for suffixes, such as `.el' and `.c', but this need not be the case. Each element of the alist looks like `(regexp . mode-function)`.

For example,

```
(("^/tmp/fol/" . text-mode)
 ("\\.texinfo$" . texinfo-mode)
 ("\\.texi$" . texinfo-mode)
```


```
("\\.el$" . emacs-lisp-mode)
("\\.c$" . c-mode)
("\\.h$" . c-mode)
...)
```

When you visit a file whose *expanded* file name (see section Functions that Expand Filenames) matches a regexp, `set-auto-mode` calls the corresponding mode-function. This feature enables Emacs to select the proper major mode for most files.

Here is an example of how to prepend several pattern pairs to `auto-mode-alist`. (You might use this sort of expression in your `.emacs' file.)

```
(setq auto-mode-alist
  (append
   ;; Filename starts with a dot.
   '(("/\\.[^/]*$" . fundamental-mode)
     ;; Filename has no dot.
     ("[^\\./]*$" . fundamental-mode)
     ("\\.C$" . c++-mode))
   auto-mode-alist))
```

Function: **hack-local-variables** *&optional force*

This function parses, and binds or evaluates as appropriate, any local variables for the current buffer.

The handling of `enable-local-variables` documented for `normal-mode` actually takes place here. The argument force reflects the argument find-file given to `normal-mode`.

Getting Help about a Major Mode

The `describe-mode` function is used to provide information about major modes. It is normally called with C-h m. The `describe-mode` function uses the value of `major-mode`, which is why every major mode function needs to set the `major-mode` variable.

Command: **describe-mode**

This function displays the documentation of the current major mode.

The `describe-mode` function calls the `documentation` function using the value of `major-mode` as an argument. Thus, it displays the documentation string of the major mode function. (See section Access to Documentation Strings.)

Variable: **major-mode**

This variable holds the symbol for the current buffer's major mode. This symbol should be the name of the function that is called to initialize the mode. The `describe-mode` function uses the documentation string of this symbol as the documentation of the major mode.

Minor Modes

A minor mode provides features that users may enable or disable independently of the choice of major mode. Minor modes can be enabled individually or in combination. Minor modes would be better named "Generally available, optional feature modes" except that such a name is unwieldy.

A minor mode is not usually a modification of single major mode. For example, Auto Fill mode may be used in any major mode that permits text insertion. To be general, a minor mode must be effectively independent of the things major modes do.

A minor mode is often much more difficult to implement than a major mode. One reason is that you should be able to deactivate a minor mode and restore the environment of the major mode to the state it was in before the minor mode was activated.

Often the biggest problem in implementing a minor mode is finding a way to insert the necessary hook into the rest of Emacs. Minor mode keymaps make this easier.

Conventions for Writing Minor Modes

There are conventions for writing minor modes just as there are for major modes. Several of the major mode conventions apply to minor modes as well: those regarding the name of the mode initialization function, the names of global symbols, and the use of keymaps and other tables.

In addition, there are several conventions that are specific to minor modes.

- Make a variable whose name ends in `-mode' to represent the minor mode. Its value should enable or disable the mode (`nil` to disable; anything else to enable.) We call this the mode variable.

  This variable is used in conjunction with the `minor-mode-alist` to display the minor mode name in the mode line. It can also enable or disable a minor mode keymap. Individual commands or hooks can also check the variable's value.

  If you want the minor mode to be enabled separately in each buffer, make the variable buffer-local.

- Define a command whose name is the same as the mode variable. Its job is to enable and disable the mode by setting the variable.

  The command should accept one optional argument. If the argument is `nil`, it should toggle the mode (turn it on if it is off, and off if it is on). Otherwise, it should turn the mode on if the argument is a positive integer, a symbol other than `nil` or `-`, or a list whose CAR is such an integer or symbol; it should turn the mode off otherwise.

  Here is an example taken from the definition of `overwrite-mode`. It shows the use of `overwrite-mode` as a variable which enables or disables the mode's behavior.

```
(setq overwrite-mode
      (if (null arg) (not overwrite-mode)
        (> (prefix-numeric-value arg) 0)))
```

- Add an element to `minor-mode-alist` for each minor mode (see section Variables Used in the Mode Line). This element should be a list of the following form:


```
(mode-variable string)
```

Here mode-variable is the variable that controls enablement of the minor mode, and string is a short string, starting with a space, to represent the mode in the mode line. These strings must be short so that there is room for several of them at once.

When you add an element to `minor-mode-alist`, use `assq` to check for an existing element, to avoid duplication. For example:

```
(or (assq 'leif-mode minor-mode-alist)
    (setq minor-mode-alist
          (cons '(leif-mode " Leif") minor-mode-alist)))
```

Keymaps and Minor Modes

As of Emacs version 19, each minor mode can have its own keymap which is active when the mode is enabled. See section Active Keymaps. To set up a keymap for a minor mode, add an element to the alist `minor-mode-map-alist`.

One use of minor mode keymaps is to modify the behavior of certain self-inserting characters so that they do something else as well as self-insert. This is the only way to accomplish this in general, since there is no way to customize what `self-insert-command` does except in certain special cases (designed for abbrevs and Auto Fill mode). (Do not try substituting your own definition of `self-insert-command` for the standard one. The editor command loop handles this function specially.)

Variable: **minor-mode-map-alist**

This variable is an alist of elements element that look like this:

```
(variable . keymap)
```

where variable is the variable which indicates whether the minor mode is enabled, and keymap is the keymap. The keymap keymap is active whenever variable has a non-`nil` value.

Note that elements of `minor-mode-map-alist` do not have the same structure as elements of `minor-mode-alist`. The map must be the CDR of the element; a list with the map as the second element will not do.

What's more, the keymap itself must appear in the CDR. It does not work to store a variable in the CDR and make the map the value of that variable.

When more than one minor mode keymap is active, their order of priority is the order of `minor-mode-map-alist`. But you should design minor modes so that they don't interfere with each other. If you do this properly, the order will not matter.

Mode Line Format

Each Emacs window (aside from minibuffer windows) includes a mode line which displays status information about the buffer displayed in the window. The mode line contains information about the buffer such as its name, associated file, depth of recursive editing, and the major and minor modes of the buffer.

This section describes how the contents of the mode line are controlled. It is in the chapter on modes because much of the information displayed in the mode line relates to the enabled major and minor modes.

`mode-line-format` is a buffer-local variable that holds a template used to display the mode line of the current buffer. All windows for the same buffer use the same `mode-line-format` and the mode lines will appear the same (except perhaps for the percentage of the file scrolled off the top).

The mode line of a window is normally updated whenever a different buffer is shown in the window, or when the buffer's modified-status changes from `nil` to `t` or vice-versa. If you modify any of the variables referenced by `mode-line-format`, you may want to force an update of the mode line so as to display the new information.

Function: **force-mode-line-update**

Force redisplay of the current buffer's mode line.

The mode line is usually displayed in inverse video; see `mode-line-inverse-video` in section Inverse Video.

The Data Structure of the Mode Line

The mode line contents are controlled by a data structure of lists, strings, symbols and numbers kept in the buffer-local variable `mode-line-format`. The data structure is called a mode line construct, and it is built in recursive fashion out of simpler mode line constructs.

Variable: **mode-line-format**

The value of this variable is a mode line construct with overall responsibility for the mode line format. The value of this variable controls which other variables are used to form the mode line text, and where they appear.

A mode line construct may be as simple as a fixed string of text, but it usually specifies how to use other variables to construct the text. Many of these variables are themselves defined to have mode line constructs as their values.

The default value of `mode-line-format` incorporates the values of variables such as `mode-name` and `minor-mode-alist`. Because of this, very few modes need to alter `mode-line-format`. For most purposes, it is sufficient to alter the variables referenced by `mode-line-format`.

A mode line construct may be a list, cons cell, symbol, or string. If the value is a list, each element may be a list, a cons cell, a symbol, or a string.

`string`

A string as a mode line construct is displayed verbatim in the mode line except for %-constructs. Decimal digits after the % specify the field width for space filling on the right (i.e., the data is left

justified). See section %-Constructs in the Mode Line.

`symbol`

A symbol as a mode line construct stands for its value. The value of symbol is used in place of symbol unless symbol is `t` or `nil`, or is void, in which case symbol is ignored.

There is one exception: if the value of symbol is a string, it is processed verbatim in that the %-constructs are not recognized.

`(string rest...) or (list rest...)`

A list whose first element is a string or list, means to concatenate all the elements. This is the most common form of mode line construct.

`(symbol then else)`

A list whose first element is a symbol is a conditional. Its meaning depends on the value of symbol. If the value is non-`nil`, the second element of the list (then) is processed recursively as a mode line element. But if the value of symbol is `nil`, the third element of the list (if there is one) is processed recursively.

`(width rest...)`

A list whose first element is an integer specifies truncation or padding of the results of rest. The remaining elements rest are processed recursively as mode line constructs and concatenated together. Then the result is space filled (if width is positive) or truncated (to -width columns, if width is negative) on the right.

For example, the usual way to show what percentage of a buffer is above the top of the window is to use a list like this: `(-3 . "%p")`.

If you do alter `mode-line-format` itself, the new value should use all the same variables that are used by the default value, rather than duplicating their contents or displaying the information in another fashion. This permits customizations made by the user, by libraries (such as `display-time`) or by major modes via changes to those variables remain effective.

Here is an example of a `mode-line-format` that might be useful for `shell-mode` since it contains the hostname and default directory.

```
(setq mode-line-format
  (list ""
   'mode-line-modified
   "%b--"
   (getenv "HOST")      ; One element is not constant.
   ":"
   'default-directory
   "   "
   'global-mode-string
   "   %[(" 'mode-name
   'minor-mode-alist
   "%n"
   'mode-line-process
   ")%]----"
   '(-3 . "%p")
```

```
  "-%-"))
```

Variables Used in the Mode Line

This section describes variables incorporated by the standard value of `mode-line-format` into the text of the mode line. There is nothing inherently special about these variables; any other variables could have the same effects on the mode line if `mode-line-format` were changed to use them.

Variable: **mode-line-modified**

This variable holds the value of the mode-line construct that displays whether the current buffer is modified.

The default value of `mode-line-modified` is `("--%1*%1*-")`. This means that the mode line displays `--**-' if the buffer is modified, `-----' if the buffer is not modified, and `--%%-' if the buffer is read only.

Changing this variable does not force an update of the mode line.

Variable: **mode-line-buffer-identification**

This variable identifies the buffer being displayed in the window. Its default value is `Emacs: %17b', which means that it displays `Emacs:' followed by the buffer name. You may want to change this in modes such as Rmail that do not behave like a "normal" Emacs.

Variable: **global-mode-string**

This variable holds a string that is displayed in the mode line. The command `display-time` puts the time and load in this variable. The `%M' construct substitutes the value of `global-mode-string`, but this is obsolete, since the variable is included directly in the mode line.

Variable: **mode-name**

This buffer-local variable holds the "pretty" name of the current buffer's major mode. Each major mode should set this variable so that the mode name will appear in the mode line.

Variable: **minor-mode-alist**

This variable holds an association list whose elements specify how the mode line should indicate that a minor mode is active. Each element of the `minor-mode-alist` should be a two-element list:

```
(minor-mode-variable mode-line-string)
```

The string mode-line-string is included in the mode line when the value of minor-mode-variable is non-`nil` and not otherwise. These strings should begin with spaces so that they don't run together. Conventionally, the minor-mode-variable for a specific mode is set to a non-`nil` value when that minor mode is activated.

The default value of `minor-mode-alist` is:

```
minor-mode-alist
=> ((abbrev-mode " Abbrev")
    (overwrite-mode " Ovwrt")
    (auto-fill-function " Fill")
    (defining-kbd-macro " Def"))
```

(In earlier Emacs versions, `auto-fill-function` was called `auto-fill-hook`.)

`minor-mode-alist` is not buffer-local. The variables mentioned in the alist should be buffer-local if the minor mode can be enabled separately in each buffer.

Variable: **mode-line-process**

This buffer-local variable contains the mode line information on process status in modes used for communicating with subprocesses. It is displayed immediately following the major mode name, with no intervening space. For example, its value in the `*shell*' buffer is `(": %s")`, which allows the shell to display its status along with the major mode as: `(Shell: run)'. Normally this variable is `nil`.

Variable: **default-mode-line-format**

This variable holds the default `mode-line-format` for buffers that do not override it. This is the same as `(default-value 'mode-line-format)`.

The default value of `default-mode-line-format` is:

```
("" 
 mode-line-modified
 mode-line-buffer-identification
 "   "
 global-mode-string
 "   %[("
 mode-name
 minor-mode-alist
 "%n"
 mode-line-process
 ")%]----"
 (-3 . "%p")
 "-%-")
```

%-Constructs in the Mode Line

The following table lists the recognized %-constructs and what they mean.

%b
: the current buffer name, using the `buffer-name` function.

%f
: the visited file name, using the `buffer-file-name` function.

%*
: `%' if the buffer is read only (see `buffer-read-only`);
`*' if the buffer is modified (see `buffer-modified-p`);
`-' otherwise.

%s
: the status of the subprocess belonging to the current buffer, using `process-status`.

%p

the percent of the buffer above the top of window, or `Top', `Bottom' or `All'.

`%n`

`Narrow' when narrowing is in effect; nothing otherwise (see `narrow-to-region` in section Narrowing).

`%[`

an indication of the depth of recursive editing levels (not counting minibuffer levels): one `[' for each editing level.

`%]`

one `]' for each recursive editing level (not counting minibuffer levels).

`%%`

the character `%'---this is how to include a literal `%' in a string in which %-constructs are allowed.

`%-`

dashes sufficient to fill the remainder of the mode line.

The following two %-constructs are still supported but are obsolete since use of the `mode-name` and `global-mode-string` variables will produce the same results.

`%m`

the value of `mode-name`.

`%M`

the value of `global-mode-string`. Currently, only `display-time` modifies the value of `global-mode-string`.

Hooks

A hook is a variable where you can store a function or functions to be called on a particular occasion by an existing program. Emacs provides lots of hooks for the sake of customization. Most often, hooks are set up in the `.emacs' file, but Lisp programs can set them also. See section Standard Hooks, for a list of standard hook variables.

Most of the hooks in Emacs are normal hooks. These variables contain lists of functions to be called with no arguments. The reason most hooks are normal hooks is so that you can use them in a uniform way. You can always tell when a hook is a normal hook, because its name ends in `-hook'.

The recommended way to add a hook function to a normal hook is by calling `add-hook` (see below). The hook functions may be any of the valid kinds of functions that `funcall` accepts (see section What Is a Function?). Most normal hook variables are initially void; `add-hook` knows how to deal with this.

As for abnormal hooks, those whose names end in `-function' have a value which is a single function. Those whose names end in `-hooks' have a value which is a list of functions. Any hook which is abnormal is abnormal because a normal hook won't do the job; either the functions are called with arguments, or their values are meaningful. The name shows you that the hook is abnormal and you need to look up how to use it properly.

Most major modes run hooks as the last step of initialization. This makes it easy for a user to customize the behavior of the mode, by overriding the local variable assignments already made by the mode. But hooks may

also be used in other contexts. For example, the hook `suspend-hook` runs just before Emacs suspends (see section Suspending Emacs).

For example, you can put the following expression in your `.emacs' file if you want to turn on Auto Fill mode when in Lisp Interaction mode:

```
(add-hook 'lisp-interaction-mode-hook 'turn-on-auto-fill)
```

The next example shows how to use a hook to customize the way Emacs formats C code. (People often have strong personal preferences for one format compared to another.) Here the hook function is an anonymous lambda expression.

```
(add-hook 'c-mode-hook
  (function (lambda ()
              (setq c-indent-level 4
                    c-argdecl-indent 0
                    c-label-offset -4
                    c-continued-statement-indent 0
                    c-brace-offset 0
                    comment-column 40))))

(setq c++-mode-hook c-mode-hook)
```

Finally, here is an example of how to use the Text mode hook to provide a customized mode line for buffers in Text mode, displaying the default directory in addition to the standard components of the mode line. (This may cause the mode line to run out of space if you have very long file names or display the time and load.)

```
(add-hook 'text-mode-hook
  (function (lambda ()
              (setq mode-line-format
                    '(mode-line-modified
                      "Emacs: %14b"
                      "  "
                      default-directory
                      " "
                      global-mode-string
                      "%[("
                      mode-name
                      minor-mode-alist
                      "%n"
                      mode-line-process
                      ") %]---"
                      (-3 . "%p")
                      "-%-"))))))
```

At the appropriate time, Emacs uses the `run-hooks` function to run particular hooks. This function calls the hook functions you have added with `add-hooks`.

Function: **run-hooks** *&rest hookvar*

This function takes one or more hook names as arguments and runs each one in turn. Each hookvar argument should be a symbol that is a hook variable. These arguments are processed in the order specified.

If a hook variable has a non-`nil` value, that value may be a function or a list of functions. If the value is a function (either a lambda expression or a symbol with a function definition), it is called. If it is a list, the elements are called, in order. The hook functions are called with no arguments.

For example:

```
(run-hooks 'emacs-lisp-mode-hook)
```

Major mode functions use this function to call any hooks defined by the user.

Function: **add-hook** *hook function &optional append*

This function is the handy way to add function function to hook variable hook. For example,

```
(add-hook 'text-mode-hook 'my-text-hook-function)
```

adds `my-text-hook-function` to the hook called `text-mode-hook`.

It is best to design your hook functions so that the order in which they are executed does not matter. Any dependence on the order is "asking for trouble." However, the order is predictable: normally, function goes at the front of the hook list, so it will be executed first (barring another `add-hook` call).

If the optional argument append is non-`nil`, the new hook function goes at the end of the hook list and will be executed last.

Go to the previous, next section.

Go to the previous, next section.

Documentation

GNU Emacs Lisp has convenient on-line help facilities, most of which derive their information from the documentation strings associated with functions and variables. This chapter describes how to write good documentation strings for your Lisp programs, as well as how to write programs to access documentation.

Note that the documentation strings for Emacs are not the same thing as the Emacs manual. Manuals have their own source files, written in the Texinfo language; documentation strings are specified in the definitions of the functions and variables they apply to. A collection of documentation strings is not sufficient as a manual because a good manual is not organized in that fashion; it is organized in terms of topics of discussion.

Documentation Basics

A documentation string is written using the Lisp syntax for strings, with double-quote characters surrounding the text of the string. This is because it really is a Lisp string object. The string serves as documentation when it is written in the proper place in the definition of a function or variable. In a function definition, the documentation string follows the argument list. In a variable definition, the documentation string follows the initial value of the variable.

When you write a documentation string, make the first line a complete sentence (or two complete sentences) since some commands, such as `apropos`, print only the first line of a multi-line documentation string. Also, you should not indent the second line of a documentation string, if you have one, because that looks odd when you use C-h f (`describe-function`) or C-h v (`describe-variable`).

Documentation strings may contain several special substrings, which stand for key bindings to be looked up in the current keymaps when the documentation is displayed. This allows documentation strings to refer to the keys for related commands and be accurate even when a user rearranges the key bindings. (See section Access to Documentation Strings.)

Within the Lisp world, a documentation string is kept with the function or variable that it describes:

- The documentation for a function is stored in the function definition itself (see section Lambda Expressions). The function `documentation` knows how to extract it.
- The documentation for a variable is stored on the variable's property list under the property name `variable-documentation`. The function `documentation-property` knows how to extract it.

However, to save space, the documentation for preloaded functions and variables (including primitive functions and autoloaded functions) are stored in the `` `emacs/etc/DOC-version' `` file. Both the `documentation` and the `documentation-property` functions know how to access `` `emacs/etc/DOC-version' ``, and the process is transparent to the user. In this case, the documentation string is replaced with an integer offset into the `` `emacs/etc/DOC-version' `` file. Keeping the documentation strings out of the Emacs core image saves a significant amount of space. See section Building Emacs.

For information on the uses of documentation strings, see section 'Help' in The GNU Emacs Manual.

The `emacs/etc' directory contains two utilities that you can use to print nice-looking hardcopy for th file `emacs/etc/DOC-version'. These are `sorted-doc.c' and `digest-doc.c'.

Access to Documentation Strings

Function: **documentation-property** *symbol property &optional verbatim*

This function returns the documentation string that is recorded symbol's property list under property property. This uses the function `get`, but does more than that: it also retrieves the string from the file `emacs/etc/DOC-version' if necessary, and runs `substitute-command-keys` to substitute the actual (current) key bindings.

If verbatim is non-`nil`, that inhibits running `substitute-command-keys`. (The verbatim argument exists only as of Emacs 19.)

```
(documentation-property 'command-line-processed
   'variable-documentation)
     => "t once command line has been processed"
(symbol-plist 'command-line-processed)
     => (variable-documentation 188902)
```

Function: **documentation** *function &optional verbatim*

This function returns the documentation string of function. If the documentation string is stored in the `emacs/etc/DOC-version' file, this function will access it there.

In addition, `documentation` runs `substitute-command-keys` on the resulting string, so the value contains the actual (current) key bindings. (This is not done if verbatim is non-`nil`; the verbatim argument exists only as of Emacs 19.)

The function `documentation` signals a `void-function` error unless function has a function definition. However, function does not need to have a documentation string. If there is no documentation string, `documentation` returns `nil`.

Here is an example of using the two functions, `documentation` and `documentation-property`, to display the documentation strings for several symbols in a `*Help*' buffer.

```
(defun describe-symbols (pattern)
  "Describe the Emacs Lisp symbols matching PATTERN.
All symbols that have PATTERN in their name are described
in the `*Help*' buffer."
  (interactive "sDescribe symbols matching: ")
  (let ((describe-func
         (function
          (lambda (s)
            ;; Print description of symbol.
            (if (fboundp s)             ; It is a function.
                (princ
```


```
                (format "%s\t%s\n%s\n\n" s
                  (if (commandp s)
                      (let ((keys (where-is-internal s)))
                        (if keys
                            (concat
                             "Keys: "
                             (mapconcat 'key-description
                                        keys " "))
                          "Keys: none"))
                    "Function")
                  (or (documentation s)
                      "not documented"))))

          (if (boundp s)             ; It is a variable.
              (princ
               (format "%s\t%s\n%s\n\n" s
                 (if (user-variable-p s)
                     "Option " "Variable")
                 (or (documentation-property
                       s 'variable-documentation)
                     "not documented")))))))
      sym-list)

    ;; Build a list of symbols that match pattern.
    (mapatoms (function
               (lambda (sym)
                 (if (string-match pattern (symbol-name sym))
                     (setq sym-list (cons sym sym-list))))))

    ;; Display the data.
    (with-output-to-temp-buffer "*Help*"
      (mapcar describe-func (sort sym-list 'string<))
      (print-help-return-message))))
```

The `describe-symbols` function works like `apropos`, but provides more information.

```
(describe-symbols "goal")

---------- Buffer: *Help* ----------
goal-column     Option
*Semipermanent goal column for vertical motion, as set by C-x C-n, or nil.

set-goal-column Command: C-x C-n
Set the current horizontal position as a goal for C-n and C-p.
Those commands will move to this position in the line moved to
rather than trying to keep the same horizontal position.
With a non-nil argument, clears out the goal column
so that C-n and C-p resume vertical motion.
```


```
The goal column is stored in the variable `goal-column'.

temporary-goal-column   Variable
Current goal column for vertical motion.
It is the column where point was
at the start of current run of vertical motion commands.
When the `track-eol' feature is doing its job, the value is 9999.
---------- Buffer: *Help* ----------
```

Function: **Snarf-documentation** *filename*

This function is used only during Emacs initialization, just before the runnable Emacs is dumped. It finds the file offsets of the documentation strings stored in the file filename, and records them in the in-core function definitions and variable property lists in place of the actual strings. See section Building Emacs.

Emacs finds the file filename in the `emacs/etc' directory. When the dumped Emacs is later executed, the same file is found in the directory `data-directory`. Usually filename is `"DOC-version"`.

Variable: **data-directory**

This variable holds the name of the directory in which Emacs finds certain data files that come with Emacs or are built as part of building Emacs. (In older Emacs versions, this directory was the same as `exec-directory`.)

Substituting Key Bindings in Documentation

This function makes it possible for you to write a documentation string that enables a user to display information about the current, actual key bindings. if you call `documentation` with non-`nil` verbatim, you might later call this function to do the substitution that you prevented `documentation` from doing.

Function: **substitute-command-keys** *string*

This function returns string with certain special substrings replaced by the actual (current) key bindings. This permits the documentation to be displayed with accurate information about key bindings. (The key bindings may be changed by the user between the time Emacs is built and the time that the documentation is asked for.)

This table lists the forms of the special substrings and what they are replaced with:

`\[command]`
: is replaced either by a keystroke sequence that will invoke command, or by `M-x command' if command is not bound to any key sequence.

`\{mapvar}`
: is replaced by a summary of the value of mapvar, taken as a keymap. (The summary is made by `describe-bindings`.)

`\<mapvar>`
: makes this call to `substitute-command-keys` use the value of mapvar as the keymap for future `\[command]' substrings. This special string does not produce any replacement text itself; it only affects the replacements done later.

Please note: each `\' must be doubled when written in a string in Emacs Lisp.

Here are examples of the special substrings:

```
(substitute-command-keys
   "To abort recursive edit, type: \\[abort-recursive-edit]")

=> "To abort recursive edit, type: C-]"

(substitute-command-keys
   "The keys that are defined for the minibuffer here are:
  \\{minibuffer-local-must-match-map}")

=> "The keys that are defined for the minibuffer here are:

?               minibuffer-completion-help
SPC             minibuffer-complete-word
TAB             minibuffer-complete
LFD             minibuffer-complete-and-exit
RET             minibuffer-complete-and-exit
C-g             abort-recursive-edit
"

(substitute-command-keys
   "To abort a recursive edit from the minibuffer, type\
\\<minibuffer-local-must-match-map>\\[abort-recursive-edit].")
=> "To abort a recursive edit from the minibuffer, type C-g."
```

Describing Characters for Help Messages

These functions convert events, key sequences or characters to textual descriptions. These descriptions are useful for including arbitrary text characters or key sequences in messages, because they convert non-printing characters to sequences of printing characters. The description of a printing character is the character itself.

Function: **key-description** *sequence*

This function returns a string containing the Emacs standard notation for the input events in sequence. The argument sequence may be a string, vector or list. See section Input Events, for more information about valid events. See also the examples for `single-key-description`, below.

Function: **single-key-description** *event*

This function returns a string describing event in the standard Emacs notation for keyboard input. A normal printing character is represented by itself, but a control character turns into a string starting with `C-', a meta character turns into a string starting with `M-', and space, linefeed, etc. are transformed to `SPC', `LFD', etc. A function key is represented by its name. An event which is a list is represented by the name of the symbol in the CAR of the list.

```
(single-key-description ?\C-x)
```


```
     => "C-x"
(key-description "\C-x \M-y \n \t \r \f123")
     => "C-x SPC M-y SPC LFD SPC TAB SPC RET SPC C-l 1 2 3"
(single-key-description 'C-mouse-1)
     => "C-mouse-1"
```

Function: **text-char-description** *character*

This function returns a string describing character in the standard Emacs notation for characters that appear in text--like `single-key-description`, except that control characters are represented with a leading caret (which is how control characters in Emacs buffers are usually displayed).

```
(text-char-description ?\C-c)
     => "^C"
(text-char-description ?\M-m)
     => "M-m"
(text-char-description ?\C-\M-m)
     => "M-^M"
```

Help Functions

Emacs provides a variety of on-line help functions, all accessible to the user as subcommands of the prefix C-h. For more information about them, see section 'Help' in The GNU Emacs Manual. Here we describe some program-level interfaces to the same information.

Command: **apropos** *regexp &optional do-all predicate*

This function finds all symbols whose names contain a match for the regular expression regexp, and returns a list of them. It also displays the symbols in a buffer named `*Help*', each with a one-line description.

If do-all is non-`nil`, then `apropos` also shows key bindings for the functions that are found.

If predicate is non-`nil`, it should be a function to be called on each symbol that has matched regexp. Only symbols for which predicate returns a non-`nil` value are listed or displayed.

In the first of the following examples, `apropos` finds all the symbols with names containing `exec'. In the second example, it finds and returns only those symbols that are also commands. (We don't show the output that results in the `*Help*' buffer.)

```
(apropos "exec")
     => (Buffer-menu-execute command-execute exec-directory
    exec-path execute-extended-command execute-kbd-macro
    executing-kbd-macro executing-macro)

(apropos "exec" nil 'commandp)
     => (Buffer-menu-execute execute-extended-command)
```

The command C-h a (`command-apropos`) calls `apropos`, but specifies a predicate to restrict the output to symbols that are commands. The call to `apropos` looks like this:


```
(apropos string t 'commandp)
```

Command: **super-apropos** *regexp &optional do-all*

This function differs from `apropos` in that it searches documentation strings as well as symbol names for matches for regexp. By default, it searches only the documentation strings, and only those of functions and variables that are included in Emacs when it is dumped. If do-all is non-`nil`, it scans the names and documentation strings of all functions and variables.

Command: **help-command**

This command is not a function, but rather a symbol which is equivalent to the keymap called `help-map`. It is defined in `` `help.el' `` as follows:

```
(define-key global-map "\C-h" 'help-command)
(fset 'help-command help-map)
```

Variable: **help-map**

The value of this variable is a local keymap for characters following the Help key, C-h.

Function: **print-help-return-message** *&optional function*

This function builds a string which is a message explaining how to restore the previous state of the windows after a help command. After building the message, it applies function to it if function is non-`nil`. Otherwise it calls `message` to display it in the echo area.

This function expects to be called inside a `with-output-to-temp-buffer` special form, and expects `standard-output` to have the value bound by that special form. For an example of its use, see the example in the section describing the `documentation` function (see section Access to Documentation Strings).

The constructed message will have one of the forms shown below.

```
---------- Echo Area ----------
Type C-x 1 to remove help window.
---------- Echo Area ----------

---------- Echo Area ----------
Type C-x 4 b RET to restore old contents of help window.
---------- Echo Area ----------
```

Variable: **help-char**

The value of this variable is the character that Emacs recognizes as meaning Help. When Emacs reads this character (which is usually 8, the value of C-h), Emacs evaluates `(eval help-form)`, and displays the result if it is a string. If `help-form`'s value is `nil`, this character is read normally.

Variable: **help-form**

The value of this variable is a form to execute when the character `help-char` is read. If the form returns a

string, that string is displayed. If `help-form` is `nil`, then the help character is not recognized.

Entry to the minibuffer binds this variable to the value of `minibuffer-help-form`.

The following two functions are found in the library `` `helper' ``. They are for modes that want to provide help without relinquishing control, such as the "electric" modes. You must load that library with `(require 'helper)` in order to use them. Their names begin with `Helper' to distinguish them from the ordinary help functions.

Command: **Helper-describe-bindings**

This command pops up a window displaying a help buffer containing a listing of all of the key bindings from both the local and global keymaps. It works by calling `describe-bindings`.

Command: **Helper-help**

This command provides help for the current mode. It prompts the user in the minibuffer with the message `Help (Type ? for further options)', and then provides assistance in finding out what the key bindings are, and what the mode is intended for. It returns `nil`.

This can be customized by changing the map `Helper-help-map`.

Go to the previous, next section.

Go to the previous, next section.

Files

In Emacs, you can find, create, view, save, and otherwise work with files and file directories. This chapter describes most of the file-related functions of Emacs Lisp, but a few others are described in section Buffers, and those related to backups and auto-saving are described in section Backups and Auto-Saving.

Visiting Files

Visiting a file means reading a file into a buffer. Once this is done, we say that the buffer is visiting that file, and call the file "the visited file" of the buffer.

A file and a buffer are two different things. A file is information recorded permanently in the computer (unless you delete it). A buffer, on the other hand, is information inside of Emacs that will vanish at the end of the editing session (or when you kill the buffer). Usually, a buffer contains information that you have copied from a file; then we say the buffer is visiting that file. The copy in the buffer is what you modify with editing commands. Such changes to the buffer do not change the file; therefore, to make the changes permanent, you must save the buffer, which means copying the altered buffer contents back into the file.

In spite of the distinction between files and buffers, people often refer to a file when they mean a buffer and vice-versa. Indeed, we say, "I am editing a file," rather than, "I am editing a buffer which I will soon save as a file of the same name." Humans do not usually need to make the distinction explicit. When dealing with a computer program, however, it is good to keep the distinction in mind.

Functions for Visiting Files

This section describes the functions normally used to visit files. For historical reasons, these functions have names starting with `find-' rather than `visit-'. See section Buffer File Name, for functions and variables that access the visited file name of a buffer or that find an existing buffer by its visited file name.

Command: **find-file** *filename*

This function reads the file filename into a buffer and displays that buffer in the selected window so that the user can edit it.

The body of the `find-file` function is very simple and looks like this:

```
(switch-to-buffer (find-file-noselect filename))
```

(See `switch-to-buffer` in section Displaying Buffers in Windows.)

When `find-file` is called interactively, it prompts for filename in the minibuffer.

Function: **find-file-noselect** *filename*

This function is the guts of all the file-visiting functions. It reads a file into a buffer and returns the buffer. You may then make the buffer current or display it in a window if you wish, but this function does not do so.

If no buffer is currently visiting filename, then one is created and the file is visited. If filename does not exist, the buffer is left empty, and `find-file-noselect` displays the message `New file' in the echo area.

If a buffer is already visiting filename, then the `find-file-noselect` function uses that buffer rather than creating a new one. However, it does verify that the file has not changed since it was last visited or saved in that buffer. If the file has changed, then this function asks the user whether to reread the changed file. If the user says `yes', any changes previously made in the buffer are lost.

The `find-file-noselect` function calls `after-find-file` after the file is read in (see section Subroutines of Visiting). The `after-find-file` function sets the buffer major mode, parses local variables, warns the user if there exists an auto-save file more recent than the file just visited, and finishes by running the functions in `find-file-hooks`.

The `find-file-noselect` function returns the buffer that is visiting the file filename.

```
(find-file-noselect "/etc/fstab")
     => #<buffer fstab>
```

Command: **find-alternate-file** *filename*

This function reads the file filename into a buffer and selects it, killing the buffer current at the time the command is run. It is useful if you have visited the wrong file by mistake, so that you can get rid of the buffer that you did not want to create, at the same time as you visit the file you intended.

When this function is called interactively, it prompts for filename.

Command: **find-file-other-window** *filename*

This function visits the file filename and displays its buffer in a window other than the selected window. It may use another existing window or split a window; see section Displaying Buffers in Windows.

When this function is called interactively, it prompts for filename.

Command: **find-file-read-only** *filename*

This function visits the file named filename and selects its buffer, just like `find-file`, but it marks the buffer as read-only. See section Read-Only Buffers, for related functions and variables.

When this function is called interactively, it prompts for filename.

Command: **view-file** *filename*

This function views filename in View mode, returning to the previous buffer when done. View mode is a mode that allows you to skim rapidly through the file but does not let you modify it.

After loading the file, `view-file` runs the normal hook `view-hook` using `run-hooks`. See section Hooks.

When this function is called interactively, it prompts for filename.

Variable: **find-file-hooks**

The value of this variable is a list of functions to be called after a file is visited. The file's local-variables specification (if any) will have been processed before the hooks are run. The buffer visiting the file is current when the hook functions are run.

This variable could be a normal hook, but we think that renaming it would not be advisable.

Variable: **find-file-not-found-hooks**

The value of this variable is a list of functions to be called when `find-file` or `find-file-noselect` is passed a nonexistent filename. These functions are called as soon as the error is detected. `buffer-file-name` is already set up. The functions are called in the order given, until one of them returns non-`nil`.

This is not a normal hook because the values of the functions are used and they may not all be run.

Subroutines of Visiting

The `find-file-noselect` function uses the `create-file-buffer` and `after-find-file` functions as subroutines. Sometimes it is useful to call them directly.

Function: **create-file-buffer** *filename*

This function creates a suitably named buffer for visiting filename, and returns it. The string filename (sans directory) is used unchanged if that name is free; otherwise, a string such as `<2>' is appended to get an unused name. See also section Creating Buffers.

Please note: `create-file-buffer` does *not* associate the new buffer with a file and does not make it the current buffer.

```
(create-file-buffer "foo")
     => #<buffer foo>
(create-file-buffer "foo")
     => #<buffer foo<2>>
(create-file-buffer "foo")
     => #<buffer foo<3>>
```

This function is used by `find-file-noselect`. It uses `generate-new-buffer` (see section Creating Buffers).

Function: **after-find-file** *&optional error warn*

This function is called by `find-file-noselect` and by the default revert function (see section Reverting). It sets the buffer major mode, and parses local variables (see section How Emacs Chooses a Major Mode).

If there was an error in opening the file, the calling function should pass error a non-`nil` value. In that case, `after-find-file` issues a warning: `(New File)'. Note that, for serious errors, you would not even call `after-find-file`. Only "file not found" errors get here with a non-`nil` error.

If warn is non-`nil`, then this function issues a warning if an auto-save file exists and is more recent than the visited file.

The last thing `after-find-file` does is call all the functions in `find-file-hooks`.

Saving Buffers

When you edit a file in Emacs, you are actually working on a buffer that is visiting that file--that is, the contents of the file are copied into the buffer and the copy is what you edit. Changes to the buffer do not change the file until you save the buffer, which means copying the contents of the buffer into the file.

Command: **save-buffer** *&optional backup-option*

This function saves the contents of the current buffer in its visited file if the buffer has been modified since it was last visited or saved. Otherwise it does nothing.

`save-buffer` is responsible for making backup files. Normally, backup-option is `nil`, and `save-buffer` makes a backup file only if this is the first save or if the buffer was previously modified. Other values for backup-option request the making of backup files in other circumstances:

- With an argument of 4 or 64, reflecting 1 or 3 C-u's, the `save-buffer` function marks this version of the file to be backed up when the buffer is next saved.
- With an argument of 16 or 64, reflecting 2 or 3 C-u's, the `save-buffer` function unconditionally backs up the previous version of the file before saving it.

Command: **save-some-buffers** *&optional save-silently-p exiting*

This command saves some modified file-visiting buffers. Normally it asks the user about each buffer. But if save-silently-p is non-`nil`, it saves all the file-visiting buffers without querying the user.

The optional exiting argument, if non-`nil`, requests this function to offer also to save certain other buffers that are not visiting files. These are buffers that have a non-`nil` local value of `buffer-offer-save`. (A user who says yes to saving one of these is asked to specify a file name to use.) The `save-buffers-kill-emacs` function passes a non-`nil` value for this argument.

Variable: **buffer-offer-save**

When this variable is non-`nil` in a buffer, Emacs offers to save the buffer on exit even if the buffer is not visiting a file. The variable is automatically local in all buffers. Normally, Mail mode (used for editing outgoing mail) sets this to `t`.

Command: **write-file** *filename*

This function writes the current buffer into file filename, makes the buffer visit that file, and marks it not modified. The buffer is renamed to correspond to filename unless that name is already in use.

Variable: **write-file-hooks**

The value of this variable is a list of functions to be called before writing out a buffer to its visited file. If one of them returns non-`nil`, the file is considered already written and the rest of the functions are not called, nor is the usual code for writing the file executed.

If a function in `write-file-hooks` returns non-`nil`, it is responsible for making a backup file (if that is appropriate). To do so, execute the following code:

```
(or buffer-backed-up (backup-buffer))
```

You might wish to save the file modes value returned by `backup-buffer` and use that to set the mode bits of the file that you write. This is what `basic-save-buffer` does when it writes a file in the usual way.

Here is an example showing how to add an element to `write-file-hooks` but avoid adding it twice:

```
(or (memq 'my-write-file-hook write-file-hooks)
    (setq write-file-hooks
          (cons
          'my-write-file-hook write-file-hooks)))
```

Variable: **local-write-file-hooks**

This works just like `write-file-hooks`, but it is intended to be made local to particular buffers. It's not a good idea to make `write-file-hooks` local to a buffer--use this variable instead.

The variable is marked as a permanent local, so that changing the major mode does not alter a buffer-local value. This is convenient for packages that read "file" contents in special ways, and set up hooks to save the data in a corresponding way.

Variable: **write-contents-hooks**

This works just like `write-file-hooks`, but it is intended to be used for hooks that pertain to the contents of the file, as opposed to hooks that pertain to where the file came from.

Variable: **after-save-hook**

This normal hook runs after a buffer has been saved in its visited file.

Variable: **file-precious-flag**

If this variable is non-`nil`, then `save-buffer` protects against I/O errors while saving by writing the new file to a temporary name instead of the name it is supposed to have, and then renaming it to the intended name after it is clear there are no errors. This procedure prevents problems such as a lack of disk space from resulting in an invalid file.

(This feature worked differently in older Emacs versions.)

Some modes set this non-`nil` locally in particular buffers.

User Option: **require-final-newline**

This variable determines whether files may be written out that do *not* end with a newline. If the value of the variable is `t`, then Emacs silently puts a newline at the end of the file whenever the buffer being saved does not already end in one. If the value of the variable is non-`nil`, but not `t`, then Emacs asks the user whether to add a newline each time the case arises.

If the value of the variable is `nil`, then Emacs doesn't add newlines at all. `nil` is the default value, but a few major modes set it to `t` in particular buffers.

Reading from Files

You can copy a file from the disk and insert it into a buffer using the `insert-file-contents` function. Don't use the user-level command `insert-file` in a Lisp program, as that sets the mark.

Function: **insert-file-contents** *filename &optional visit*

This function inserts the contents of file filename into the current buffer after point. It returns a list of the absolute file name and the length of the data inserted. An error is signaled if filename is not the name of a file that can be read.

If visit is non-`nil`, it also marks the buffer as unmodified and sets up various fields in the buffer so that it is visiting the file filename: these include the buffer's visited file name and its last save file modtime. This feature is used by `find-file-noselect` and you should probably not use it yourself.

If you want to pass a file name to another process so that another program can read the file, see the function `file-local-copy` in section Making Certain File Names "Magic".

Writing to Files

You can write the contents of a buffer, or part of a buffer, directly to a file on disk using the `append-to-file` and `write-region` functions. Don't use these functions to write to files that are being visited; that could cause confusion in the mechanisms for visiting.

Command: **append-to-file** *start end filename*

This function appends the contents of the region delimited by start and end in the current buffer to the end of file filename. If that file does not exist, it is created. This function returns `nil`.

An error is signaled if filename specifies a nonwritable file, or a nonexistent file in a directory where files cannot be created.

Command: **write-region** *start end filename &optional append visit*

This function writes the region (of the current buffer) delimited by start and end into the file specified by filename.

If start is a string, then `write-region` writes or appends that string, rather than text from the buffer.

If append is non-`nil`, then the region is appended to the existing file contents (if any).

If visit is `t`, then Emacs establishes an association between the buffer and the file: the buffer is then visiting that file. It also sets the last file modification time for the current buffer to filename's modtime, and marks the buffer as not modified. This feature is used by `write-file` and you should probably not use it yourself.

If visit is a string, it specifies the file name to visit. This way, you can write the data to one file (filename) while recording the buffer as visiting another file (visit). The argument visit is used in the echo area message and also for file locking; visit is stored in `buffer-file-name`. This feature is used to implement `file-precious-flag`; don't use it yourself unless you really know what you're doing.

Normally, `write-region` displays a message `Wrote file filename' in the echo area. If visit is neither `t` nor `nil` nor a string, then this message is inhibited. This feature is useful for programs that use files for internal purposes, files which the user does not need to know about.

File Locks

When two users edit the same file at the same time, they are likely to interfere with each other. Emacs tries to prevent this situation from arising by recording a file lock when a file is being modified. Emacs can then detect the first attempt to modify a buffer visiting a file that is locked by another Emacs job, and ask the user what to do.

File locks do not work properly when multiple machines can share file systems, such as with NFS. Perhaps a better file locking system will be implemented in the future. When file locks do not work, it is possible for two users to make changes simultaneously, but Emacs can still warn the user who saves second. Also, the detection of modification of a buffer visiting a file changed on disk catches some cases of simultaneous editing; see section Comparison of Modification Time.

Function: **file-locked-p** *filename*

This function returns `nil` if the file filename is not locked by this Emacs process. It returns `t` if it is locked by this Emacs, and it returns the name of the user who has locked it if it is locked by someone else.

```
(file-locked-p "foo")
     => nil
```

Function: **lock-buffer** *&optional filename*

This function locks the file filename, if the current buffer is modified. The argument filename defaults to the current buffer's visited file. Nothing is done if the current buffer is not visiting a file, or is not

modified.

Function: **unlock-buffer**

This function unlocks the file being visited in the current buffer, if the buffer is modified. If the buffer is not modified, then the file should not be locked, so this function does nothing. It also does nothing if the current buffer is not visiting a file.

Function: **ask-user-about-lock** *file other-user*

This function is called when the user tries to modify file, but it is locked by another user name other-user. The value it returns tells Emacs what to do next:

- A value of `t` tells Emacs to grab the lock on the file. Then this user may edit the file and other-user loses the lock.
- A value of `nil` tells Emacs to ignore the lock and let this user edit the file anyway.
- This function may instead signal a `file-locked` error, in which case the change to the buffer which the user was about to make does not take place.

  The error message for this error looks like this:

  ```
  error--> File is locked: file other-user
  ```

  where `file` is the name of the file and other-user is the name of the user who has locked the file.

The default definition of this function asks the user to choose what to do. If you wish, you can replace the `ask-user-about-lock` function with your own version that decides in another way. The code for its usual definition is in `` `userlock.el' ``.

Information about Files

The functions described in this section are similar in as much as they all operate on strings which are interpreted as file names. All have names that begin with the word `file'. These functions all return information about actual files or directories, so their arguments must all exist as actual files or directories unless otherwise noted.

Most of the file-oriented functions take a single argument, filename, which must be a string. The file name is expanded using `expand-file-name`, so `` `~' `` is handled correctly, as are relative file names (including `../'). Environment variable substitutions, such as `$HOME', are not recognized by these functions. See section Functions that Expand Filenames.

Testing Accessibility

These functions test for permission to access a file in specific ways.

Function: **file-exists-p** *filename*

This function returns `t` if a file named filename appears to exist. This does not mean you can necessarily

read the file, only that you can find out its attributes. (On Unix, this is true if the file exists and you have execute permission on the containing directories, regardless of the protection of the file itself.)

If the file does not exist, or if fascist access control policies prevent you from finding the attributes of the file, this function returns `nil`.

Function: **file-readable-p** *filename*

This function returns `t` if a file named filename exists and you can read it. It returns `nil` otherwise.

```
(file-readable-p "files.texi")
     => t
(file-exists-p "/usr/spool/mqueue")
     => t
(file-readable-p "/usr/spool/mqueue")
     => nil
```

Function: **file-executable-p** *filename*

This function returns `t` if a file named filename exists and you can execute it. It returns `nil` otherwise. If the file is a directory, execute permission means you can access files inside the directory.

Function: **file-writable-p** *filename*

This function returns `t` if filename can be written or created by you. It is writable if the file exists and you can write it. It is creatable if the file does not exist, but the specified directory does exist and you can write in that directory. `file-writable-p` returns `nil` otherwise.

In the third example below, `` `foo' `` is not writable because the parent directory does not exist, even though the user could create it.

```
(file-writable-p "~rms/foo")
     => t
(file-writable-p "/foo")
     => nil
(file-writable-p "~rms/no-such-dir/foo")
     => nil
```

Function: **file-accessible-directory-p** *dirname*

This function returns `t` if you have permission to open existing files in directory dirname; otherwise (and if there is no such directory), it returns `nil`. The value of dirname may be either a directory name or the file name of a directory.

Example: after the following,

```
(file-accessible-directory-p "/foo")
     => nil
```

we can deduce that any attempt to read a file in `/foo/' will give an error.

Function: **file-newer-than-file-p** *filename1 filename2*

This functions returns `t` if the file filename1 is newer than file filename2. If filename1 does not exist, it returns `nil`. If filename2 does not exist, it returns `t`.

You can use `file-attributes` to get a file's last modification time as a list of two numbers. See section Other Information about Files.

In the following example, assume that the file `aug-19' was written on the 19th, and `aug-20' was written on the 20th. The file `no-file' doesn't exist at all.

```
(file-newer-than-file-p "aug-19" "aug-20")
     => nil
(file-newer-than-file-p "aug-20" "aug-19")
     => t
(file-newer-than-file-p "aug-19" "no-file")
     => t
(file-newer-than-file-p "no-file" "aug-19")
     => nil
```

Distinguishing Kinds of Files

This section describes how to distinguish directories and symbolic links from ordinary files.

Function: **file-symlink-p** *filename*

If filename is a symbolic link, the `file-symlink-p` function returns the file name to which it is linked. This may be the name of a text file, a directory, or even another symbolic link, or of no file at all.

If filename is not a symbolic link (or there is no such file), `file-symlink-p` returns `nil`.

```
(file-symlink-p "foo")
     => nil
(file-symlink-p "sym-link")
     => "foo"
(file-symlink-p "sym-link2")
     => "sym-link"
(file-symlink-p "/bin")
     => "/pub/bin"
```

Function: **file-directory-p** *filename*

This function returns `t` if filename is the name of an existing directory, `nil` otherwise.

```
(file-directory-p "~rms")
     => t
```


```
(file-directory-p "~rms/lewis/files.texi")
     => nil
(file-directory-p "~rms/lewis/no-such-file")
     => nil
(file-directory-p "$HOME")
     => nil
(file-directory-p
 (substitute-in-file-name "$HOME"))
     => t
```

Truenames

The truename of a file is the name that you get by following symbolic links until none remain, then expanding to get rid of `.' and `..' as components. Strictly speaking, a file need not have a unique truename; the number of distinct truenames a file has is equal to the number of hard links to the file. However, truenames are useful because they eliminate symbolic links as a cause of name variation.

Function: **file-truename** *filename*

The function `file-truename` returns the true name of the file filename. This is the name that you get by following symbolic links until none remain. The argument must be an absolute file name.

See section Buffer File Name, for related information.

Other Information about Files

This section describes the functions for getting detailed information about a file, other than its contents. This information includes the mode bits that control access permission, the owner and group numbers, the number of names, the inode number, the size, and the times of access and modification.

Function: **file-modes** *filename*

This function returns the mode bits of filename, as an integer. The mode bits are also called the file permissions, and they specify access control in the usual Unix fashion. If the low-order bit is 1, then the file is executable by all users, if the second lowest-order bit is 1, then the file is writable by all users, etc.

The highest value returnable is 4095 (7777 octal), meaning that everyone has read, write, and execute permission, that the SUID bit is set for both others and group, and that the sticky bit is set.

```
(file-modes "~/junk/diffs")
     => 492               ; Decimal integer.
(format "%o" 492)
     => 754               ; Convert to octal.

(set-file-modes "~/junk/diffs" 438)
     => nil
```


```
(format "%o" 438)
     => 666               ; Convert to octal.

% ls -l diffs
  -rw-rw-rw-  1 lewis 0 3063 Oct 30 16:00 diffs
```

Function: **file-nlinks** *filename*

This functions returns the number of names (i.e., hard links) that file filename has. If the file does not exist, then this function returns `nil`. Note that symbolic links have no effect on this function, because they are not considered to be names of the files they link to.

```
% ls -l foo*
-rw-rw-rw-  2 rms       4 Aug 19 01:27 foo
-rw-rw-rw-  2 rms       4 Aug 19 01:27 foo1

(file-nlinks "foo")
     => 2
(file-nlinks "doesnt-exist")
     => nil
```

Function: **file-attributes** *filename*

This function returns a list of attributes of file filename. If the specified file cannot be opened, it returns `nil`.

The elements of the list, in order, are:

1. `t` for a directory, a string for a symbolic link (the name linked to), or `nil` for a text file.
2. The number of names the file has. Alternate names, also known as hard links, can be created by using the `add-name-to-file` function (see section Changing File Names and Attributes).
3. The file's UID.
4. The file's GID.
5. The time of last access, as a list of two integers. The first integer has the high-order 16 bits of time, the second has the low 16 bits. (This is similar to the value of `current-time`; see section Time of Day.)
6. The time of last modification as a list of two integers (as above).
7. The time of last status change as a list of two integers (as above).
8. The size of the file in bytes.
9. The file's modes, as a string of ten letters or dashes as in `ls -l'.
10. `t` if the file's GID would change if file were deleted and recreated; `nil` otherwise.
11. The file's inode number.
12. The file system number of the file system that the file is in. This element together with the file's inode number, give enough information to distinguish any two files on the system--no two files can have the same values for both of these numbers.

For example, here are the file attributes for `files.texi':

```
(file-attributes "files.texi")
     =>  (nil
          1
          2235
          75
          (8489 20284)
          (8489 20284)
          (8489 20285)
          14906
          "-rw-rw-rw-"
          nil
          129500
          -32252)
```

and here is how the result is interpreted:

`nil`
: is neither a directory nor a symbolic link.

`1`
: has only one name (the name `files.texi' in the current default directory).

`2235`
: is owned by the user with UID 2235.

`75`
: is in the group with GID 75.

`(8489 20284)`
: was last accessed on Aug 19 00:09. Unfortunately, you cannot convert this number into a time string in Emacs.

`(8489 20284)`
: was last modified on Aug 19 00:09.

`(8489 20285)`
: last had its inode changed on Aug 19 00:09.

`14906`
: is 14906 characters long.

`"-rw-rw-rw-"`
: has a mode of read and write access for the owner, group, and world.

`nil`
: would retain the same GID if it were recreated.

`129500`
: has an inode number of 129500.

`-32252`

is on file system number -32252.

Contents of Directories

A directory is a kind of file that contains other files entered under various names. Directories are a feature of the file system.

Emacs can list the names of the files in a directory as a Lisp list, or display the names in a buffer using the `ls` shell command. In the latter case, it can optionally display information about each file, depending on the value of switches passed to the `ls` command.

Function: **directory-files** *directory &optional full-name match-regexp nosort*

This function returns a list of the names of the files in the directory directory. By default, the list is in alphabetical order.

If full-name is non-`nil`, the function returns the files' absolute file names. Otherwise, it returns just the names relative to the specified directory.

If match-regexp is non-`nil`, this function returns only those file names that contain that regular expression--the other file names are discarded from the list.

If nosort is non-`nil`, that inhibits sorting the list, so you get the file names in no particular order. Use this if you want the utmost possible speed and don't care what order the files are processed in. If the order of processing is visible to the user, then the user will probably be happier if you do sort the names.

```
(directory-files "~lewis")
     => ("#foo#" "#foo.el#" "." ".."
         "dired-mods.el" "files.texi"
         "files.texi.~1~")
```

An error is signaled if directory is not the name of a directory that can be read.

Function: **file-name-all-versions** *file dirname*

This function returns a list of all versions of the file named file in directory dirname.

Function: **insert-directory** *file switches &optional wildcard full-directory-p*

This function inserts a directory listing for directory dir, formatted according to switches. It leaves point after the inserted text.

The argument dir may be either a directory name or a file specification including wildcard characters. If wildcard is non-`nil`, that means treat file as a file specification with wildcards.

If full-directory-p is non-`nil`, that means file is a directory and switches do not contain `d', so that a full listing is expected.

This function works by running a directory listing program whose name is in the variable `insert-directory-program`. If wildcard is non-`nil`, it also runs the shell specified by `shell-file-name`, to expand the wildcards.

Variable: **insert-directory-program**

This variable's value is the program to run to generate a directory listing for the function `insert-directory`.

Creating and Deleting Directories

Function: **make-directory** *dirname*

This function creates a directory named dirname.

Function: **delete-directory** *dirname*

This function deletes the directory named dirname. The function `delete-file` does not work for files that are directories; you must use `delete-directory` in that case.

Changing File Names and Attributes

The functions in this section rename, copy, delete, link, and set the modes of files.

In the functions that have an argument newname, if a file by the name of newname already exists, the actions taken depend on the value of the argument ok-if-already-exists:

- A `file-already-exists` error is signaled if ok-if-already-exists is `nil`.
- Confirmation is requested if ok-if-already-exists is a number.
- No confirmation is requested if ok-if-already-exists is any other value, in which case the old file is removed.

Function: **add-name-to-file** *oldname newname &optional ok-if-already-exists*

This function gives the file named oldname the additional name newname. This means that newname becomes a new "hard link" to oldname.

In the first part of the following example, we list two files, `` `foo' `` and `` `foo3' ``.

```
% ls -l fo*
-rw-rw-rw-  1 rms       29 Aug 18 20:32 foo
-rw-rw-rw-  1 rms       24 Aug 18 20:31 foo3
```

Then we evaluate the form `(add-name-to-file "~/lewis/foo" "~/lewis/foo2")`. Again we list the files. This shows two names, `` `foo' `` and `` `foo2' ``.

```
(add-name-to-file "~/lewis/foo1" "~/lewis/foo2")
```


```
     => nil

% ls -l fo*
-rw-rw-rw-  2 rms       29 Aug 18 20:32 foo
-rw-rw-rw-  2 rms       29 Aug 18 20:32 foo2
-rw-rw-rw-  1 rms       24 Aug 18 20:31 foo3
```

Finally, we evaluate the following:

```
(add-name-to-file "~/lewis/foo" "~/lewis/foo3" t)
```

and list the files again. Now there are three names for one file: `foo', `foo2', and `foo3'. The old contents of `foo3' are lost.

```
(add-name-to-file "~/lewis/foo1" "~/lewis/foo3")
     => nil

% ls -l fo*
-rw-rw-rw-  3 rms       29 Aug 18 20:32 foo
-rw-rw-rw-  3 rms       29 Aug 18 20:32 foo2
-rw-rw-rw-  3 rms       29 Aug 18 20:32 foo3
```

This function is meaningless on VMS, where multiple names for one file are not allowed.

See also `file-nlinks` in section Other Information about Files.

Command: **rename-file** *filename newname &optional ok-if-already-exists*

This command renames the file filename as newname.

If filename has additional names aside from filename, it continues to have those names. In fact, adding the name newname with `add-name-to-file` and then deleting filename has the same effect as renaming, aside from momentary intermediate states.

In an interactive call, this function prompts for filename and newname in the minibuffer; also, it requests confirmation if newname already exists.

Command: **copy-file** *oldname newname &optional ok-if-exists time*

This command copies the file oldname to newname. An error is signaled if oldname does not exist.

If time is non-`nil`, then this functions gives the new file the same last-modified time that the old one has. (This works on only some operating systems.)

In an interactive call, this function prompts for filename and newname in the minibuffer; also, it requests confirmation if newname already exists.

Command: **delete-file** *filename*

This command deletes the file filename, like the shell command `rm filename'. If the file has multiple

names, it continues to exist under the other names.

A suitable kind of `file-error` error is signaled if the file does not exist, or is not deletable. (In Unix, a file is deletable if its directory is writable.)

See also `delete-directory` in section Creating and Deleting Directories.

Command: **make-symbolic-link** *filename newname &optional ok-if-exists*

This command makes a symbolic link to filename, named newname. This is like the shell command `ln -s filename newname'.

In an interactive call, filename and newname are read in the minibuffer, and ok-if-exists is set to the numeric prefix argument.

Function: **define-logical-name** *varname string*

This function defines the logical name name to have the value string. It is available only on VMS.

Function: **set-file-modes** *filename mode*

This function sets mode bits of filename to mode (which must be an integer). Only the 12 low bits of mode are used.

Function: **set-default-file-modes** *mode*

This function sets the default file protection for new files created by Emacs and its subprocesses. Every file created with Emacs initially has this protection. On Unix, the default protection is the bitwise complement of the "umask" value.

The argument mode must be an integer. Only the 9 low bits of mode are used.

Saving a modified version of an existing file does not count as creating the file; it does not change the file's mode, and does not use the default file protection.

Function: **default-file-modes**

This function returns the current default protection value.

File Names

Files are generally referred to by their names, in Emacs as elsewhere. File names in Emacs are represented as strings. The functions that operate on a file all expect a file name argument.

In addition to operating on files themselves, Emacs Lisp programs often need to operate on the names; i.e., to take them apart and to use part of a name to construct related file names. This section describes how to manipulate file names.

The functions in this section do not actually access files, so they can operate on file names that do not refer to an existing file or directory.

On VMS, all these functions understand both VMS file name syntax and Unix syntax. This is so that the standard Lisp libraries can specify file names in Unix syntax and work properly on VMS without change.

File Name Components

The operating system groups files into directories. To specify a file, you must specify the directory, and the file's name in that directory. Therefore, a file name in Emacs is considered to have two main parts: the directory name part, and the nondirectory part (or file name within the directory). Either part may be empty. Concatenating these two parts reproduces the original file name.

On Unix, the directory part is everything up to and including the last slash; the nondirectory part is the rest. The rules in VMS syntax are complicated.

For some purposes, the nondirectory part is further subdivided into the name proper and the version number. On Unix, only backup files have version numbers in their names; on VMS, every file has a version number, but most of the time the file name actually used in Emacs omits the version number. Version numbers are found mostly in directory lists.

Function: **file-name-directory** *filename*

This function returns the directory part of filename (or `nil` if filename does not include a directory part). On Unix, the function returns a string ending in a slash. On VMS, it returns a string ending in one of the three characters `:', `]', or `>'.

```
(file-name-directory "lewis/foo")  ; Unix example
     => "lewis/"
(file-name-directory "foo")        ; Unix example
     => nil
(file-name-directory "[X]FOO.TMP") ; VMS example
     => "[X]"
```

Function: **file-name-nondirectory** *filename*

This function returns the nondirectory part of filename.

```
(file-name-nondirectory "lewis/foo")
     => "foo"
(file-name-nondirectory "foo")
     => "foo"
;; The following example is accurate only on VMS.
(file-name-nondirectory "[X]FOO.TMP")
     => "FOO.TMP"
```

Function: **file-name-sans-versions** *filename*

This function returns filename without any file version numbers, backup version numbers, or trailing tildes.


```
(file-name-sans-versions "~rms/foo.~1~")
     => "~rms/foo"
(file-name-sans-versions "~rms/foo~")
     => "~rms/foo"
(file-name-sans-versions "~rms/foo")
     => "~rms/foo"
;; The following example applies to VMS only.
(file-name-sans-versions "foo;23")
     => "foo"
```

Directory Names

A directory name is the name of a directory. A directory is a kind of file, and it has a file name, which is related to the directory name but not identical to it. (This is not quite the same as the usual Unix terminology.) These two different names for the same entity are related by a syntactic transformation. On Unix, this is simple: a directory name ends in a slash, whereas the directory's name as a file lacks that slash. On VMS, the relationship is more complicated.

The difference between a directory name and its name as a file is subtle but crucial. When an Emacs variable or function argument is described as being a directory name, a file name of a directory is not acceptable.

These two functions take a single argument, filename, which must be a string. Environment variable substitutions such as `$HOME', and the symbols `~', and `..', are *not* expanded. Use `expand-file-name` or `substitute-in-file-name` for that (see section Functions that Expand Filenames).

Function: **file-name-as-directory** *filename*

This function returns a string representing filename in a form that the operating system will interpret as the name of a directory. In Unix, this means that a slash is appended to the string. On VMS, the function converts a string of the form `` `[X]Y.DIR.1' `` to the form `` `[X.Y]' ``.

```
(file-name-as-directory "~rms/lewis")
     => "~rms/lewis/"
```

Function: **directory-file-name** *dirname*

This function returns a string representing dirname in a form that the operating system will interpret as the name of a file. On Unix, this means removing a final slash from the string. On VMS, the function converts a string of the form `` `[X.Y]' `` to `` `[X]Y.DIR.1' ``.

```
(directory-file-name "~lewis/")
     => "~lewis"
```

Directory name abbreviations are useful for directories that are normally accessed through symbolic

links. Sometimes the users recognize primarily the link's name as "the name" of the directory, and fi
annoying to see the directory's "real" name. If you define the link name as an abbreviation for the "re
name, Emacs shows users the abbreviation instead.

If you wish to convert a directory name to its abbreviation, use this function:

Function: **abbreviate-file-name** *dirname*

This function applies abbreviations from `directory-abbrev-alist` to its argument, and substitutes `~' for the user's home directory.

Variable: **directory-abbrev-alist**

The variable `directory-abbrev-alist` contains an alist of abbreviations to use for file directories. Each element has the form `(from . to)`, and says to replace from with to when it appears in a directory name. The from string is actually a regular expression; it should always start with `^'. The function `abbreviate-file-name` performs these substitutions.

You can set this variable in `site-init.el' to describe the abbreviations appropriate for your site.

Here's an example, from a system on which file system `/home/fsf' and so on are normally accessed through symbolic links named `/fsf' and so on.

```
(("^/home/fsf" . "/fsf")
 ("^/home/gp" . "/gp")
 ("^/home/gd" . "/gd"))
```

Absolute and Relative File Names

All the directories in the file system form a tree starting at the root directory. A file name can specify all the directory names starting from the root of the tree; then it is called an absolute file name. Or it can specify the position of the file in the tree relative to a default directory; then it is called an relative file name. On Unix, an absolute file name starts with a slash or a tilde (`~'), and a relative one does not. The rules on VMS are complicated.

Function: **file-name-absolute-p** *filename*

This function returns `t` if file filename is an absolute file name, `nil` otherwise. On VMS, this function understands both Unix syntax and VMS syntax.

```
(file-name-absolute-p "~rms/foo")
     => t
(file-name-absolute-p "rms/foo")
     => nil
(file-name-absolute-p "/user/rms/foo")
     => t
```

Functions that Expand Filenames

Expansion of a file name means converting a relative file name to an absolute one. Since this is done relative to a default directory, you must specify the default directory name as well as the file name to be expanded. Expansion also simplifies file names by eliminating redundancies such as `./' and `name/../'.

Function: **expand-file-name** *filename &optional directory*

This function converts filename to an absolute file name. If directory is supplied, it is the directory to start with if filename is relative. (The value of directory should itself be an absolute, expanded file name; it should not start with `~'.) Otherwise, the current buffer's value of `default-directory` is used. For example:

```
(expand-file-name "foo")
     => "/xcssun/users/rms/lewis/foo"
(expand-file-name "../foo")
     => "/xcssun/users/rms/foo"
(expand-file-name "foo" "/usr/spool/")
     => "/usr/spool/foo"
(expand-file-name "$HOME/foo")
     => "/xcssun/users/rms/lewis/$HOME/foo"
```

Filenames containing `.' or `..' are simplified to their canonical form:

```
(expand-file-name "bar/../foo")
     => "/xcssun/users/rms/lewis/foo"
```

`~/' is expanded into the user's home directory. A `/' or `~' following a `/' is taken to be the start of an absolute file name that overrides what precedes it, so everything before that `/' or `~' is deleted. For example:

```
(expand-file-name
 "/a1/gnu//usr/local/lib/emacs/etc/MACHINES")
     => "/usr/local/lib/emacs/etc/MACHINES"
(expand-file-name "/a1/gnu/~/foo")
     => "/xcssun/users/rms/foo"
```

In both cases, `/a1/gnu/' is discarded because an absolute file name follows it.

Note that `expand-file-name` does *not* expand environment variables; that is done only by `substitute-in-file-name`.

Function: **file-relative-name** *filename directory*

This function does the inverse of expansion--it tries to return a relative name which is equivalent to filename when interpreted relative to directory. (If such a relative name would be longer than the

absolute name, it returns the absolute name instead.)

```
(file-relative-name "/foo/bar" "/foo/")
     => "bar")
(file-relative-name "/foo/bar" "/hack/")
     => "/foo/bar")
```

Variable: **default-directory**

The value of this buffer-local variable is the default directory for the current buffer. It is local in every buffer. `expand-file-name` uses the default directory when its second argument is `nil`.

On Unix systems, the value is always a string ending with a slash.

```
default-directory
     => "/user/lewis/manual/"
```

Function: **substitute-in-file-name** *filename*

This function replaces environment variables names in filename with the values to which they are set by the operating system. Following standard Unix shell syntax, `$' is the prefix to substitute an environment variable value.

The environment variable name is the series of alphanumeric characters (including underscores) that follow the `$'. If the character following the `$' is a `{', then the variable name is everything up to the matching `}'.

Here we assume that the environment variable `HOME`, which holds the user's home directory name, has the value `/xcssun/users/rms'.

```
(substitute-in-file-name "$HOME/foo")
     => "/xcssun/users/rms/foo"
```

If a `~' or a `/' appears following a `/', after substitution, everything before the following `/' is discarded:

```
(substitute-in-file-name "bar/~/foo")
     => "~/foo"
(substitute-in-file-name "/usr/local/$HOME/foo")
     => "/xcssun/users/rms/foo"
```

On VMS, `$' substitution is not done, so this function does nothing on VMS except discard superfluous initial components as shown above.

Generating Unique File Names

Some programs need to write temporary files. Here is the usual way to construct a name for such a file:


```
(make-temp-name (concat "/tmp/" name-of-application))
```

Here we use the directory `/tmp/' because that is the standard place on Unix for temporary files. The job of `make-temp-name` is to prevent two different users or two different jobs from trying to use the same name.

Function: **make-temp-name** *string*

This function generates string that can be used as a unique name. The name starts with the prefix string, and ends with a number that is different in each Emacs job.

```
(make-temp-name "/tmp/foo")
     => "/tmp/foo021304"
```

To prevent conflicts among different application libraries run in the same Emacs, each application should have its own string. The number added to the end of the name distinguishes between the same application running in different Emacs jobs.

File Name Completion

This section describes low-level subroutines for completing a file name. For other completion functions, see section Completion.

Function: **file-name-all-completions** *partial-filename directory*

This function returns a list of all possible completions for a file whose name starts with partial-filename in directory directory. The order of the completions is the order of the files in the directory, which is unpredictable and conveys no useful information.

The argument partial-filename must be a file name containing no directory part and no slash. The current buffer's default directory is prepended to directory, if directory is not an absolute file name.

In the following example, suppose that the current default directory, `~rms/lewis', has five files whose names begin with `f': `foo', `file~', `file.c', `file.c.~1~', and `file.c.~2~'.

```
(file-name-all-completions "f" "")
     => ("foo" "file~" "file.c.~2~"
               "file.c.~1~" "file.c")

(file-name-all-completions "fo" "")
     => ("foo")
```

Function: **file-name-completion** *filename directory*

This function completes the file name filename in directory directory. It returns the longest prefix common to all file names in directory directory that start with filename.

If only one match exists and filename matches it exactly, the function returns `t`. The function returns `nil` if directory directory contains no name starting with filename.

In the following example, suppose that the current default directory has five files whose names begin with `f': `foo', `file~', `file.c', `file.c.~1~', and `file.c.~2~'.

```
(file-name-completion "fi" "")
     => "file"

(file-name-completion "file.c.~1" "")
     => "file.c.~1~"

(file-name-completion "file.c.~1~" "")
     => t

(file-name-completion "file.c.~3" "")
     => nil
```

User Option: **completion-ignored-extensions**

`file-name-completion` usually ignores file names that end in any string in this list. It does not ignore them when all the possible completions end in one of these suffixes or when a buffer showing all possible completions is displayed.

A typical value might look like this:

```
completion-ignored-extensions
     => (".o" ".elc" "~" ".dvi")
```

Making Certain File Names "Magic"

You can implement special handling for certain file names. This is called making those names magic. You must supply a regular expression to define the class of names (all those which match the regular expression), plus a handler that implements all the primitive Emacs file operations for file names that do match.

The value of `file-name-handler-alist` is a list of handlers, together with regular expressions that decide when to apply each handler. Each element has this form:

```
(regexp . handler)
```

All the Emacs primitives for file access and file name transformation check the given file name against `file-name-handler-alist`. If the file name matches regexp, the primitives handle that file by calling handler.

The first argument given to handler is the name of the primitive; the remaining arguments are the arguments that were passed to that operation. (The first of these arguments is typically the file name itself.) For example, if you do this:


```
(file-exists-p filename)
```

and filename has handler handler, then handler is called like this:

```
(funcall handler 'file-exists-p filename)
```

Here are the operations that you can handle for a magic file name:

```
add-name-to-file, copy-file, delete-directory,
delete-file, directory-file-name, directory-files,
dired-compress-file, dired-uncache,
expand-file-name, file-accessible-directory-p,
file-attributes, file-directory-p,
file-executable-p, file-exists-p, file-local-copy,
file-modes, file-name-all-completions,
file-name-as-directory, file-name-completion,
file-name-directory, file-name-nondirectory,
file-name-sans-versions, file-newer-than-file-p,
file-readable-p, file-symlink-p, file-writable-p,
insert-directory, insert-file-contents,
make-directory, make-symbolic-link, rename-file,
set-file-modes, set-visited-file-modtime,
unhandled-file-name-directory,
verify-visited-file-modtime, write-region.
```

The handler function must handle all of the above operations, and possibly others to be added in the future. Therefore, it should always reinvoke the ordinary Lisp primitive when it receives an operation it does not recognize. Here's one way to do this:

```
(defun my-file-handler (operation &rest args)
  ;; First check for the specific operations
  ;; that we have special handling for.
  (cond ((eq operation 'insert-file-contents) ...)
        ((eq operation 'write-region) ...)
        ...
        ;; Handle any operation we don't know about.
        (t (let (file-name-handler-alist)
             (apply operation args)))))
```

Function: **find-file-name-handler** *file*

This function returns the handler function for file name file, or `nil` if there is none.

Function: **file-local-copy** *filename*

This function copies file filename to the local site, if it isn't there already. If filename specifies a "magic" file name which programs outside Emacs cannot directly read or write, this copies the contents to an

ordinary file and returns that file's name.

If filename is an ordinary file name, not magic, then this function does nothing and returns `nil`.

Function: **unhandled-file-name-directory** *filename*

This function returns the name of a directory that is not magic. It uses the directory part of filename if that is not magic. Otherwise, it asks the handler what to do.

This is used for running a subprocess; any subprocess must have a non-magic directory to serve as its current directory.

Go to the previous, next section.

Go to the previous, next section.

Backups and Auto-Saving

Backup files and auto-save files are two methods by which Emacs tries to protect the user from the consequences of crashes or of the user's own errors. Auto-saving preserves the text from earlier in the current editing session; backup files preserve file contents prior to the current session.

Backup Files

A backup file is a copy of the old contents of a file you are editing. Emacs makes a backup file the first time you save a buffer into its visited file. Normally, this means that the backup file contains the contents of the file as it was before the current editing session. The contents of the backup file normally remain unchanged once it exists.

Backups are usually made by renaming the visited file to a new name. Optionally, you can specify that backup files should be made by copying the visited file. This choice makes a difference for files with multiple names; it also can affect whether the edited file remains owned by the original owner or becomes owned by the user editing it.

By default, Emacs makes a single backup file for each file edited. You can alternatively request numbered backups; then each new backup file gets a new name. You can delete old numbered backups when you don't want them any more, or Emacs can delete them automatically.

Making Backup Files

Function: **backup-buffer**

This function makes a backup of the file visited by the current buffer, if appropriate. It is called by `save-buffer` before saving the buffer the first time.

Variable: **buffer-backed-up**

This buffer-local variable indicates whether this buffer's file has been backed up on account of this buffer. If it is non-`nil`, then the backup file has been written. Otherwise, the file should be backed up when it is next saved (if backup files are enabled). This is a permanent local; `kill-local-variables` does not alter it.

User Option: **make-backup-files**

This variable determines whether or not to make backup files. If it is non-`nil`, then Emacs creates a backup of each file when it is saved for the first time.

The following example shows how to change the `make-backup-files` variable only in the `` `RMAIL' `` buffer and not elsewhere. Setting it `nil` stops Emacs from making backups of the `` `RMAIL' `` file, which may save disk space. (You would put this code in your `` `.emacs' `` file.)


```
(add-hook 'rmail-mode-hook
          (function (lambda ()
                      (make-local-variable
                       'make-backup-files)
                      (setq make-backup-files nil))))
```

Variable: **backup-enable-predicate** *filename*

This variable's value is a function to be called on certain occasions to decide whether a there should be backup files for file name filename. If it returns `nil`, backups are disabled. Otherwise, backups are enabled (if `make-backup-files` is true).

Backup by Renaming or by Copying?

There are two ways that Emacs can make a backup file:

- Emacs can rename the original file so that it becomes a backup file, and then write the buffer being saved into a new file. After this procedure, any other names (i.e., hard links) of the original file now refer to the backup file. The new file is owned by the user doing the editing, and its group is the default for new files written by the user in that directory.
- Emacs can copy the original file into a backup file, and then overwrite the original file with new contents. After this procedure, any other names (i.e., hard links) of the original file still refer to the current version of the file. The file's owner and group will be unchanged.

The first method, renaming, is the default.

The variable `backup-by-copying`, if non-`nil`, says to use the second method, which is to copy the original file and overwrite it with the new buffer contents. The variable `file-precious-flag`, if non-`nil`, also has this effect (as a sideline of its main significance). See section Saving Buffers.

The following two variables, when non-`nil`, cause the second method to be used in certain special cases. They have no effect on the treatment of files that don't fall into the special cases.

Variable: **backup-by-copying**

This variable controls whether to make backup files by copying. If it is non-`nil`, then Emacs always copies the current contents of the file into the backup file before writing the buffer to be saved to the file. (In many circumstances, this has the same effect as `file-precious-flag`.)

Variable: **backup-by-copying-when-linked**

This variable controls whether to make backups by copying for files with multiple names (hard links). If it is non-`nil`, then Emacs uses copying to create backups for those files.

This variable is significant only if `backup-by-copying` is `nil`, since copying is always used when that variable is non-`nil`.

Variable: **backup-by-copying-when-mismatch**

This variable controls whether to make backups by copying in cases where renaming would change either the owner or the group of the file. If it is non-`nil` then Emacs creates backups by copying in s cases.

The value has no effect when renaming would not alter the owner or group of the file; that is, for files which are owned by the user and whose group matches the default for a new file created there by the user.

This variable is significant only if `backup-by-copying` is `nil`, since copying is always used when that variable is non-`nil`.

Making and Deleting Numbered Backup Files

If a file's name is `` `foo' ``, the names of its numbered backup versions are `` `foo.~v~' ``, for various integers v, like this: `` `foo.~1~' ``, `` `foo.~2~' ``, `` `foo.~3~' ``, ..., `` `foo.~259~' ``, and so on.

User Option: **version-control**

This variable controls whether to make a single non-numbered backup file or multiple numbered backups.

`nil`
: Make numbered backups if the visited file already has numbered backups; otherwise, do not.

`never`
: Do not make numbered backups.

anything else
: Do make numbered backups.

The use of numbered backups ultimately leads to a large number of backup versions, which must then be deleted. Emacs can do this automatically.

User Option: **kept-new-versions**

The value of this variable is the number of oldest versions to keep when a new numbered backup is made. The newly made backup is included in the count. The default value is 2.

User Option: **kept-old-versions**

The value of this variable is the number of oldest versions to keep when a new numbered backup is made. The default value is 2.

User Option: **dired-kept-versions**

This variable plays a role in Dired's `dired-clean-directory` (.) command like that played by `kept-old-versions` when a backup file is made. The default value is 2.

If there are backups numbered 1, 2, 3, 5, and 7, and both of these variables have the value 2, then the backups numbered 1 and 2 are kept as old versions and those numbered 5 and 7 are kept as new versions; backup version 3 is deleted. The function `find-backup-file-name` (see section Naming Backup

Files) is responsible for determining which backup versions to delete, but does not delete them itself.

User Option: **trim-versions-without-asking**

If this variable is non-`nil`, then saving a file deletes excess backup versions silently. Otherwise, it asks the user whether to delete them.

Naming Backup Files

The functions in this section are documented mainly because you can customize the naming conventions for backup files by redefining them. If you change one, you probably need to change the rest.

Function: **backup-file-name-p** *filename*

This function returns a non-`nil` value if filename is a possible name for a backup file. A file with the name filename need not exist; the function just checks the name.

```
(backup-file-name-p "foo")
     => nil
(backup-file-name-p "foo~")
     => 3
```

The standard definition of this function is as follows:

```
(defun backup-file-name-p (file)
  "Return non-nil if FILE is a backup file \
name (numeric or not)..."
  (string-match "~$" file))
```

Thus, the function returns a non-`nil` value if the file name ends with a `~'. (We use a backslash to split the documentation string's first line into two lines in the text, but produce just one line in the string itself.)

This simple expression is placed in a separate function to make it easy to redefine for customization.

Function: **make-backup-file-name** *filename*

This function returns a string which is the name to use for a non-numbered backup file for file filename. On Unix, this is just filename with a tilde appended.

The standard definition of this function is as follows:

```
(defun make-backup-file-name (file)
  "Create the non-numeric backup file name for FILE..."
  (concat file "~"))
```

You can change the backup file naming convention by redefining this function. In the following example, `make-backup-file-name` is redefined to prepend a `.' as well as to append a tilde.


```
(defun make-backup-file-name (filename)
  (concat "." filename "~"))

(make-backup-file-name "backups.texi")
     => ".backups.texi~"
```

Function: **find-backup-file-name** *filename*

This function computes the file name for a new backup file for filename. It may also propose certain existing backup files for deletion. `find-backup-file-name` returns a list whose CAR is the name for the new backup file and whose CDR is a list of backup files whose deletion is proposed.

Two variables, `kept-old-versions` and `kept-new-versions`, determine which old backup versions should be kept (by excluding them from the list of backup files ripe for deletion). See section Making and Deleting Numbered Backup Files.

In this example, the value says that `` `~rms/foo.~5~' `` is the name to use for the new backup file, and `` `~rms/foo.~3~' `` is an "excess" version that the caller should consider deleting now.

```
(find-backup-file-name "~rms/foo")
     => ("~rms/foo.~5~" "~rms/foo.~3~")
```

Function: **file-newest-backup** *filename*

This function returns the name of the most recent backup file for filename, or `nil` that file has no backup files.

Some file comparison commands use this function in order to compare a file by default with its most recent backup.

Auto-Saving

Emacs periodically saves all files that you are visiting; this is called auto-saving. Auto-saving prevents you from losing more than a limited amount of work if the system crashes. By default, auto-saves happen every 300 keystrokes, or after around 30 seconds of idle time. See section 'Auto-Saving: Protection Against Disasters' in The GNU Emacs Manual, for information on auto-save for users. Here we describe the functions used to implement auto-saving and the variables that control them.

Variable: **buffer-auto-save-file-name**

This buffer-local variable is the name of the file used for auto-saving the current buffer. It is `nil` if the buffer should not be auto-saved.

```
buffer-auto-save-file-name
=> "/xcssun/users/rms/lewis/#files.texi#"
```

Command: **auto-save-mode** *arg*

When used interactively without an argument, this command is a toggle switch: it turns on auto-saving in the current buffer if it is off, and vice-versa. With an argument arg, the command turns auto-saving on if the value of arg is `t`, a nonempty list, or a positive integer. Otherwise, it turns auto-saving off.

Function: **auto-save-file-name-p** *filename*

This function returns a non-`nil` value if filename is a string that could be the name of an auto-save file. It works based on knowledge of the naming convention for auto-save files: a name that begins and ends with hash marks (`#') is a possible auto-save file name. The argument filename should not contain a directory part.

```
(make-auto-save-file-name)
     => "/xcssun/users/rms/lewis/#files.texi#"
(auto-save-file-name-p "#files.texi#")
     => 0
(auto-save-file-name-p "files.texi")
     => nil
```

The standard definition of this function is as follows:

```
(defun auto-save-file-name-p (filename)
  "Return non-nil if FILENAME can be yielded by..."
  (string-match "^#.*#$" filename))
```

This function exists so that you can customize it if you wish to change the naming convention for auto-save files. If you redefine it, be sure to redefine the function `make-auto-save-file-name` correspondingly.

Function: **make-auto-save-file-name**

This function returns the file name to use for auto-saving the current buffer. This is just the file name with hash marks (`#') appended and prepended to it. This function does not look at the variable `auto-save-visited-file-name`; that should be checked before this function is called.

```
(make-auto-save-file-name)
     => "/xcssun/users/rms/lewis/#backup.texi#"
```

The standard definition of this function is as follows:

```
(defun make-auto-save-file-name ()
  "Return file name to use for auto-saves \
of current buffer..."
  (if buffer-file-name
      (concat
       (file-name-directory buffer-file-name)
       "#"
       (file-name-nondirectory buffer-file-name)
```


```
    "#")
  (expand-file-name
   (concat "#%" (buffer-name) "#"))))
```

This exists as a separate function so that you can redefine it to customize the naming convention for auto-save files. Be sure to change `auto-save-file-name-p` in a corresponding way.

Variable: **auto-save-visited-file-name**

If this variable is non-`nil`, Emacs auto-saves buffers in the files they are visiting. That is, the auto-save is done in the same file which you are editing. Normally, this variable is `nil`, so auto-save files have distinct names that are created by `make-auto-save-file-name`.

When you change the value of this variable, the value does not take effect until the next time auto-save mode is reenabled in any given buffer. If auto-save mode is already enabled, auto-saves continue to go in the same file name until `auto-save-mode` is called again.

Function: **recent-auto-save-p**

This function returns `t` if the current buffer has been auto-saved since the last time it was read in or saved.

Function: **set-buffer-auto-saved**

This function marks the current buffer as auto-saved. The buffer will not be auto-saved again until the buffer text is changed again. The function returns `nil`.

User Option: **auto-save-interval**

The value of this variable is the number of characters that Emacs reads from the keyboard between auto-saves. Each time this many more characters are read, auto-saving is done for all buffers in which it is enabled.

User Option: **auto-save-timeout**

The value of this variable is the number of seconds of idle time that should cause auto-saving. Each time the user pauses for this long, Emacs auto-saves any buffers that need it. (Actually, the specified timeout is multiplied by a factor depending on the size of the current buffer.)

Variable: **auto-save-hook**

This normal hook is run whenever an auto-save is about to happen.

User Option: **auto-save-default**

If this variable is non-`nil`, buffers that are visiting files have auto-saving enabled by default. Otherwise, they do not.

Command: **do-auto-save** *&optional no-message*

This function auto-saves all buffers that need to be auto-saved. This is all buffers for which auto-saving is enabled and that have been changed since the last time they were auto-saved.

Normally, if any buffers are auto-saved, a message `Auto-saving...' is displayed in the echo area while auto-saving is going on. However, if no-message is non-`nil`, the message is inhibited.

Function: **delete-auto-save-file-if-necessary**

This function deletes the current buffer's auto-save file if `delete-auto-save-files` is non-`nil`. It is called every time a buffer is saved.

Variable: **delete-auto-save-files**

This variable is used by the function `delete-auto-save-file-if-necessary`. If it is non-`nil`, Emacs deletes auto-save files when a true save is done (in the visited file). This saves on disk space and unclutters your directory.

Function: **rename-auto-save-file**

This function adjusts the current buffer's auto-save file name if the visited file name has changed. It also renames an existing auto-save file. If the visited file name has not changed, this function does nothing.

Reverting

If you have made extensive changes to a file and then change your mind about them, you can get rid of them by reading in the previous version of the file with the `revert-buffer` command. See section 'Reverting a Buffer' in The GNU Emacs Manual.

Command: **revert-buffer** *&optional check-auto-save noconfirm*

This command replaces the buffer text with the text of the visited file on disk. This action undoes all changes since the file was visited or saved.

If the argument check-auto-save is non-`nil`, and the latest auto-save file is more recent than the visited file, `revert-buffer` asks the user whether to use that instead. Otherwise, it always uses the text of the visited file itself. Interactively, check-auto-save is set if there is a numeric prefix argument.

When the value of the noconfirm argument is non-`nil`, `revert-buffer` does not ask for confirmation for the reversion action. This means that the buffer contents are deleted and replaced by the text from the file on the disk, with no further opportunities for the user to prevent it.

Since reverting works by deleting the entire text of the buffer and inserting the file contents, all the buffer's markers are relocated to point at the beginning of the buffer. This is not "correct", but then, there is no way to determine what would be correct. It is not possible to determine, from the text before and after, which characters after reversion correspond to which characters before.

If the value of the `revert-buffer-function` variable is non-`nil`, it is called as a function with no arguments to do the work.

Variable: **revert-buffer-function**

The value of this variable is the function to use to revert this buffer; but if the value of this variable is `nil`, then the `revert-buffer` function carries out its default action. Modes such as Dired mode, in

which the text being edited does not consist of a file's contents but can be regenerated in some other fashion, give this variable a buffer-local value that is a function to regenerate the contents.

Variable: **revert-buffer-insert-file-contents-function**

The value of this variable, if non-`nil`, is the function to use to insert contents when reverting this buffer. The function receives two arguments, first the file name to use, and second, `t` if the user has asked to read the auto-save file.

Command: **recover-file** *filename*

This function visits filename, but gets the contents from its last auto-save file. This is useful after the system has crashed, to resume editing the same file without losing all the work done in the previous session.

An error is signaled if there is no auto-save file for filename, or if filename is newer than its auto-save file. If filename does not exist, but its auto-save file does, then the auto-save file is read as usual. This last situation may occur if you visited a nonexistent file and never actually saved it.

Go to the previous, next section.

Go to the previous, next section.

Buffers

A buffer is a Lisp object containing text to be edited. Buffers are used to hold the contents of files that are being visited; there may also be buffers which are not visiting files. While several buffers may exist at one time, exactly one buffer is designated the current buffer at any time. Most editing commands act on the contents of the current buffer. Each buffer, including the current buffer, may or may not be displayed in any windows.

Buffer Basics

Buffers in Emacs editing are objects which have distinct names and hold text that can be edited. Buffers appear to Lisp programs as a special data type. The contents of a buffer may be viewed as an extendable string; insertions and deletions may occur in any part of the buffer. See section Text.

A Lisp buffer object contains numerous pieces of information. Some of this information is directly accessible to the programmer through variables, while other information is only accessible through special-purpose functions. For example, the width of a tab character is directly accessible through a variable, while the value of point is accessible only through a primitive function.

Buffer-specific information that is directly accessible is stored in buffer-local variable bindings, which are variable values that are effective only in a particular buffer. This feature allows each buffer to override the values of certain variables. Most major modes override variables such as `fill-column` or `comment-column` in this way. For more information about buffer-local variables and functions related to them, see section Buffer-Local Variables.

For functions and variables related to visiting files in buffers, see section Visiting Files and section Saving Buffers. For functions and variables related to the display of buffers in windows, see section Buffers and Windows.

Function: **bufferp** *object*

This function returns `t` if object is a buffer, `nil` otherwise.

Buffer Names

Each buffer has a unique name, which is a string. Many of the functions that work on buffers accept either a buffer or a buffer name as an argument. Any argument called buffer-or-name is of this sort, and an error is signaled if it is neither a string nor a buffer. Any argument called buffer is required to be an actual buffer object, not a name.

Buffers that are ephemeral and generally uninteresting to the user have names starting with a space, which prevents them from being listed by the `list-buffers` or `buffer-menu` commands. (A name

starting with space also initially disables recording undo information; see section Undo.)

Function: **buffer-name** *&optional buffer*

This function returns the name of buffer as a string. If buffer is not supplied, it defaults to the current buffer.

If `buffer-name` returns `nil`, it means that buffer has been killed. See section Killing Buffers.

```
(buffer-name)
     => "buffers.texi"

(setq foo (get-buffer "temp"))
     => #<buffer temp>
(kill-buffer foo)
     => nil
(buffer-name foo)
     => nil
foo
     => #<killed buffer>
```

Command: **rename-buffer** *newname &optional unique*

This function renames the current buffer to newname. An error is signaled if newname is not a string, or if there is already a buffer with that name. The function returns `nil`.

Ordinarily, `rename-buffer` signals an error if newname is already in use. However, if unique is non-`nil`, it modifies newname to make a name that is not in use. Interactively, you can make unique non-`nil` with a numeric prefix argument.

One application of this command is to rename the `*shell*' buffer to some other name, thus making it possible to create a second shell buffer under the name `*shell*'.

Function: **get-buffer** *buffer-or-name*

This function returns the buffer specified by buffer-or-name. If buffer-or-name is a string and there is no buffer with that name, the value is `nil`. If buffer-or-name is a buffer, it is returned as given. (That is not very useful, so the argument is usually a name.) For example:

```
(setq b (get-buffer "lewis"))
     => #<buffer lewis>
(get-buffer b)
     => #<buffer lewis>
(get-buffer "Frazzle-nots")
     => nil
```

See also the function `get-buffer-create` in section Creating Buffers.

Function: **generate-new-buffer-name** *starting-name*

This function returns a name that would be unique for a new buffer--but does not create the buffer. It starts with starting-name, and produces a name not currently in use for any buffer by appending a number inside of `<...>'.

See the related function `generate-new-buffer` in section Creating Buffers.

Buffer File Name

The buffer file name is the name of the file that is visited in that buffer. When a buffer is not visiting a file, its buffer file name is `nil`. Most of the time, the buffer name is the same as the nondirectory part of the buffer file name, but the buffer file name and the buffer name are distinct and can be set independently. See section Visiting Files.

Function: **buffer-file-name** *&optional buffer*

This function returns the absolute file name of the file that buffer is visiting. If buffer is not visiting any file, `buffer-file-name` returns `nil`. If buffer is not supplied, it defaults to the current buffer.

```
(buffer-file-name (other-buffer))
     => "/usr/user/lewis/manual/files.texi"
```

Variable: **buffer-file-name**

This buffer-local variable contains the name of the file being visited in the current buffer, or `nil` if it is not visiting a file. It is a permanent local, unaffected by `kill-local-variables`.

```
buffer-file-name
     => "/usr/user/lewis/manual/buffers.texi"
```

It is risky to change this variable's value without doing various other things. See the definition of `set-visited-file-name` in `files.el`; some of the things done there, such as changing the buffer name, are not strictly necessary, but others are essential to avoid confusing Emacs.

Variable: **buffer-file-truename**

This buffer-local variable holds the truename of the file visited in the current buffer, or `nil` if no file is visited. It is a permanent local, unaffected by `kill-local-variables`. See section Truenames.

Variable: **buffer-file-number**

This buffer-local variable holds the file number and directory device number of the file visited in the current buffer, or `nil` if no file or a nonexistent file is visited. It is a permanent local, unaffected by `kill-local-variables`. See section Truenames.

The value is normally a list of the form `(filenum devnum)`. This pair of numbers uniquely identifies the file among all files accessible on the system. See the function `file-attributes`, in section Other Information about Files, for more information about them.

TOC

Function: **get-file-buffer** *filename*

This function returns the buffer visiting file filename. If there is no such buffer, it returns `nil`. The argument filename, which must be a string, is expanded (see section Functions that Expand Filenames), then compared against the visited file names of all live buffers.

```
(get-file-buffer "buffers.texi")
    => #<buffer buffers.texi>
```

In unusual circumstances, there can be more than one buffer visiting the same file name. In such cases, this function returns the first such buffer in the buffer list.

Command: **set-visited-file-name** *filename*

If filename is a non-empty string, this function changes the name of the file visited in current buffer to filename. (If the buffer had no visited file, this gives it one.) The *next time* the buffer is saved it will go in the newly-specified file. This command marks the buffer as modified, since it does not (as far as Emacs knows) match the contents of filename, even if it matched the former visited file.

If filename is `nil` or the empty string, that stands for "no visited file". In this case, `set-visited-file-name` marks the buffer as having no visited file.

When the function `set-visited-file-name` is called interactively, it prompts for filename in the minibuffer.

See also `clear-visited-file-modtime` and `verify-visited-file-modtime` in section Buffer Modification.

Variable: **list-buffers-directory**

This buffer-local variable records a string to display in a buffer listing in place of the visited file name, for buffers that don't have a visited file name. Dired buffers use this variable.

Buffer Modification

Emacs keeps a flag called the modified flag for each buffer, to record whether you have changed the text of the buffer. This flag is set to `t` whenever you alter the contents of the buffer, and cleared to `nil` when you save it. Thus, the flag shows whether there are unsaved changes. The flag value is normally shown in the mode line (see section Variables Used in the Mode Line), and controls saving (see section Saving Buffers) and auto-saving (see section Auto-Saving).

Some Lisp programs set the flag explicitly. For example, the function `set-visited-file-name` sets the flag to `t`, because the text does not match the newly-visited file, even if it is unchanged from the file formerly visited.

The functions that modify the contents of buffers are described in section Text.

Function: **buffer-modified-p** *&optional buffer*

This function returns `t` if the buffer buffer has been modified since it was last read in from a file or saved, or `nil` otherwise. If buffer is not supplied, the current buffer is tested.

Function: **set-buffer-modified-p** *flag*

This function marks the current buffer as modified if flag is non-`nil`, or as unmodified if the flag is `nil`.

Another effect of calling this function is to cause unconditional redisplay of the mode line for the current buffer. In fact, the function `force-mode-line-update` works by doing this:

```
(set-buffer-modified-p (buffer-modified-p))
```

Command: **not-modified**

This command marks the current buffer as unmodified, and not needing to be saved. Don't use this function in programs, since it prints a message in the echo area; use `set-buffer-modified-p` (above) instead.

Function: **buffer-modified-tick** *&optional buffer*

This function returns buffer`s modification-count. This is a counter that increments every time the buffer is modified. If buffer is `nil` (or omitted), the current buffer is used.

Comparison of Modification Time

Suppose that you visit a file and make changes in its buffer, and meanwhile the file itself is changed on disk. At this point, saving the buffer would overwrite the changes in the file. Occasionally this may be what you want, but usually it would lose valuable information. Emacs therefore checks the file's modification time using the functions described below before saving the file.

Function: **verify-visited-file-modtime** *buffer*

This function compares Emacs's record of the modification time for the file that the buffer is visiting against the actual modification time of the file as recorded by the operating system. The two should be the same unless some other process has written the file since Emacs visited or saved it.

The function returns `t` if the last actual modification time and Emacs's recorded modification time are the same, `nil` otherwise.

Function: **clear-visited-file-modtime**

This function clears out the record of the last modification time of the file being visited by the current buffer. As a result, the next attempt to save this buffer will not complain of a discrepancy in file modification times.

This function is called in `set-visited-file-name` and other exceptional places where the usual test to avoid overwriting a changed file should not be done.

Function: **set-visited-file-modtime** *&optional time*

This function updates the buffer's record of the last modification time of the visited file, to the value specified by time if time is not `nil`, and otherwise to the last modification time of the visited file.

If time is not `nil`, it should have the form `(high . low)` or `(high low)`, in either case containing two integers, each of which holds 16 bits of the time. (This is the same format that `file-attributes` uses to return time values; see section Other Information about Files.)

This function is useful if the buffer was not read from the file normally, or if the file itself has been changed for some known benign reason.

Function: **visited-file-modtime**

This function returns the buffer's recorded last file modification time, as a list of the form `(high . low)`. Note that this is not identical to the last modification time of the file that is visited (though under normal circumstances the values are equal).

Function: **ask-user-about-supersession-threat** *fn*

This function is used to ask a user how to proceed after an attempt to modify an obsolete buffer. An obsolete buffer is an unmodified buffer for which the associated file on disk is newer than the last save-time of the buffer. This means some other program has probably altered the file.

This function is called automatically by Emacs on the proper occasions. It exists so you can customize Emacs by redefining it. See the file `` `userlock.el' `` for the standard definition.

Depending on the user's answer, the function may return normally, in which case the modification of the buffer proceeds, or it may signal a `file-supersession` error with data `(fn)`, in which case the proposed buffer modification is not allowed.

See also the file locking mechanism in section File Locks.

Read-Only Buffers

A buffer may be designated as read-only. This means that the buffer's contents may not be modified, although you may change your view of the contents by scrolling, narrowing, or widening, etc.

Read-only buffers are used in two kinds of situations:

- A buffer visiting a file is made read-only if the file is write-protected.

  Here, the purpose is to show the user that editing the buffer with the aim of saving it in the file may be futile or undesirable. The user who wants to change the buffer text despite this can do so after clearing the read-only flag with the function `toggle-read-only`.
- Modes such as Dired and Rmail make buffers read-only when altering the contents with the usual editing commands is probably a mistake.

  The special commands of the mode in question bind `buffer-read-only` to `nil` (with `let`) around the places where they change the text.

Variable: **buffer-read-only**

This buffer-local variable specifies whether the buffer is read-only. The buffer is read-only if this variable is non-`nil`.

Command: **toggle-read-only**

This command changes whether the current buffer is read-only. It is intended for interactive use; don't use it in programs. At any given point in a program, you should know whether you want the read-only flag on or off; so you can set `buffer-read-only` explicitly to the proper value, `t` or `nil`.

Function: **barf-if-buffer-read-only**

This function signals a `buffer-read-only` error if the current buffer is read-only. See section Interactive Call, for another way to signal an error if the current buffer is read-only.

The Buffer List

The buffer list is a list of all buffers that have not been killed. The order of the buffers in the list is based primarily on how recently each buffer has been displayed in the selected window. Several functions, notably `other-buffer`, make use of this ordering.

Function: **buffer-list**

This function returns a list of all buffers, including those whose names begin with a space. The elements are actual buffers, not their names.

```
(buffer-list)
     => (#<buffer buffers.texi>
         #<buffer  *Minibuf-1*> #<buffer buffer.c>
         #<buffer *Help*> #<buffer TAGS>)

;; Note that the name of the minibuffer
;;   begins with a space!

(mapcar (function buffer-name) (buffer-list))
    => ("buffers.texi" " *Minibuf-1*"
         "buffer.c" "*Help*" "TAGS")
```

Buffers appear earlier in the list if they were current more recently.

This list is a copy of a list used inside Emacs; modifying it has no effect on the buffers.

Function: **other-buffer** *&optional buffer-or-name visible-ok*

This function returns the first buffer in the buffer list other than buffer-or-name. Usually this is the buffer most recently shown in the selected window, aside from buffer-or-name. Buffers are moved to the front of the list when they are selected and to the end when they are buried. Buffers whose names start with a

space are not even considered.

If buffer-or-name is not supplied (or if it is not a buffer), then `other-buffer` returns the first buffer on the buffer list that is not visible in any window.

Normally, `other-buffer` avoids returning a buffer visible in any window, except as a last resort. However, if visible-ok is non-`nil`, then a buffer displayed in some window is admissible to return.

If no suitable buffer exists, the buffer `*scratch*' is returned (and created, if necessary).

Command: **list-buffers** *&optional files-only*

This function displays a listing of the names of existing buffers. It clears the buffer `*Buffer List*', then inserts the listing into that buffer and displays it in a window. `list-buffers` is intended for interactive use, and is described fully in The GNU Emacs Manual. It returns `nil`.

Command: **bury-buffer** *&optional buffer-or-name*

This function puts buffer-or-name at the end of the buffer list without changing the order of any of the other buffers on the list. This buffer therefore becomes the least desirable candidate for `other-buffer` to return, and appears last in the list displayed by `list-buffers`.

If buffer-or-name is `nil` or omitted, this means to bury the current buffer. In addition, this switches to some other buffer (obtained using `other-buffer`) in the selected window. If the buffer is displayed in a window other than the selected one, it remains there.

If you wish to remove a buffer from all the windows that display it, you can do so with a loop that uses `get-buffer-window`. See section Buffers and Windows.

Creating Buffers

This section describes the two primitives for creating buffers. `get-buffer-create` creates a buffer if it finds no existing buffer; `generate-new-buffer` always creates a new buffer, and gives it a unique name.

Other functions you can use to create buffers include `with-output-to-temp-buffer` (see section Temporary Displays) and `create-file-buffer` (see section Visiting Files).

Function: **get-buffer-create** *name*

This function returns a buffer named name. If such a buffer already exists, it is returned. If such a buffer does not exist, one is created and returned. The buffer does not become the current buffer--this function does not change which buffer is current.

An error is signaled if name is not a string.

```
(get-buffer-create "foo")
     => #<buffer foo>
```

The major mode for the new buffer is set by the value of `default-major-mode`. See section How Emacs Chooses a Major Mode.

Function: **generate-new-buffer** *name*

This function returns a newly created, empty buffer, but does not make it current. If there is no buffer named name, then that is the name of the new buffer. If that name is in use, this function adds suffixes of the form `<n>' are added to name, where n is an integer. It tries successive integers starting with 2 until it finds an available name.

An error is signaled if name is not a string.

```
(generate-new-buffer "bar")
     => #<buffer bar>
(generate-new-buffer "bar")
     => #<buffer bar<2>>
(generate-new-buffer "bar")
     => #<buffer bar<3>>
```

The major mode for the new buffer is set by the value of `default-major-mode`. See section How Emacs Chooses a Major Mode.

See the related function `generate-new-buffer-name` in section Buffer Names.

Killing Buffers

Killing a buffer makes its name unknown to Emacs and makes its space available for other use.

The buffer object for the buffer which has been killed remains in existence as long as anything refers to it, but it is specially marked so that you cannot make it current or display it. Killed buffers retain their identity, however; two distinct buffers, when killed, remain distinct according to `eq`.

If you kill a buffer that is current or displayed in a window, Emacs automatically selects or displays some other buffer instead. This means that killing a buffer can in general change the current buffer. Therefore, when you kill a buffer, you should also take the precautions associated with changing the current buffer (unless you happen to know that the buffer being killed isn't current). See section The Current Buffer.

The `buffer-name` of a killed buffer is `nil`. You can use this feature to test whether a buffer has been killed:

```
(defun killed-buffer-p (buffer)
  "Return t if BUFFER is killed."
  (not (buffer-name buffer)))
```

Command: **kill-buffer** *buffer-or-name*

This function kills the buffer buffer-or-name, freeing all its memory for use as space for other buffers.

(Emacs version 18 and older was unable to return the memory to the operating system.) It returns `nil`.

Any processes that have this buffer as the `process-buffer` are sent the `SIGHUP` signal, which normally causes them to terminate. (The usual meaning of `SIGHUP` is that a dialup line has been disconnected.) See section Deleting Processes.

If the buffer is visiting a file when `kill-buffer` is called and the buffer has not been saved since it was last modified, the user is asked to confirm before the buffer is killed. This is done even if `kill-buffer` is not called interactively. To prevent the request for confirmation, clear the modified flag before calling `kill-buffer`. See section Buffer Modification.

Just before actually killing the buffer, after asking all questions, `kill-buffer` runs the normal hook `kill-buffer-hook`. The buffer to be killed is current when the hook functions run. See section Hooks.

Killing a buffer that is already dead has no effect.

```
(kill-buffer "foo.unchanged")
     => nil
(kill-buffer "foo.changed")

---------- Buffer: Minibuffer ----------
Buffer foo.changed modified; kill anyway? (yes or no) yes
---------- Buffer: Minibuffer ----------

     => nil
```

The Current Buffer

There are, in general, many buffers in an Emacs session. At any time, one of them is designated as the current buffer. This is the buffer in which most editing takes place, because most of the primitives for examining or changing text in a buffer operate implicitly on the current buffer (see section Text). Normally the buffer that is displayed on the screen in the selected window is the current buffer, but this is not always so: a Lisp program can designate any buffer as current temporarily in order to operate on its contents, without changing what is displayed on the screen.

The way to designate a current buffer in a Lisp program is by calling `set-buffer`. The specified buffer remains current until a new one is designated.

When an editing command returns to the editor command loop, the command loop designates the buffer displayed in the selected window as current, to prevent confusion: the buffer that the cursor is in, when Emacs reads a command, is the one to which the command will apply. (See section Command Loop.) Therefore, `set-buffer` is not usable for switching visibly to a different buffer so that the user can edit it. For this, you must use the functions described in section Displaying Buffers in Windows.

However, Lisp functions that change to a different current buffer should not leave it to the command loop

to set it back afterwards. Editing commands written in Emacs Lisp can be called from other programs as well as from the command loop. It is convenient for the caller if the subroutine does not change which buffer is current (unless, of course, that is the subroutine's purpose). Therefore, you should normally use `set-buffer` within a `save-excursion` that will restore the current buffer when your program is done (see section Excursions). Here is an example, the code for the command `append-to-buffer` (with the documentation string abridged):

```
(defun append-to-buffer (buffer start end)
  "Append to specified buffer the text of the region..."
  (interactive "BAppend to buffer: \nr")
  (let ((oldbuf (current-buffer)))
    (save-excursion
      (set-buffer (get-buffer-create buffer))
      (insert-buffer-substring oldbuf start end))))
```

This function binds a local variable to the current buffer, and then `save-excursion` records the values of point, the mark, and the original buffer. Next, `set-buffer` makes another buffer current. Finally, `insert-buffer-substring` copies the string from the original current buffer to the new current buffer.

If the buffer appended to happens to be displayed in some window, then the next redisplay will show how its text has changed. Otherwise, you will not see the change immediately on the screen. The buffer becomes current temporarily during the execution of the command, but this does not cause it to be displayed.

Changing the current buffer between the binding and unbinding of a buffer-local variable can cause it to be bound in one buffer, and then unbound in another! You can avoid this problem by using save-excursion to make sure that the buffer from which the variable was bound is current again whenever the variable is unbound.

```
(let (buffer-read-only)
  (save-excursion
    (set-buffer ...)
    ...))
```

Function: **current-buffer**

This function returns the current buffer.

```
(current-buffer)
     => #<buffer buffers.texi>
```

Function: **set-buffer** *buffer-or-name*

This function makes buffer-or-name the current buffer. However, it does not display the buffer in the currently selected window or in any other window. This means that the user cannot necessarily see the buffer, but Lisp programs can in any case work on it.

TOC

This function returns the buffer identified by buffer-or-name. An error is signaled if buffer-or-name does not identify an existing buffer.

Go to the previous, next section.

Go to the previous, next section.

Windows

This chapter describes most of the functions and variables related to Emacs windows. See section Emacs Display, for information on how text is displayed in windows.

Basic Concepts of Emacs Windows

A window is the physical area of the screen in which a buffer is displayed. The term is also used to refer to a Lisp object which represents that screen area in Emacs Lisp. It should be clear from the context which is meant.

There is always at least one window displayed on the screen, and there is exactly one window that we call the selected window. The cursor is in the selected window. The selected window's buffer is usually the current buffer (except when `set-buffer` has been used.) See section The Current Buffer.

For all intents, a window only exists while it is displayed on the terminal. Once removed from the display, the window is effectively deleted and should not be used, *even though there may still be references to it* from other Lisp objects. Restoring a saved window configuration is the only way for a window no longer on the screen to come back to life. (See section Deleting Windows.)

Each window has the following attributes:

- containing frame
- window height
- window width
- window edges with respect to the screen or frame
- the buffer it displays
- position within the buffer at the upper left of the window
- the amount of horizontal scrolling, in columns
- point
- the mark
- how recently the window was selected

Applications use multiple windows for a variety of reasons, but most often to give different views of the same information. In Rmail, for example, you can move through a summary buffer in one window while the other window shows messages one at a time as they are reached.

The term "window" in Emacs means something similar to what it means in the context of general puprpose window systems such as X, but not identical. The X Window System subdivides the screen into X windows; Emacs uses one or more X windows, called frames in Emacs terminology, and subdivides each of them into (nonoverlapping) Emacs windows. When you use Emacs on an ordinary display

terminal, Emacs subdivides the terminal screen into Emacs windows.

Most window systems support arbitrarily located overlapping windows. In contrast, Emacs windows are tiled; they never overlap, and together they fill the whole of the screen or frame. Because of the way in which Emacs creates new windows and resizes them, you can't create every conceivable tiling on an Emacs screen. See section Splitting Windows. Also, see section The Size of a Window.

See section Emacs Display, for information on how the contents of the window's buffer are displayed in the window.

Function: **windowp** *object*

This function returns `t` if object is a window.

Splitting Windows

The functions described here are the primitives used to split a window into two windows. Two higher level functions sometimes split a window, but not always: `pop-to-buffer` and `display-buffer` (see section Displaying Buffers in Windows).

The functions described here do not accept a buffer as an argument. They let the two "halves" of the split window display the same buffer previously visible in the window that was split.

Function: **one-window-p** *&optional no-mini*

This function returns non-`nil` if there is only one window. The argument no-mini, if non-`nil`, means don't count the minibuffer even if it is active; otherwise, the minibuffer window is included, if active, in the total number of windows which is compared against one.

Command: **split-window** *&optional window size horizontal*

This function splits window into two windows. The original window window remains the selected window, but occupies only part of its former screen area. The rest is occupied by a newly created window which is returned as the value of this function.

If horizontal is non-`nil`, then window splits side by side, keeping the leftmost size columns and giving the rest of the columns to the new window. Otherwise, it splits into halves one above the other, keeping the upper size lines and giving the rest of the lines to the new window. The original window is therefore the right-hand or upper of the two, and the new window is the left-hand or lower.

If window is omitted or `nil`, then the selected window is split. If size is omitted or `nil`, then window is divided evenly into two parts. (If there is an odd line, it is allocated to the new window.) When `split-window` is called interactively, all its arguments are `nil`.

The following example starts with one window on a screen that is 50 lines high by 80 columns wide; then the window is split.

```
(setq w (selected-window))
     => #<window 8 on windows.texi>
```


```
(window-edges)          ; Edges in order:
     => (0 0 80 50)     ;   left--top--right--bottom

;; Returns window created
(setq w2 (split-window w 15))
     => #<window 28 on windows.texi>
(window-edges w2)
     => (0 15 80 50)    ; Bottom window;
                        ;   top is line 15
(window-edges w)
     => (0 0 80 15)     ; Top window
```

The screen looks like this:

```
          ___________
         |           |  line 0
         |     w     |
         |___________|
         |           |  line 15
         |     w2    |
         |___________|
                       line 50
 column 0   column 80
```

Next, the top window is split horizontally:

```
(setq w3 (split-window w 35 t))
     => #<window 32 on windows.texi>
(window-edges w3)
     => (35 0 80 15)  ; Left edge at column 35
(window-edges w)
     => (0 0 35 15)   ; Right edge at column 35
(window-edges w2)
     => (0 15 80 50)  ; Bottom window unchanged
```

Now, the screen looks like this:

```
     column 35
          ___________
         |   |       |  line 0
         | w |  w3   |
         |___|_______|
         |           |  line 15
         |     w2    |
         |___________|
                       line 50
```


```
column 0   column 80
```

Command: **split-window-vertically** *size*

This function splits the selected window into two windows, one above the other, leaving the selected window with size lines.

This function is simply an interface to `split-windows`. Here is the complete function definition for it:

```
(defun split-window-vertically (&optional arg)
  "Split selected window into two windows,
one above the other..."
  (interactive "P")
  (split-window nil (and arg (prefix-numeric-value arg))))
```

Command: **split-window-horizontally** *size*

This function splits the selected window into two windows side-by-side, leaving the selected window with size columns.

This function is simply an interface to `split-windows`. Here is the complete definition for `split-window-horizontally` (except for part of the documentation string):

```
(defun split-window-horizontally (&optional arg)
  "Split selected window into two windows
side by side..."
  (interactive "P")
  (split-window nil (and arg (prefix-numeric-value arg)) t))
```

Deleting Windows

A window remains visible on its frame unless you delete it by calling certain functions that delete windows. A deleted window cannot appear on the screen, but continues to exist as a Lisp object until there are no references to it. There is no way to cancel the deletion of a window aside from restoring a saved window configuration (see section Window Configurations). Restoring a window configuration also deletes any windows that aren't part of that configuration.

When you delete a window, the space it took up is given to one adjacent sibling. (In Emacs version 18, the space was divided evenly among all the siblings.)

Function: **window-live-p** *window*

This function returns `nil` if window is deleted, and `t` otherwise.

Warning: erroneous information or fatal errors may result from using a deleted window as if it were live.

Command: **delete-window** *&optional window*

This function removes window from the display. If window is omitted, then the selected window is deleted. An error is signaled if there is only one window when `delete-window` is called.

This function returns `nil`.

When `delete-window` is called interactively, window defaults to the selected window.

Command: **delete-other-windows** *&optional window*

This function makes window the only window on its frame, by deleting all the other windows. If window is omitted or `nil`, then the selected window is used by default.

The result is `nil`.

Command: **delete-windows-on** *buffer*

This function deletes all windows showing buffer. If there are no windows showing buffer, then this function does nothing. If all windows in some frame are showing buffer (including the case where there is only one window), then the frame reverts to having a single window showing the buffer chosen by `other-buffer`. See section The Buffer List.

If there are several windows showing different buffers, then those showing buffer are removed, and the others are expanded to fill the void.

The result is `nil`.

Selecting Windows

When a window is selected, the buffer in the window becomes the current buffer, and the cursor will appear in it.

Function: **selected-window**

This function returns the selected window. This is the window in which the cursor appears and to which many commands apply.

Function: **select-window** *window*

This function makes window the selected window. The cursor then appears in window (on redisplay). The buffer being displayed in window is immediately designated the current buffer.

The return value is window.

```
(setq w (next-window))
(select-window w)
     => #<window 65 on windows.texi>
```

The following functions choose one of the windows on the screen, offering various criteria for the choice.

Function: **get-lru-window** *&optional all-frames*

This function returns the window least recently "used" (that is, selected). The selected window is always the most recently used window.

The selected window can be the least recently used window if it is the only window. A newly created window becomes the least recently used window until it is selected. The minibuffer window is not considered a candidate.

The argument all-frames controls which set of windows are considered. If it is non-`nil`, then all windows on all frames are considered. Otherwise, only windows in the selected frame are considered.

Function: **get-largest-window** *&optional all-frames*

This function returns the window with the largest area (height times width). If there are no side-by-side windows, then this is the window with the most lines. The minibuffer window is not considered a candidate.

If there are two windows of the same size, then the function returns the window which is first in the cyclic ordering of windows (see following section), starting from the selected window.

The argument all-frames controls which set of windows are considered. If it is non-`nil`, then all windows on all frames are considered. Otherwise, only windows in the selected frame are considered.

Cycling Ordering of Windows

When you use the command C-x o (`other-window`) to select the next window, it moves through all the windows on the screen in a specific cyclic order. For any given configuration of windows, this order never varies. It is called the cyclic ordering of windows.

This ordering generally goes from top to bottom, and from left to right. But it may go down first or go right first, depending on the order in which the windows were split.

If the first split was vertical (into windows one above each other), and then the subwindows were split horizontally, then the ordering is left to right in the top, and then left to right in the next lower part of the frame, and so on. If the first split was horizontal, the ordering is top to bottom in the left part, and so on. In general, within each set of siblings at any level in the window tree, the order is left to right, or top to bottom.

Function: **next-window** *window &optional minibuf all-frames*

This function returns the window following window in the cyclic ordering of windows. This is the window which C-x o would select if done when window is selected. If window is the only window visible, then this function returns window.

The value of the argument minibuf determines whether the minibuffer is included in the window order. Normally, when minibuf is `nil`, the minibuffer is included if it is currently active; this is the behavior of C-x o.

If minibuf is `t`, then the cyclic ordering includes the minibuffer window even if it is not active.

If minibuf is neither `t` nor `nil`, then the minibuffer window is not included even if it is active. (The minibuffer window is active while the minibuffer is in use. See section Minibuffers.)

When there are multiple frames, this functions normally cycles through all the windows in the selected frame, plus the minibuffer used by the selected frame even if it lies in some other frame.

If all-frames is `t`, then it cycles through all the windows in all the frames that currently exist.

If all-frames is neither `t` nor `nil`, then it cycles through precisely the windows in the selected frame, excluding the minibuffer in use if it lies in some other frame.

This example shows two windows, which both happen to be displaying the same buffer:

```
(selected-window)
     => #<window 56 on windows.texi>
(next-window (selected-window))
     => #<window 52 on windows.texi>
(next-window (next-window (selected-window)))
     => #<window 56 on windows.texi>
```

Function: **previous-window** *window &optional minibuf all-frames*

This function returns the window preceding window in the cyclic ordering of windows. The other arguments affect which windows are included in the cycle, as in `next-window`.

Command: **other-window** *count*

This function selects the countth next window in the cyclic order. If count is negative, then it selects the -countth preceding window. It returns `nil`.

In an interactive call, count is the numeric prefix argument.

Function: **walk-windows** *proc &optional minibuf all-frames*

This function cycles through all visible windows, calling `proc` once for each window with the window as its sole argument.

The optional argument minibuf says whether to include minibuffer windows. A value of `t` means count the minibuffer window even if not active. A value of `nil` means count it only if active. Any other value means not to count the minibuffer even if it is active.

If the optional third argument all-frames is `t`, that means include all windows in all frames. If all-frames is `nil`, it means to cycle within the selected frame, but include the minibuffer window (if minibuf says so) that that frame uses, even if it is on another frame. If all-frames is neither `nil` nor `t`, `walk-windows` sticks strictly to the selected frame.

Buffers and Windows

This section describes low-level functions to examine windows or to show buffers in windows in a precisely controlled fashion. See the following section for related functions that find a window to use and specify a buffer for it. The functions described there are easier to use than these, but they employ heuristics in choosing or creating a window; use these functions when you need complete control.

Function: **set-window-buffer** *window buffer-or-name*

This function makes window display buffer-or-name as its contents. It returns `nil`.

```
(set-window-buffer (selected-window) "foo")
     => nil
```

Function: **window-buffer** *&optional window*

This function returns the buffer that window is displaying. If window is omitted, then this function returns the buffer for the selected window.

```
(window-buffer)
     => #<buffer windows.texi>
```

Function: **get-buffer-window** *buffer-or-name &optional all-frames*

This function returns a window currently displaying buffer-or-name, or `nil` if there is none. If there are several such windows, then the function returns the first one in the cyclic ordering of windows, starting from the selected window. See section Cycling Ordering of Windows.

The argument all-frames controls which set of windows are considered. If it is non-`nil`, then all windows on all frames are considered. Otherwise, only windows in the selected frame are considered.

Command: **replace-buffer-in-windows** *buffer*

This function replaces buffer with some other buffer in all windows displaying it. The other buffer used is chosen with `other-buffer`. In the usual applications of this function, you don't care which other buffer is used; you just want to make sure that buffer is no longer displayed.

This function returns `nil`.

Displaying Buffers in Windows

In this section we describe convenient functions that choose a window automatically and use it to display a specified buffer. These functions can also split an existing window in certain circumstances. We also describe variables that parameterize the heuristics used for choosing a window. See the preceding section for low-level functions that give you more precise control.

Do not use the functions in this section in order to make a buffer current so that a Lisp program can

access or modify it; they are too drastic for that purpose, since they change the display of buffers in windows, which is gratuitous and will surprise the user. Instead, use `set-buffer` (see section The Current Buffer) and `save-excursion` (see section Excursions), which designate buffers as current for programmed access without affecting the display of buffers in windows.

Command: **switch-to-buffer** *buffer-or-name &optional norecord*

This function makes buffer-or-name the current buffer, and also displays the buffer in the selected window. This means that a human can see the buffer and subsequent keyboard commands will apply to it. Contrast this with `set-buffer`, which makes buffer-or-name the current buffer but does not display it in the selected window. See section The Current Buffer.

If buffer-or-name does not identify an existing buffer, then a new buffer by that name is created.

Normally the specified buffer is put at the front of the buffer list. This affects the operation of `other-buffer`. However, if norecord is non-`nil`, this is not done. See section The Buffer List.

The `switch-to-buffer` function is often used interactively, as the binding of C-x b. It is also used frequently in programs. It always returns `nil`.

Command: **switch-to-buffer-other-window** *buffer-or-name*

This function makes buffer-or-name the current buffer and displays it in a window not currently selected. It then selects that window. The handling of the buffer is the same as in `switch-to-buffer`.

The previously selected window is absolutely never used to display the buffer. If it is the only window, then it is split to make a distinct window for this purpose. If the selected window is already displaying the buffer, then it continues to do so, but another window is nonetheless found to display it in as well.

Function: **pop-to-buffer** *buffer-or-name &optional other-window*

This function makes buffer-or-name the current buffer and switches to it in some window, preferably not the window previously selected. The "popped-to" window becomes the selected window.

If the variable `pop-up-frames` is non-`nil`, `pop-to-buffer` creates a new frame to display the buffer in. Otherwise, if the variable `pop-up-windows` is non-`nil`, windows may be split to create a new window that is different from the original window. For details, see section Choosing a Window.

If other-window is non-`nil`, `pop-to-buffer` finds or creates another window even if buffer-or-name is already visible in the selected window. Thus buffer-or-name could end up displayed in two windows. On the other hand, if buffer-or-name is already displayed in the selected window and other-window is `nil`, then the selected window is considered sufficient display for buffer-or-name, so that nothing needs to be done.

If buffer-or-name is a string that does not name an existing buffer, a buffer by that name is created.

An example use of this function is found at the end of section Process Filter Functions.

Choosing a Window

This section describes the basic facility which chooses a window to display a buffer in---`display-buffer`. All the higher-level functions and commands use this subroutine. Here we describe how to use `display-buffer` and how to customize it.

Function: **display-buffer** *buffer-or-name &optional not-this-window*

This function makes buffer-or-name appear in some window, like `pop-to-buffer`, but it does not select that window and does not make the buffer current. The identity of the selected window is unaltered by this function.

If not-this-window is non-`nil`, it means that the specified buffer should be displayed in a window other than the selected one, even if it is already on display in the selected window. This can cause the buffer to appear in two windows at once. Otherwise, if buffer-or-name is already being displayed in any window, that is good enough, so this function does nothing.

`display-buffer` returns the window chosen to display buffer-or-name.

Precisely how `display-buffer` finds or creates a window depends on the variables described below.

A window can be marked as "dedicated" to its buffer. Then `display-buffer` does not try to use that window.

Function: **window-dedicated-p** *window*

This function returns `t` if window is marked as dedicated; otherwise `nil`.

Function: **set-window-dedicated-p** *window flag*

This function marks window as dedicated if flags is non-`nil`, and nondedicated otherwise.

User Option: **pop-up-windows**

This variable controls whether `display-buffer` makes new windows. If it is non-`nil` and there is only one window, then that window is split. If it is `nil`, then `display-buffer` does not split the single window, but rather replaces its buffer.

User Option: **split-height-threshold**

This variable determines when `display-buffer` may split a window, if there are multiple windows. `display-buffer` splits the largest window if it has at least this many lines.

If there is only one window, it is split regardless of this value, provided `pop-up-windows` is non-`nil`.

User Option: **pop-up-frames**

This variable controls whether `display-buffer` makes new frames. If it is non-`nil`, `display-buffer` makes a new frame. If it is `nil`, then `display-buffer` either splits a window or reuses one.

If this is non-`nil`, the variables `pop-up-windows` and `split-height-threshold` do not matter.

See section Frames, for more information.

Variable: **pop-up-frame-function**

This variable specifies how to make a new frame if `pop-up-frame` is non-`nil`.

Its value should be a function of no arguments. When `display-buffer` makes a new frame, it does so by calling that function, which should return a frame. The default value of the variable is a function which creates a frame using parameters from `pop-up-frame-alist`.

Variable: **pop-up-frame-alist**

This variable holds an alist specifying frame parameters used when `display-buffer` makes a new frame. See section Frame Parameters, for more information about frame parameters.

Variable: **display-buffer-function**

This variable is the most flexible way to customize the behavior of `display-buffer`. If it is non-`nil`, it should be a function that `display-buffer` calls to do the work. The function should accept two arguments, the same two arguments that `display-buffer` received. It should choose or create a window, display the specified buffer, and then return the window.

This hook takes precedence over all the other options and hooks described above.

Window Point

Each window has its own value of point, independent of the value of point in other windows displaying the same buffer. This makes it useful to have multiple windows showing one buffer.

- The window point is established when a window is first created; it is initialized from the buffer's point, or from the window point of another window opened on the buffer if such a window exists.
- Selecting a window sets the value of point in its buffer to the window's value of point. Conversely, deselecting a window sets the window's value of point from that of the buffer. Thus, when you switch between windows that display a given buffer, the point value for the selected window is in effect in the buffer, while the point values for the other windows are stored in those windows.
- As long as the selected window displays the current buffer, the window's point and the buffer's point always move together; they remain equal.
- See section Positions, for more details on positions.

As far as the user is concerned, point is where the cursor is, and when the user switches to another buffer, the cursor jumps to the position of point in that buffer.

Function: **window-point** *window*

This function returns the current position of point in window. For a nonselected window, this is the value point would have (in that window's buffer) if that window were selected.

When window is the selected window and its buffer is also the current buffer, the value returned is the same as point in that buffer.

Strictly speaking, it would be more correct to return the "top-level" value of point, outside of any `save-excursion` forms. But that value is hard to find.

Function: **set-window-point** *window position*

This function positions point in window at position position in window's buffer.

The Window Start Position

Each window contains a marker used to keep track of a buffer position which specifies where in the buffer display should start. This position is called the display-start position of the window (or just the start). The character after this position is the one that appears at the upper left corner of the window. It is usually, but not inevitably, at the beginning of a text line.

Function: **window-start** *&optional window*

This function returns the display-start position of window window. If window is `nil`, the selected window is used.

```
(window-start)
     => 7058
```

For a more complicated example of use, see the description of `count-lines` in section Motion by Text Lines.

Function: **window-end** *&optional window*

This function returns the position of the end of the display in window window. If window is `nil`, the selected window is used.

Function: **set-window-start** *window position &optional noforce*

This function sets the display-start position of window to position in window's buffer.

The display routines insist that the position of point be visible when a buffer is displayed. Normally, they change the display-start position (that is, scroll the window) whenever necessary to make point visible. However, if you specify the start position with this function with `nil` for noforce, it means you want display to start at position even if that would put the location of point off the screen. What the display routines do in this case is move point instead, to the left margin on the middle line in the window.

For example, if point is 1 and you attempt to set the start of the window to 2, then the position of point would be "above" the top of the window. The display routines would automatically move point if it is still 1 when redisplay occurs. Here is an example:

```
;; Here is what `foo' looks like before executing
```


```
;;   the set-window-start expression.

---------- Buffer: foo ----------
-!-This is the contents of buffer foo.
2
3
4
5
6
---------- Buffer: foo ----------

(set-window-start
 (selected-window)
 (1+ (window-start)))

;; Here is what `foo' looks like after executing
;;   the set-window-start expression.

---------- Buffer: foo ----------
his is the contents of buffer foo.
2
3
-!-4
5
6
---------- Buffer: foo ----------

     => 2
```

However, when noforce is non-`nil`, `set-window-start` does nothing if the specified start position would make point invisible.

This function returns position, regardless of whether the noforce option caused that position to be overruled.

Function: **pos-visible-in-window-p** *&optional position window*

This function returns `t` if position is within the range of text currently visible on the screen in window. It returns `nil` if position is scrolled vertically out of view. The argument position defaults to the current position of point; window, to the selected window. Here is an example:

```
(or
(pos-visible-in-window-p
 (point) (selected-window))
   (recenter 0))
```

The `pos-visible-in-window-p` function considers only vertical scrolling. It returns `t` if position

is out of view only because window has been scrolled horizontally. See section Horizontal Scrolling.

Vertical Scrolling

Vertical scrolling means moving the text up or down in a window. It works by changing the value of the window's display-start location. It may also change the value of `window-point` to keep it on the screen.

In the commands `scroll-up` and `scroll-down`, the directions "up" and "down" refer to the motion of the text in the buffer at which you are looking through the window. Imagine that the text is written on a long roll of paper and that the scrolling commands move the paper up and down. Thus, if you are looking at text in the middle of a buffer and repeatedly call `scroll-down`, you will eventually see the beginning of the buffer.

Some people have urged that the opposite convention be used: they imagine that the window moves over text that remains in place. Then "down" commands would take you to the end of the buffer. This view is more consistent with the actual relationship between windows and the text in the buffer, but it is less like what the user sees. The position of a window on the terminal does not move, and short scrolling commands clearly move the text up or down on the screen. We have chosen names that fit the user's point of view.

The scrolling functions (aside from `scroll-other-window`) will have unpredictable results if the current buffer is different from the buffer that is displayed in the selected window. See section The Current Buffer.

Command: **scroll-up** *&optional count*

This function scrolls the text in the selected window upward count lines. If count is negative, scrolling is actually downward.

If count is `nil` (or omitted), then the length of scroll is `next-screen-context-lines` lines less than the usable height of the window (not counting its mode line).

`scroll-up` returns `nil`.

Command: **scroll-down** *&optional count*

This function scrolls the text in the selected window downward count lines. If count is negative, scrolling is actually upward.

If count is omitted or `nil`, then the length of the scroll is `next-screen-context-lines` lines less than the usable height of the window.

`scroll-down` returns `nil`.

Command: **scroll-other-window** *&optional count*

This function scrolls the text in another window upward count lines. Negative values of count, or `nil`, are handled as in `scroll-up`.

The window that is scrolled is normally the one following the selected window in the cyclic ordering of windows--the window that `next-window` would return. See section Cycling Ordering of Windows.

If the selected window is the minibuffer, the next window is normally the one at the top left corner. However, you can specify the window to scroll by binding the variable `minibuffer-scroll-window`. This variable has no effect when any other window is selected. See section Minibuffer Miscellany.

When the minibuffer is active, it is the next window if the selected window is the one at the bottom right corner. In this case, `scroll-other-window` attempts to scroll the minibuffer. If the minibuffer contains just one line, it has nowhere to scroll to, so the line reappears after the echo area momentarily displays the message "Beginning of buffer".

Variable: **other-window-scroll-buffer**

If this variable is non-`nil`, it tells `scroll-other-window` which buffer to scroll.

User Option: **scroll-step**

This variable controls how scrolling is done automatically when point moves off the screen. If the value is zero, then the text is scrolled so that point is centered vertically in the window. If the value is a positive integer n, then if it is possible to bring point back on screen by scrolling n lines in either direction, that is done; otherwise, point is centered vertically as usual. The default value is zero.

User Option: **next-screen-context-lines**

The value of this variable is the number of lines of continuity to retain when scrolling by full screens. For example, when `scroll-up` executes, this many lines that were visible at the bottom of the window move to the top of the window. The default value is `2`.

Command: **recenter** *&optional count*

This function scrolls the selected window to put the text where point is located at a specified vertical position within the window.

If count is a nonnegative number, it puts the line containing point count lines down from the top of the window. If count is a negative number, then it counts upward from the bottom of the window, so that -1 stands for the last usable line in the window. If count is a non-`nil` list, then it stands for the line in the middle of the window.

If count is `nil`, then it puts the line containing point in the middle of the window, then clears and redisplays the entire selected frame.

When `recenter` is called interactively, Emacs sets count to the raw prefix argument. Thus, typing C-u as the prefix sets the count to a non-`nil` list, while typing C-u 4 sets count to 4, which positions the current line four lines from the top.

Typing C-u 0 C-l positions the current line at the top of the window. This action is so handy that some people bind the command to a function key. For example,


```
(defun line-to-top-of-window ()
  "Scroll current line to top of window.
Replaces three keystroke sequence C-u 0 C-l."
  (interactive)
  (recenter 0))

(global-set-key "\C-cl" 'line-to-top-of-window)
```

Horizontal Scrolling

Because we read English first from top to bottom and second from left to right, horizontal scrolling is not like vertical scrolling. Vertical scrolling involves selection of a contiguous portion of text to display. Horizontal scrolling causes part of each line to go off screen. The amount of horizontal scrolling is therefore specified as a number of columns rather than as a position in the buffer. It has nothing to do with the display-start position returned by `window-start`.

Usually, no horizontal scrolling is in effect; then the leftmost column is at the left edge of the window. In this state, scrolling to the right is meaningless, since there is no data to the left of the screen to be revealed by it, so it is not allowed. Scrolling to the left is allowed; it causes the first columns of text to go off the edge of the window and can reveal additional columns on the right that were truncated before. Once a window has a nonzero amount of leftward horizontal scrolling, you can scroll it back to the right, but only so far as to reduce the net horizontal scroll to zero. There is no limit to how far left you can scroll, but eventually all the text will disappear off the left edge.

Command: **scroll-left** *count*

This function scrolls the selected window count columns to the left (or to the right if count is negative). The return value is the total amount of leftward horizontal scrolling in effect after the change--just like the value returned by `window-hscroll`.

Command: **scroll-right** *count*

This function scrolls the selected window count columns to the right (or to the left if count is negative). The return value is the total amount of leftward horizontal scrolling in effect after the change--just like the value returned by `window-hscroll`.

Once you scroll a window as far right as it can go, back to its normal position where the total leftward scrolling is zero, attempts to scroll any farther have no effect.

Function: **window-hscroll** *&optional window*

This function returns the total leftward horizontal scrolling of window---the number of columns by which the text in window is scrolled left past the left margin.

The value is never negative. It is zero when no horizontal scrolling has been done in window (which is usually the case).

If window is `nil`, the selected window is used.


```
(window-hscroll)
     => 0
(scroll-left 5)
     => 5
(window-hscroll)
     => 5
```

Function: **set-window-hscroll** *window columns*

This function sets the number of columns from the left margin that window is scrolled to the value of columns. The argument columns should be zero or positive; if not, it is taken as zero.

The value returned is columns.

```
(set-window-hscroll (selected-window) 10)
     => 10
```

Here is how you can determine whether a given position position is off the screen due to horizontal scrolling:

```
(save-excursion
  (goto-char position)
  (and
   (>= (- (current-column) (window-hscroll window)) 0)
   (< (- (current-column) (window-hscroll window))
      (window-width window))))
```

The Size of a Window

An Emacs window is rectangular, and its size information consists of the height (the number of lines) and the width (the number of character positions in each line). The mode line is included in the height. For a window that does not abut the right hand edge of the screen, the column of `|' characters that separates it from the window on the right is included in the width.

The following three functions return size information about a window:

Function: **window-height** *&optional window*

This function returns the number of lines in window, including its mode line. If window fills its entire frame, this is one less than the value of `frame-height` on that frame (since the last line is always reserved for the minibuffer).

If window is `nil`, the function uses the selected window.

```
(window-height)
     => 23
```


```
(split-window-vertically)
     => #<window 4 on windows.texi>
(window-height)
     => 11
```

Function: **window-width** *&optional window*

This function returns the number of columns in window. If window fills its entire frame, this is the same as the value of `frame-width` on that frame.

If window is `nil`, the function uses the selected window.

```
(window-width)
     => 80
```

Function: **window-edges** *&optional window*

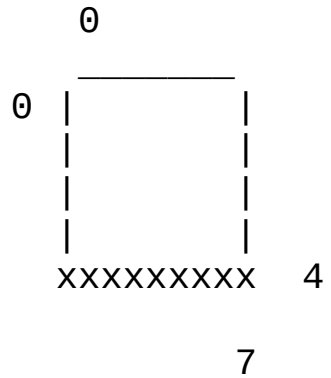
This function returns a list of the edge coordinates of window. If window is `nil`, the selected window is used.

The order of the list is `(left top right bottom)`, all elements relative to 0, 0 at the top left corner of the frame. The element right of the value is one more than the rightmost column used by window, and bottom is one more than the bottommost row used by window and its mode-line.

Here is the result obtained on a typical 24-line terminal with just one window:

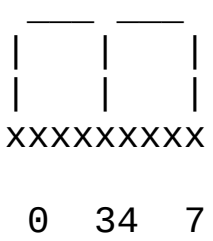
```
(window-edges (selected-window))
     => (0 0 80 23)
```

If window is at the upper left corner of its frame, right and bottom are the same as the values returned by `(window-width)` and `(window-height)` respectively, and top and bottom are zero. For example, the edges of the following window are `0 0 5 8'. Assuming that the frame has more than 8 columns, the last column of the window (column 7) holds a border rather than text. The last row (row 4) holds the mode line, shown here with `xxxxxxxxx'.



When there are side-by-side windows, any window not at the right edge of its frame has a border in its last column. This border counts as one column in the width of the window. A window never includes a border on its left, since the border there belongs to the window to the left.

In the following example, let's imagine that the frame is 7 columns wide. Then the edges of the left window are `0 0 4 3' and the edges of the right window are `4 0 7 3'.



Changing the Size of a Window

The window size functions fall into two classes: high-level commands that change the size of windows and low-level functions that access window size. Emacs does not permit overlapping windows or gaps between windows, so resizing one window affects other windows.

Command: **enlarge-window** *size &optional horizontal*

This function makes the selected window size lines bigger, stealing lines from neighboring windows. It takes the lines from one window at a time until that window is used up, then takes from another. If a window from which lines are stolen shrinks below `window-min-height` lines, then that window disappears.

If horizontal is non-`nil`, then this function makes window wider by size columns, stealing columns instead of lines. If a window from which columns are stolen shrinks below `window-min-width` columns, then that window disappears.

If the window's frame is smaller than size lines (or columns), then the function makes the window occupy the entire height (or width) of the frame.

If size is negative, this function shrinks the window by -size lines. If it becomes shorter than `window-min-height`, it disappears.

`enlarge-window` returns `nil`.

Command: **enlarge-window-horizontally** *columns*

This function makes the selected window columns wider. It could be defined as follows:

```
(defun enlarge-window-horizontally (columns)
  (enlarge-window columns t))
```

Command: **shrink-window** *size &optional horizontal*

This function is like `enlarge-window` but negates the argument size, making the selected window smaller by giving lines (or columns) to the other windows. If the window shrinks below `window-min-height` or `window-min-width`, then it disappears.

If size is negative, the window is enlarged by -size lines.

Command: **shrink-window-horizontally** *columns*

This function makes the selected window columns narrower. It could be defined as follows:

```
(defun shrink-window-horizontally (columns)
  (shrink-window columns t))
```

The following two variables constrain the window size changing functions to a minimum height and width.

User Option: **window-min-height**

The value of this variable determines how short a window may become before it disappears. A window disappears when it becomes smaller than `window-min-height`, and no window may be created that is smaller. The absolute minimum height is two (allowing one line for the mode line, and one line for the buffer display). Actions which change window sizes reset this variable to two if it is less than two. The default value is 4.

User Option: **window-min-width**

The value of this variable determines how narrow a window may become before it disappears. A window disappears when it becomes narrower than `window-min-width`, and no window may be created that is narrower. The absolute minimum width is one; any value below that is ignored. The default value is 10.

Coordinates and Windows

This section describes how to compare screen coordinates with windows.

Function: **window-at** *x y &optional frame*

This function returns the window containing the specified cursor position in the frame frame. The coordinates x and y are measured in characters and count from the top left corner of the screen or frame.

If you omit frame, the selected frame is used.

Function: **coordinates-in-window-p** *coordinates window*

This function checks whether a particular frame position falls within the window window.

The argument coordinates is a cons cell of this form:

```
(x . y)
```

The coordinates x and y are measured in characters, and count from the top left corner of the screen or frame.

The value of `coordinates-in-window-p` is non-`nil` if the coordinates are inside window. The

value also indicates what part of the window the position is in, as follows:

`(relx . rely)`
: The coordinates are inside window. The numbers relx and rely are the equivalent window-relative coordinates for the specified position, counting from 0 at the top left corner of the window.

`mode-line`
: The coordinates are in the mode line of window.

`vertical-split`
: The coordinates are in the vertical line between window and its neighbor to the right.

`nil`
: The coordinates are not in any sense within window.

The function `coordinates-in-window-p` does not require a frame as argument because it always uses the frame that window window is on.

Window Configurations

A window configuration records the entire layout of a frame--all windows, their sizes, which buffers they contain, what part of each buffer is displayed, and the values of point and the mark. You can bring back an entire previous layout by restoring a window configuration previously saved.

If you want to record all frames instead of just one, use a frame configuration instead of a window configuration. See section Frame Configurations.

Function: **current-window-configuration**

This function returns a new object representing Emacs's current window configuration, namely the number of windows, their sizes and current buffers, which window is the selected window, and for each window the displayed buffer, the display-start position, and the positions of point and the mark. An exception is made for point in the current buffer, whose value is not saved.

Function: **set-window-configuration** *configuration*

This function restores the configuration of Emacs's windows and buffers to the state specified by configuration. The argument configuration must be a value that was previously returned by `current-window-configuration`.

Here is a way of using this function to get the same effect as `save-window-excursion`:

```
(let ((config (current-window-configuration)))
  (unwind-protect
      (progn (split-window-vertically nil)
             ...)
    (set-window-configuration config)))
```

Special Form: **save-window-excursion** *forms...*

This special form executes forms in sequence, preserving window sizes and contents, including the w of point and the portion of the buffer which is visible. It also preserves the choice of selected windo However, it does not restore the value of point in the current buffer; use `save-excursion` for that.

The return value is the value of the final form in forms. For example:

```
(split-window)
     => #<window 25 on control.texi>
(setq w (selected-window))
     => #<window 19 on control.texi>
(save-window-excursion
  (delete-other-windows w)
  (switch-to-buffer "foo")
  'do-something)
     => do-something
     ;; The screen is now split again.
```

Function: **window-configuration-p** *object*

This function returns `t` if object is a window configuration.

Primitives to look inside of window configurations would make sense, but none are implemented. It is not clear they are useful enough to be worth implementing.

Go to the previous, next section.

Go to the previous, next section.

Frames

A frame is a rectangle on the screen that contains one or more Emacs windows. A frame initially contains a single main window (plus perhaps a minibuffer window) which you can subdivide vertically or horizontally into smaller windows.

When Emacs runs on a text-only terminal, it has just one frame, a terminal frame. There is no way to create another terminal frame after startup. If Emacs has an X display, it does not make a terminal frame; instead, it initially creates a single X window frame. You can create more; see section Creating Frames.

Function: **framep** *object*

This predicate returns `t` if object is a frame, and `nil` otherwise.

See section Emacs Display, for related information.

Creating Frames

To create a new frame, call the function `make-frame`.

Function: **make-frame** *alist*

This function creates a new frame, if the display mechanism permits creation of frames. (An X server does; an ordinary terminal does not.)

The argument is an alist specifying frame parameters. Any parameters not mentioned in alist default according to the value of the variable `default-frame-alist`; parameters not specified there either default from the standard X defaults file and X resources.

The set of possible parameters depends in principle on what kind of window system Emacs uses to display its the frames. See section X Window Frame Parameters, for documentation of individual parameters you can specify when creating an X window frame.

Variable: **default-frame-alist**

An alist specifying default values of frame parameters. Each element has the form:

```
(parameter . value)
```

If you use options that specify window appearance when you invoke Emacs, they take effect by adding elements to `default-frame-alist`.

Frame Parameters

A frame has many parameters that control how it displays.

Access to Frame Parameters

These functions let you read and change the parameter values of a frame.

Function: **frame-parameters** *frame*

The function `frame-parameters` returns an alist of all the parameters of frame.

Function: **modify-frame-parameters** *frame alist*

This function alters the parameters of frame frame based on the elements of alist. Each element of alist has the form `(parm . value)`, where parm is a symbol naming a parameter. If you don't mention a parameter in alist, its value doesn't change.

Initial Frame Parameters

You can specify the parameters for the initial startup frame by setting `initial-frame-alist` in your `` `.emacs' `` file.

Variable: **initial-frame-alist**

This variable's value is an alist of parameter values to when creating the initial X window frame.

If these parameters specify a separate minibuffer-only frame, and you have not created one, Emacs creates one for you.

Variable: **minibuffer-frame-alist**

This variable's value is an alist of parameter values to when creating an initial minibuffer-only frame--if such a frame is needed, according to the parameters for the main initial frame.

X Window Frame Parameters

Just what parameters a frame has depends on what display mechanism it uses. Here is a table of the parameters of an X window frame:

`name`
: The name of the frame.

`left`
: The screen position of the left edge, in pixels.

`top`
: The screen position of the top edge, in pixels.

`height`

The height of the frame contents, in pixels.

`width`

The width of the frame contents, in pixels.

`window-id`

The number of the X window for the frame.

`minibuffer`

Whether this frame has its own minibuffer. The value `t` means yes, `nil` means no, `only` means this frame is just a minibuffer, a minibuffer window (in some other frame) means the new frame uses that minibuffer.

`font`

The name of the font for text in the frame. This is a string.

`auto-raise`

Whether selecting the frame raises it (non-`nil` means yes).

`auto-lower`

Whether deselecting the frame lowers it (non-`nil` means yes).

`vertical-scroll-bars`

Whether the frame has a scroll bar for vertical scrolling (non-`nil` means yes).

`horizontal-scroll-bars`

Whether the frame has a scroll bar for horizontal scrolling (non-`nil` means yes). (Horizontal scroll bars are not currently implemented.)

`icon-type`

The type of icon to use for this frame when it is iconified. Non-`nil` specifies a bitmap icon, `nil` a text icon.

`foreground-color`

The color to use for the inside of a character. We use strings to designate colors; the X server defines the meaningful color names.

`background-color`

The color to use for the background of text.

`mouse-color`

The color for the mouse cursor.

`cursor-color`

The color for the cursor that shows point.

`border-color`

The color for the border of the frame.

`cursor-type`

The way to display the cursor. There are two legitimate values: `bar` and `box`. The value `bar` specifies a vertical bar between characters as the cursor. The value `box` specifies an ordinary black box overlaying the character after point; that is the default.

`border-width`

The width in pixels of the window border.

`internal-border-width`

The distance in pixels between text and border.

`unsplittable`

If non-`nil`, this frame's window is never split automatically.

`visibility`

The state of visibility of the frame. There are three possibilities: `nil` for invisible, `t` for visible, and `icon` for iconified. See section Visibility of Frames.

`menu-bar-lines`

The number of lines to allocate at the top of the frame for a menu bar. The default is zero. See section The Menu Bar.

`parent-id`

The X Window number of the window that should be the parent of this one. Specifying this lets you create an Emacs window inside some other application's window. (It is not certain this will be implemented; try it and see if it works.)

Frame Size And Position

You can read or change the size and position of a frame using the frame parameters `left`, `top`, `height` and `width`. When you create a frame, you must specify either both size parameters or neither. Likewise, you must specify either both position parameters or neither. Whatever geometry parameters you don't specify are chosen by the window manager in its usual fashion.

Here are some special features for working with sizes and positions:

Function: **set-frame-position** *frame left top*

This function sets the position of the top left corner of frame---to left and top. These arguments are measured in pixels, counting from the top left corner of the screen.

Function: **frame-height** *&optional frame*

Function: **frame-width** *&optional frame*

These functions return the height and width of frame, measured in characters. If you don't supply frame, they use the selected frame.

Function: **frame-pixel-height** *&optional frame*

Function: **frame-pixel-width** *&optional frame*

These functions return the height and width of frame, measured in pixels. If you don't supply frame, they use the selected frame.

Function: **frame-char-height** *&optional frame*

Function: **frame-char-width** *&optional frame*

These functions return the height and width, respectively, of a character in frame, measured in pixels. The values depend on the choice of font. If you don't supply frame, these functions use the selected frame.

Function: **set-frame-size** *frame cols rows*

This function sets the size of frame, measured in characters; cols and rows specify the new width and height.

To set the size with values measured in pixels, use `modify-frame-parameters` to set the `width` and `height` parameters. See section X Window Frame Parameters.

The old-fashioned functions `set-screen-height` and `set-screen-width`, which were used to specify the height and width of the screen in Emacs versions that did not support multiple frames, are still usable. They apply to the selected frame. See section Screen Size.

Function: **x-parse-geometry** *geom*

The function `x-parse-geometry` converts a standard X windows geometry string to an alist which you can use as part of the argument to `x-create-frame`.

The alist describes which parameters were specified in geom, and gives the values specified for them. Each element looks like `(parameter . value)`. The possible parameter values are `left`, `top`, `width`, and `height`.

```
(x-geometry "35x70+0-0")
     => ((width . 35) (height . 70) (left . 0) (top . -1))
```

Deleting Frames

Frames remain potentially visible until you explicitly delete them. A deleted frame cannot appear on the screen, but continues to exist as a Lisp object until there are no references to it. There is no way to cancel the deletion of a frame aside from restoring a saved frame configuration (see section Frame Configurations); this is similar to the way windows behave.

Command: **delete-frame** *&optional frame*

This function deletes the frame frame. By default, frame is the selected frame.

Function: **frame-live-p** *frame*

The function `frame-live-p` returns non-`nil` if the frame frame has not been deleted.

Finding All Frames

Function: **frame-list**

The function `frame-list` returns a list of all the frames that have not been deleted. It is analogous to `buffer-list` for buffers. The list that you get is newly created, so modifying the list doesn't have any effect on the internals of Emacs.

Function: **visible-frame-list**

This function returns a list of just the currently visible frames.

Function: **next-frame** *&optional frame minibuf*

The function `next-frame` lets you cycle conveniently through all the frames from an arbitrary starting point. It returns the "next" frame after frame in the cycle. If frame is omitted or `nil`, it defaults to the selected frame.

The second argument, minibuf, says which frames to consider:

`nil`
: Exclude minibuffer-only frames.

a window
: Consider only the frames using that particular window as their minibuffer.

anything else
: Consider all frames.

Frames and Windows

All the non-minibuffer windows in a frame are arranged in a tree of subdivisions; the root of this tree is available via the function `frame-root-window`. Each window is part of one and only one frame; you can get the frame with `window-frame`.

Function: **frame-root-window** *frame*

This returns the root window of frame frame.

Function: **window-frame** *window*

This function returns the frame that window is on.

At any time, exactly one window on any frame is selected within the frame. The significance of this designation is that selecting the frame also selects this window. You can get the frame's current selected window with `frame-selected-window`.

Function: **frame-selected-window** *frame*

This function returns the window on frame which is selected within frame.

Conversely, selecting a window for Emacs with `select-window` also makes that window selected within its frame. See section Selecting Windows.

Minibuffers and Frames

Normally, each frame has its own minibuffer window at the bottom, which is used whenever that frame is selected. If the frame has a minibuffer, you can get it with `minibuffer-window` (see section Minibuffer Miscellany).

However, you can also create a frame with no minibuffer. Such a frame must use the minibuffer window of some other frame. When you create the frame, you can specify explicitly the frame on which to find the minibuffer to use. If you don't, then the minibuffer is found in the frame which is the value of the variable `default-minibuffer-frame`. Its value should be a frame which does have a minibuffer.

Input Focus

At any time, one frame in Emacs is the selected frame. The selected window always resides on the selected frame.

Function: **selected-frame**

This function returns the selected frame.

The X server normally directs keyboard input to the X window that the mouse is in. Some window managers use mouse clicks or keyboard events to shift the focus to various X windows, overriding the normal behavior of the server.

Lisp programs can switch frames "temporarily" by calling the function `select-frame`. This does not override the window manager; rather, it escapes from the window manager's control until that control is somehow reasserted.

Function: **select-frame** *frame*

This function selects frame frame, temporarily disregarding the X Windows focus. The selection of frame lasts until the next time the user does something to select a different frame, or until the next time this function is called.

Emacs cooperates with the X server and the window managers by arranging to select frames according to what the server and window manager ask for. It does so by generating a special kind of input event, called a focus event. The command loop handles a focus event by calling `internal-select-frame`. See section Focus Events.

Function: **internal-select-frame** *frame*

This function selects frame frame, assuming that the X server focus already points to frame.

Focus events normally do their job by invoking this command. Don't call it for any other reason.

Visibility of Frames

A frame may be visible, invisible, or iconified. If it is visible, you can see its contents. If it is iconified, the frame's contents do not appear on the screen, but an icon does. If the frame is invisible, it doesn't show in the screen, not even as an icon.

Command: **make-frame-visible** *&optional frame*

This function makes frame frame visible. If you omit frame, it makes the selected frame visible.

Command: **make-frame-invisible** *&optional frame*

This function makes frame frame invisible. If you omit frame, it makes the selected frame invisible.

Command: **iconify-frame** *&optional frame*

This function iconifies frame frame. If you omit frame, it iconifies the selected frame.

Function: **frame-visible-p** *frame*

This returns the visibility status of frame frame. The value is `t` if frame is visible, `nil` if it is invisible, and `icon` if it is iconified.

The visibility status of a frame is also available as a frame parameter. You can read or change it as such. See section X Window Frame Parameters.

Raising and Lowering Frames

The X window system uses a desktop metaphor. Part of this metaphor is the idea that windows are stacked in a notional third dimension perpendicular to the screen surface, and thus ordered from "highest" to "lowest". Where two windows overlap, the one higher up covers the one underneath. Even a window at the bottom of the stack can be seen if no other window overlaps it.

A window's place in this ordering is not fixed; in fact, users tend to change the order frequently. Raising a window means moving it "up", to the top of the stack. Lowering a window means moving it to the bottom of the stack. This motion is in the notional third dimension only, and does not change the position of the window on the screen.

You can raise and lower Emacs's X windows with these functions:

Function: **raise-frame** *frame*

This function raises frame frame.

Function: **lower-frame** *frame*

This function lowers frame frame.

You can also specify auto-raise (raising automatically when a frame is selected) or auto-lower (lowering automatically when it is deselected) for any frame using frame parameters. See section X Window Frame

Parameters.

Frame Configurations

Function: **current-frame-configuration**

This function returns a frame configuration list which describes the current arrangement of frames, all their properties, and the window configuration of each one.

Function: **set-frame-configuration** *configuration*

This function restores the state of frames described in configuration.

Mouse Tracking

Sometimes it is useful to track the mouse, which means, to display something to indicate where the mouse is and move the indicator as the mouse moves. For efficient mouse tracking, you need a way to wait until the mouse actually moves.

The convenient way to track the mouse is to ask for events to represent mouse motion. Then you can wait for motion by waiting for an event. In addition, you can easily handle any other sorts of events that may occur. That is useful, because normally you don't want to track the mouse forever--only until some other event, such as the release of a button.

Special Form: **track-mouse** *body...*

Execute body, meanwhile generating input events for mouse motion. The code in body can read these events with `read-event` or `read-key-sequence`. See section Motion Events, for the format of mouse motion events.

The value of `track-mouse` is that of the last form in body.

The usual purpose of tracking mouse motion is to indicate on the screen the consequences of pushing or releasing a button at the current position.

Mouse Position

The new functions `mouse-position` and `set-mouse-position` give access to the current position of the mouse.

Function: **mouse-position**

This function returns a description of the position of the mouse. The value looks like `(frame x . y)`, where x and y are integers giving the position in pixels relative to the top left corner of the inside of frame.

Function: **set-mouse-position** *frame x y*

Thus function warps the mouse to position x, y in frame frame. The arguments x and y are integers, giving the position in pixels relative to the top left corner of the inside of frame.

Warping the mouse means changing the screen position of the mouse as if the user had moved the physical mouse--thus simulating the effect of actual mouse motion.

Pop-Up Menus

Function: **x-popup-menu** *position menu*

This function displays a pop-up menu and returns an indication of what selection the user makes.

The argument position specifies where on the screen to put the menu. It can be either a mouse button event (which says to put the menu where the user actuated the button) or a list of this form:

```
((xoffset yoffset) window)
```

where xoffset and yoffset are positions measured in characters, counting from the top left corner of window's frame.

The argument menu says what to display in the menu. It can be a keymap or a list of keymaps (see section Menu Keymaps). Alternatively, it can have the following form:

```
(title pane1 pane2...)
```

where each pane is a list of form

```
(title (line item)...)
```

Each line should be a string, and each item should be the value to return if that line is chosen.

X Selections

The X server records a set of selections which permit transfer of data between application programs. The various selections are distinguished by selection types, represented in Emacs by symbols. X clients including Emacs can read or set the selection for any given type.

Function: **x-set-selection** *type data*

This function sets a "selection" in the X server. It takes two arguments: a selection type type, and the value to assign to it, data. If data is `nil`, it means to clear out the selection. Otherwise, data may be a string, a symbol, an integer (or a cons of two integers or list of two integers), or a cons of two markers pointing to the same buffer. In the last case, the selection is considered to be the text between the markers. The data may also be a vector of valid non-vector selection values.

Each possible type has its own selection value, which changes independently. The usual values of type are `PRIMARY` and `SECONDARY`; these are symbols with upper-case names, in accord with X Windows

conventions. The default is `PRIMARY`.

Function: **x-get-selection** *type data-type*

This function accesses selections set up by Emacs or by other X clients. It takes two optional arguments, type and data-type. The default for type, the selection type, is `PRIMARY`.

The data-type argument specifies the form of data conversion to use, to convert the raw data obtained from another X client into Lisp data. Meaningful values include `TEXT`, `STRING`, `TARGETS`, `LENGTH`, `DELETE`, `FILE_NAME`, `CHARACTER_POSITION`, `LINE_NUMBER`, `COLUMN_NUMBER`, `OWNER_OS`, `HOST_NAME`, `USER`, `CLASS`, `NAME`, `ATOM`, and `INTEGER`. (These are symbols with upper-case names in accord with X conventions.) The default for data-type is `STRING`.

The X server also has a set of numbered cut buffers which can store text or other data being moved between applications. Cut buffers are considered obsolete, but Emacs supports them for the sake of X clients that still use them.

Function: **x-get-cut-buffer** *n*

This function returns the contents of cut buffer number n.

Function: **x-set-cut-buffer** *string*

This function stores string into the first cut buffer (cut buffer 0), moving the other values down through the series of cut buffers, kill-ring-style.

X Server

This section describes how to access and change the overall status of the X server Emacs is using.

X Connections

You can close the connection with the X server with the function `x-close-current-connection`, and open a new one with `x-open-connection` (perhaps with a different server and display).

Function: **x-close-current-connection**

This function closes the connection to the X server. It deletes all frames, making Emacs effectively inaccessible to the user; therefore, a Lisp program that closes the connection should open another one.

Function: **x-open-connection** *display &optional resource-string*

This function opens a connection to an X server, for use of display display.

The optional argument resource-string is a string of resource names and values, in the same format used in the `` `.Xresources' `` file. The values you specify override the resource values recorded in the X server itself. Here's an example of what this string might look like:

```
"*BorderWidth: 3\n*InternalBorder: 2\n"
```

Function: **x-color-display-p**

This returns `t` if the connected X display has color, and `nil` otherwise.

Function: **x-color-defined-p** *color*

This function reports whether a color name is meaningful and supported on the X display Emacs is using. It returns `t` if the display supports that color; otherwise, `nil`.

Black-and-white displays support just two colors, `"black"` or `"white"`. Color displays support many other colors.

Function: **x-synchronize** *flag*

The function `x-synchronize` enables or disables synchronous communication with the X server. It enables synchronous communication if flag is non-`nil`, and disables it if flag is `nil`.

In synchronous mode, Emacs waits for a response to each X protocol command before doing anything else. This is useful for debugging Emacs, because protocol errors are reported right away, which helps you find the erroneous command. Synchronous mode is not the default because it is much slower.

Resources

Function: **x-get-resource** *attribute &optional name class*

The function `x-get-resource` retrieves a resource value from the X Windows defaults database.

Resources are indexed by a combination of a key and a class. This function searches using a key of the form `instance.attribute', where instance is the name under which Emacs was invoked, and uses `Emacs' as the class.

The optional arguments component and subclass add to the key and the class, respectively. You must specify both of them or neither. If you specify them, the key is `instance.component.attribute', and the class is `Emacs.subclass'.

Rebinding X Server Keys

The X server allows each client to specify what sequence of characters each keyboard key should generate, depending on the set of shift keys held down. Emacs has functions to redefine these sequences in the X server. Redefinitions via `x-rebind-key` apply only to Emacs. Other clients using the same X server are not affected.

Function: **x-rebind-key** *keysym modifiers newstring*

This function redefines a keyboard key in the X server. keysym is a string which conforms to the X keysym definitions found in `` `X11/keysymdef.h' ``, but without the prefix `XK_`. modifiers is either `nil`, meaning no modifier keys, or a list of names of modifier keys, again using the names from `` `X11/keysymdef.h' `` but without the `XK_` prefix.

The third argument, newstring, is the new definition of the key. It is the sequence of characters that the

key should produce as input.

For example,

```
(x-rebind-key "F1" nil "abc")
```

causes the F1 function key to generate the string `"abc"`. Similarly,

```
(x-rebind-key "BackSpace"
              (list "Shift" "Control_L" "c-s-BackSpace")
```

makes the BS key send the string `"c-s-BackSpace"` if either the shift key or the left-hand control key is held down.

Function: **x-rebind-keys** *keysym strings*

This function redefines the complete meaning of a single keyboard key, specifying the behavior for each of the 16 shift masks independently.

The argument keysym specifies the key to rebind, as in `x-rebind-key`.

The argument strings is a list of 16 elements, one for each possible shift mask value; the nth element says how to redefine the key keycode with shift mask value n. If element n is a string, it is the new definition for shift mask n. If element n is `nil`, the definition for shift mask n is unchanged.

Data about the X Server

This section describes functions and a variable that you can use to get information about the capabilities and origin of the X server that Emacs is displaying its frames on.

Function: **x-display-screens**

This function returns the number of screens associated with the current display.

Function: **x-server-version**

This function returns the list of version numbers of the X server in use.

Function: **x-server-vendor**

This function returns the vendor supporting the X server in use.

Function: **x-display-pixel-height**

This function returns the height of this X screen in pixels.

Function: **x-display-mm-height**

This function returns the height of this X screen in millimeters.

Function: **x-display-pixel-width**

This function returns the width of this X screen in pixels.

Function: **x-display-mm-width**

This function returns the width of this X screen in millimeters.

Function: **x-display-backing-store**

This function returns the backing store capability of this screen. Values can be the symbols `always`, `when-mapped`, or `not-useful`.

Function: **x-display-save-under**

This function returns non-`nil` if this X screen supports the SaveUnder feature.

Function: **x-display-planes**

This function returns the number of planes this display supports.

Function: **x-display-visual-class**

This function returns the visual class for this X screen. The value is one of the symbols `static-gray`, `gray-scale`, `static-color`, `pseudo-color`, `true-color`, and `direct-color`.

Function: **x-display-color-p**

This function returns `t` if the X screen in use is a color screen.

Function: **x-display-color-cells**

This function returns the number of color cells this X screen supports.

Variable: **x-no-window-manager**

This variable's value is is `t` if no X window manager is in use.

Go to the previous, next section.

Go to the previous, next section.

Positions

A position is the index of a character in the text of buffer. More precisely, a position identifies the place between two characters (or before the first character, or after the last character), so we can speak of the character before or after a given position. However, the character after a position is often said to be "at" that position.

Positions are usually represented as integers starting from 1, but can also be represented as markers---special objects which relocate automatically when text is inserted or deleted so they stay with the surrounding characters. See section Markers.

Point

Point is a special buffer position used by many editing commands, including the self-inserting typed characters and text insertion functions. Other commands move point through the text to allow editing and insertion at different places.

Like other positions, point designates a place between two characters (or before the first character, or after the last character), rather than a particular character. Many terminals display the cursor over the character that immediately follows point; on such terminals, point is actually before the character on which the cursor sits.

The value of point is a number between 1 and the buffer size plus 1. If narrowing is in effect (see section Narrowing), then point is constrained to fall within the accessible portion of the buffer (possibly at one end of it).

Each buffer has its own value of point, which is independent of the value of point in other buffers. Each window also has a value of point, which is independent of the value of point in other windows on the same buffer. This is why point can have different values in various windows that display the same buffer. When a buffer appears in only one window, the buffer's point and the window's point normally have the same value, so the distinction is rarely important. See section Window Point, for more details.

Function: **point**

This function returns the position of point in the current buffer, as an integer.

```
(point)
     => 175
```

Function: **point-min**

This function returns the minimum accessible value of point in the current buffer. This is 1, unless narrowing is in effect, in which case it is the position of the start of the region that you narrowed to. (See section Narrowing.)

Function: **point-max**

This function returns the maximum accessible value of point in the current buffer. This is `(1+ (buffer-size))`, unless narrowing is in effect, in which case it is the position of the end of the region that you narrowed to. (See section Narrowing).

Function: **buffer-end** *flag*

This function returns `(point-min)` if flag is less than 1, `(point-max)` otherwise. The argument flag must be a number.

Function: **buffer-size**

This function returns the total number of characters in the current buffer. In the absence of any narrowing (see section Narrowing), `point-max` returns a value one larger than this.

```
(buffer-size)
     => 35
(point-max)
     => 36
```

Variable: **buffer-saved-size**

The value of this buffer-local variable is the former length of the current buffer, as of the last time it was read in, saved or auto-saved.

Motion

Motion functions change the value of point, either relative to the current value of point, relative to the beginning or end of the buffer, or relative to the edges of the selected window. See section Point.

Motion by Characters

These functions move point based on a count of characters. `goto-char` is a fundamental primitive because it is the way to move point to a specified position.

Command: **goto-char** *position*

This function sets point in the current buffer to the value position. If position is less than 1, then point is set to the beginning of the buffer. If it is greater than the length of the buffer, then point is set to the end of the buffer.

If narrowing is in effect, then the position is still measured from the beginning of the buffer, but point cannot be moved outside of the accessible portion. Therefore, if position is too small, point is set to the beginning of the accessible portion of the text; if position is too large, point is set to the end.

When this function is called interactively, position is the numeric prefix argument, if provided; otherwise it is read from the minibuffer.

`goto-char` returns position.

Command: **forward-char** *&optional count*

This function moves point forward, towards the end of the buffer, count characters (or backward, towards the beginning of the buffer, if count is negative). If the function attempts to move point past the beginning or end of the buffer (or the limits of the accessible portion, when narrowing is in effect), an error is signaled with error code `beginning-of-buffer` or `end-of-buffer`.

In an interactive call, count is the numeric prefix argument.

Command: **backward-char** *&optional count*

This function moves point backward, towards the beginning of the buffer, count characters (or forward, towards the end of the buffer, if count is negative). If the function attempts to move point past the beginning or end of the buffer (or the limits of the accessible portion, when narrowing is in effect), an error is signaled with error code `beginning-of-buffer` or `end-of-buffer`.

In an interactive call, count is the numeric prefix argument.

Motion by Words

These functions for parsing words use the syntax table to decide whether a given character is part of a word. See section Syntax Tables.

Command: **forward-word** *count*

This function moves point forward count words (or backward if count is negative). Normally it returns `t`. If this motion encounters the beginning or end of the buffer, or the limits of the accessible portion when narrowing is in effect, point stops there and the value is `nil`.

In an interactive call, count is set to the numeric prefix argument.

Command: **backward-word** *count*

This function just like `forward-word`, except that it moves backward until encountering the front of a word, rather than forward.

In an interactive call, count is set to the numeric prefix argument.

This function is rarely used in programs, as it is more efficient to call `forward-word` with negative argument.

Variable: **words-include-escapes**

This variable affects the behavior of `forward-word` and everything that uses it. If it is non-`nil`, then characters in the "escape" and "character quote" syntax classes count as part of words. Otherwise, they do not.

Motion to an End of the Buffer

To move point to the beginning of the buffer, write:

```
(goto-char (point-min))
```

Likewise, to move to the end of the buffer, use:

```
(goto-char (point-max))
```

Here are two commands which users use to do these things. They are documented here to warn you not to use them in Lisp programs, because they set the mark and display messages in the echo area.

Command: **beginning-of-buffer** *&optional n*

This function moves point to the beginning of the buffer (or the limits of the accessible portion, when narrowing is in effect), setting the mark at the previous position. If n is non-`nil`, then it puts point n tenths of the way from the beginning of the buffer.

In an interactive call, n is the numeric prefix argument, if provided; otherwise n defaults to `nil`.

Don't use this function in Lisp programs!

Command: **end-of-buffer** *&optional n*

This function moves point to the end of the buffer (or the limits of the accessible portion, when narrowing is in effect), setting the mark at the previous position. If n is non-`nil`, then it puts point n tenths of the way from the end.

In an interactive call, n is the numeric prefix argument, if provided; otherwise n defaults to `nil`.

Don't use this function in Lisp programs!

Motion by Text Lines

Text lines are portions of the buffer delimited by newline characters, which are regarded as part of the previous line. The first text line begins at the beginning of the buffer, and the last text line ends at the end of the buffer whether or not the last character is a newline. The division of the buffer into text lines is not affected by the width of the window, or by how tabs and control characters are displayed.

Command: **goto-line** *line*

This function moves point to the front of the lineth line, counting from line 1 at beginning of buffer. If line is less than 1, then point is set to the beginning of the buffer. If line is greater than the number of lines in the buffer, then point is set to the *end of the last line* of the buffer.

If narrowing is in effect, then line still counts from the beginning of the buffer, but point cannot go outside the accessible portion. So point is set at the beginning or end of the accessible portion of the text if the line number specifies a position that is inaccessible.

The return value of `goto-line` is the difference between line and the line number of the line to wh point actually was able move (before taking account of any narrowing). Thus, the value is positive if scan encounters the end of the buffer.

In an interactive call, line is the numeric prefix argument if one has been provided. Otherwise line is read in the minibuffer.

Command: **beginning-of-line** *&optional count*

This function moves point to the beginning of the current line. With an argument count not `nil` or 1, it moves forward count-1 lines and then to the beginning of the line.

If this function reaches the end of the buffer (or of the accessible portion, if narrowing is in effect), it positions point at the beginning of the last line. No error is signaled.

Command: **end-of-line** *&optional count*

This function moves point to the end of the current line. With an argument count not `nil` or 1, it moves forward count-1 lines and then to the end of the line.

If this function reaches the end of the buffer (or of the accessible portion, if narrowing is in effect), it positions point at the end of the last line. No error is signaled.

Command: **forward-line** *&optional count*

This function moves point forward count lines, to the beginning of the line. If count is negative, it moves point -count lines backward, to the beginning of the line.

If the beginning or end of the buffer (or of the accessible portion) is encountered before that many lines are found, then point stops at the beginning or end. No error is signaled.

`forward-line` returns the difference between count and the number of lines actually moved. If you attempt to move down five lines from the beginning of a buffer that has only three lines, point will positioned at the end of the last line, and the value will be 2.

In an interactive call, count is the numeric prefix argument.

Function: **count-lines** *start end*

This function returns the number of lines between the positions start and end in the current buffer. If start and end are equal, then it returns 0. Otherwise it returns at least 1, even if start and end are on the same line. This is because the text between them, considered in isolation, must contain at least one line unless it is empty.

Here is an example of using `count-lines`:

```
(defun current-line ()
  "Return the vertical position of point
in the selected window.  Top line is 0.
Counts each text line only once, even if it wraps."
  (+ (count-lines (window-start) (point))
```


```
    (if (= (current-column) 0) 1 0)
    -1))
```

Also see the functions `bolp` and `eolp` in section Examining Text Near Point. These functions do not move point, but test whether it is already at the beginning or end of a line.

Motion by Screen Lines

The line functions in the previous section count text lines, delimited only by newline characters. By contrast, these functions count screen lines, which are defined by the way the text appears on the screen. A text line is a single screen line if it is short enough to fit the width of the selected window, but otherwise it may occupy several screen lines.

In some cases, text lines are truncated on the screen rather than continued onto additional screen lines. Then `vertical-motion` moves point just like `forward-line`. See section Truncation.

Because the width of a given string depends on the flags which control the appearance of certain characters, `vertical-motion` will behave differently on a given piece of text found in different buffers. It will even act differently in different windows showing the same buffer, because the width may differ and so may the truncation flag. See section Usual Display Conventions.

Function: **vertical-motion** *count*

This function moves point to the start of the screen line count screen lines down from the screen line containing point. If count is negative, it moves up instead.

This function returns the number of lines moved. The value may be less in absolute value than count if the beginning or end of the buffer was reached.

Command: **move-to-window-line** *count*

This function moves point with respect to the text currently displayed in the selected window. Point is moved to the beginning of the screen line count screen lines from the top of the window. If count is negative, point moves either to the beginning of the line -count lines from the bottom or else to the last line of the buffer if the buffer ends above the specified screen position.

If count is `nil`, then point moves to the beginning of the line in the middle of the window. If the absolute value of count is greater than the size of the window, then point moves to the place which would appear on that screen line if the window were tall enough. This will probably cause the next redisplay to scroll to bring that location onto the screen.

In an interactive call, count is the numeric prefix argument.

The value returned is the window line number, with the top line in the window numbered 0.

The User-Level Vertical Motion Commands

A goal column is useful if you want to edit text such as a table in which you want to move point to a certain column on each line. The goal column affects the vertical text line motion commands, `next-line` and `previous-line`. See section 'Basic Editing Commands' in The GNU Emacs Manual.

User Option: **goal-column**

This variable holds an explicitly specified goal column for vertical line motion commands. If it is an integer, it specifies a column, and these commands try to move to that column on each line. If it is `nil`, then the commands set their own goal columns. Any other value is invalid.

Variable: **temporary-goal-column**

This variable holds the temporary goal column during a sequence of consecutive vertical line motion commands. It is overridden by `goal-column` if that is non-`nil`. It is set each time a vertical motion command is invoked, unless the previous command was also a vertical motion command.

User Option: **track-eol**

This variable controls how the vertical line motion commands operate when starting at the end of a line. If `track-eol` is non-`nil`, then vertical motion starting at the end of a line will keep to the ends of lines. This means moving to the end of each line moved onto. The value of `track-eol` has no effect if point is not at the end of a line when the first vertical motion command is given.

`track-eol` has its effect by causing `temporary-goal-column` to be set to 9999 instead of to the current column.

Command: **set-goal-column** *unset*

This command sets the variable `goal-column` to specify a permanent goal column for the vertical line motion commands. If unset is `nil`, then `goal-column` is set to the current column of point. If unset is non-`nil`, then `goal-column` is set to `nil`.

This function is intended for interactive use; and in an interactive call, unset is the raw prefix argument.

Moving over Balanced Expressions

Here are several functions concerned with balanced-parenthesis expressions (also called sexps in connection with moving across them in Emacs). The syntax table controls how these functions interpret various characters; see section Syntax Tables. See section Parsing Balanced Expressions, for lower-level primitives for scanning sexps or parts of sexps. For user-level commands, see section 'Lists and Sexps' in GNU Emacs Manual.

Command: **forward-list** *arg*

Move forward across arg balanced groups of parentheses. (Other syntactic entities such as words or paired string quotes are ignored.)

Command: **backward-list** *arg*

Move backward across arg balanced groups of parentheses. (Other syntactic entities such as words or paired string quotes are ignored.)

Command: **up-list** *arg*

Move forward out of arg levels of parentheses. A negative argument means move backward but still to a less deep spot.

Command: **down-list** *arg*

Move forward down arg levels of parentheses. A negative argument means move backward but still go down arg level.

Command: **forward-sexp** *arg*

Move forward across arg balanced expressions. Balanced expressions include both those delimited by parentheses and other kinds, such as words and string constants. For example,

```
---------- Buffer: foo ----------
(concat-!- "foo " (car x) y z)
---------- Buffer: foo ----------

(forward-sexp 3)
     => nil

---------- Buffer: foo ----------
(concat "foo " (car x) y-!- z)
---------- Buffer: foo ----------
```

Command: **backward-sexp** *arg*

Move backward across arg balanced expressions.

Skipping Characters

The following two functions move point over a specified set of characters. For example, they are often used to skip whitespace. For related functions, see section Motion and Syntax.

Function: **skip-chars-forward** *character-set &optional limit*

This function moves point in the current buffer forward, skipping over a given set of characters. Emacs first examines the character following point; if it matches character-set, then point is advanced and the next character is examined. This continues until a character is found that does not match. The function returns `nil`.

The argument character-set is like the inside of a `[...]' in a regular expression except that `]' is never special and `\' quotes `^', `-' or `\'. Thus, `"a-zA-Z"` skips over all letters, stopping before the first

nonletter, and "^a-zA-Z" skips nonletters stopping before the first letter. See section Regular Expressions.

If limit is supplied (it must be a number or a marker), it specifies the maximum position in the buffer that point can be skipped to. Point will stop at or before limit.

In the following example, point is initially located directly before the `T'. After the form is evaluated, point is located at the end of that line (between the `t' of `hat' and the newline). The function skips all letters and spaces, but not newlines.

```
---------- Buffer: foo ----------
I read "-!-The cat in the hat
comes back" twice.
---------- Buffer: foo ----------

(skip-chars-forward "a-zA-Z ")
     => nil

---------- Buffer: foo ----------
I read "The cat in the hat-!-
comes back" twice.
---------- Buffer: foo ----------
```

Function: **skip-chars-backward** *character-set &optional limit*

This function moves point backward, skipping characters that match character-set. It just like `skip-chars-forward` except for the direction of motion.

Excursions

It is often useful to move point "temporarily" within a localized portion of the program, or to switch buffers temporarily. This is called an excursion, and it is done with the `save-excursion` special form. This construct saves the current buffer and its values of point and the mark so they can be restored after the completion of the excursion.

The forms for saving and restoring the configuration of windows are described elsewhere (see section Window Configurations, and see section Frame Configurations).

Special Form: **save-excursion** *forms...*

The `save-excursion` special form saves the identity of the current buffer and the values of point and the mark in it, evaluates forms, and finally restores the buffer and its saved values of point and the mark. All three saved values are restored even in case of an abnormal exit via throw or error (see section Nonlocal Exits).

The `save-excursion` special form is the standard way to switch buffers or move point within one part of a program and avoid affecting the rest of the program. It is used more than 500 times in the Lisp

sources of Emacs.

The values of point and the mark for other buffers are not saved by `save-excursion`, so any changes made to point and the mark in the other buffers will remain in effect after `save-excursion` exits.

Likewise, `save-excursion` does not restore window-buffer correspondences altered by functions such as `switch-to-buffer`. One way to restore these correspondences, and the selected window, is to use `save-window-excursion` inside `save-excursion` (see section Window Configurations).

The value returned by `save-excursion` is the result of the last of forms, or `nil` if no forms are given.

```
(save-excursion
  forms)
==
(let ((old-buf (current-buffer))
      (old-pnt (point-marker))
      (old-mark (copy-marker (mark-marker))))
  (unwind-protect
      (progn forms)
    (set-buffer old-buf)
    (goto-char old-pnt)
    (set-marker (mark-marker) old-mark)))
```

Narrowing

Narrowing means limiting the text addressable by Emacs editing commands to a limited range of characters in a buffer. The text that remains addressable is called the accessible portion of the buffer.

Narrowing is specified with two buffer positions which become the beginning and end of the accessible portion. For most editing commands these positions replace the values of the beginning and end of the buffer. While narrowing is in effect, no text outside the accessible portion is displayed, and point cannot move outside the accessible portion.

Values such as positions or line numbers which usually count from the beginning of the buffer continue to do so, but the functions which use them will refuse to operate on text that is inaccessible.

The commands for saving buffers are unaffected by narrowing; the entire buffer is saved regardless of the any narrowing.

Command: **narrow-to-region** *start end*

This function sets the accessible portion of the current buffer to start at start and end at end. Both arguments should be character positions.

In an interactive call, start and end are set to the bounds of the current region (point and the mark, with the smallest first).

Command: **narrow-to-page** *move-count*

This function sets the accessible portion of the current buffer to include just the current page. An optional first argument move-count non-`nil` means to move forward or backward by move-count pages and then narrow.

In an interactive call, move-count is set to the numeric prefix argument.

Command: **widen**

This function cancels any narrowing in the current buffer, so that the entire contents are accessible. This is called widening. It is equivalent to the following expression:

```
(narrow-to-region 1 (1+ (buffer-size)))
```

Special Form: **save-restriction** *body...*

This special form saves the current bounds of the accessible portion, evaluates the body forms, and finally restores the saved bounds, thus restoring the same state of narrowing (or absence thereof) formerly in effect. The state of narrowing is restored even in the event of an abnormal exit via throw or error (see section Nonlocal Exits). Therefore, this construct is a clean way to narrow a buffer temporarily.

The value returned by `save-restriction` is that returned by the last form in body, or `nil` if no body forms were given.

Caution: it is easy to make a mistake when using the `save-restriction` function. Read the entire description here before you try it.

If body changes the current buffer, `save-restriction` still restores the restrictions on the original buffer (the buffer they came from), but it does not restore the identity of the current buffer.

Point and the mark are *not* restored by this special form; use `save-excursion` for that. If you use both `save-restriction` and `save-excursion` together, `save-excursion` should come first (on the outside). Otherwise, the old point value would be restored with temporary narrowing still in effect. If the old point value were outside the limits of the temporary narrowing, this would fail to restore it accurately.

The `save-restriction` special form records the values of the beginning and end of the accessible portion as distances from the beginning and end of the buffer. In other words, it records the amount of inaccessible text before and after the accessible portion.

This technique yields correct results if body does further narrowing. However, `save-restriction` can become confused if they widen and then make changes outside the area of the saved narrowing. When this is what you want to do, `save-restriction` is not the right tool for the job. Here is what you must use instead:

```
(let ((beg (point-min-marker))
      (end (point-max-marker)))
```


```
  (unwind-protect
      (progn body)
    (save-excursion
      (set-buffer (marker-buffer beg))
      (narrow-to-region beg end))))
```

Here is a simple example of correct use of `save-restriction`:

```
---------- Buffer: foo ----------
This is the contents of foo
This is the contents of foo
This is the contents of foo-!-
---------- Buffer: foo ----------

(save-excursion
  (save-restriction
    (goto-char 1)
    (forward-line 2)
    (narrow-to-region 1 (point))
    (goto-char (point-min))
    (replace-string "foo" "bar")))

---------- Buffer: foo ----------
This is the contents of bar
This is the contents of bar
This is the contents of foo-!-
---------- Buffer: foo ----------
```

Go to the previous, next section.

Go to the previous, next section.

Markers

A marker is a Lisp object used to specify a position in a buffer relative to the surrounding text. A marker changes its offset from the beginning of the buffer automatically whenever text is inserted or deleted, so that it stays with the two characters on either side of it.

Overview of Markers

A marker specifies a buffer and a position in that buffer. The marker can be used to represent a position in the functions that require one, just as an integer could be used. See section Positions, for a complete description of positions.

A marker has two attributes: the marker position, and the marker buffer. The marker position is an integer which is equivalent (at the moment) to the marker as a position in that buffer; however, as text is inserted or deleted in the buffer, the marker is relocated, so that its integer equivalent changes. The idea is that a marker positioned between two characters in a buffer will remain between those two characters despite any changes made to the contents of the buffer; thus, a marker's offset from the beginning of a buffer may change often during the life of the marker.

If the text around a marker is deleted, the marker is repositioned between the characters immediately before and after the deleted text. If text is inserted at the position of a marker, the marker remains in front of the new text unless it is inserted with `insert-before-markers` (see section Insertion). When text is inserted or deleted somewhere before the marker position (not next to the marker), the marker moves back and forth with the two neighboring characters.

When a buffer is modified, all of its markers must be checked so that they can be relocated if necessary. This slows processing in a buffer with a large number of markers. For this reason, it is a good idea to make a marker point nowhere if you are sure you don't need it any more. Unreferenced markers will eventually be garbage collected, but until then will continue to be updated if they do point somewhere.

Because it is quite common to perform arithmetic operations on a marker position, most of the arithmetic operations (including `+` and `-`) accept markers as arguments. In such cases, the current position of the marker is used.

Here are examples of creating markers, setting markers, and moving point to markers:

```
;; Make a new marker that initially does not point anywhere:
(setq m1 (make-marker))
     => #<marker in no buffer>

;; Set m1 to point between the 100th and 101st characters
;;   in the current buffer:
(set-marker m1 100)
```


```
     => #<marker at 100 in markers.texi>

;; Now insert one character at the beginning of the buffer:
(goto-char (point-min))
     => 1
(insert "Q")
     => nil

;; m1 is updated appropriately.
m1
     => #<marker at 101 in markers.texi>

;; Two markers that point to the same position
;;   are not eq, but they are equal.
(setq m2 (copy-marker m1))
     => #<marker at 101 in markers.texi>
(eq m1 m2)
     => nil
(equal m1 m2)
     => t

;; When you are finished using a marker, make it point nowhere.
(set-marker m1 nil)
     => #<marker in no buffer>
```

Predicates on Markers

You can test an object to see whether it is a marker, or whether it is either an integer or a marker. The latter test is useful when you are using the arithmetic functions that work with both markers and integers.

Function: **markerp** *object*

This function returns `t` if object is a marker, `nil` otherwise. In particular, integers are not markers, even though many functions will accept either a marker or an integer.

Function: **integer-or-marker-p** *object*

This function returns `t` if object is an integer or a marker, `nil` otherwise.

Function: **number-or-marker-p** *object*

This function returns `t` if object is a number (of any type) or a marker, `nil` otherwise.

Functions That Create Markers

When you create a new marker, you can make it point nowhere, or point to the present position of point, or to the beginning or end of the accessible portion of the buffer, or to the same place as another given marker.

Function: **make-marker**

This functions returns a newly allocated marker that does not point anywhere.

```
(make-marker)
     => #<marker in no buffer>
```

Function: **point-marker**

This function returns a new marker that points to the present position of point in the current buffer. See section Point. For an example, see `copy-marker`, below.

Function: **point-min-marker**

This function returns a new marker that points to the beginning of the accessible portion of the buffer. This will be the beginning of the buffer unless narrowing is in effect. See section Narrowing.

Function: **point-max-marker**

This function returns a new marker that points to the end of the accessible portion of the buffer. This will be the end of the buffer unless narrowing is in effect. See section Narrowing.

Here are examples of this function and `point-min-marker`, shown in a buffer containing a version of the source file for the text of this chapter.

```
(point-min-marker)
     => #<marker at 1 in markers.texi>
(point-max-marker)
     => #<marker at 15573 in markers.texi>

(narrow-to-region 100 200)
     => nil
(point-min-marker)
     => #<marker at 100 in markers.texi>
(point-max-marker)
     => #<marker at 200 in markers.texi>
```

Function: **copy-marker** *marker-or-integer*

If passed a marker as its argument, `copy-marker` returns a new marker that points to the same place and the same buffer as does marker-or-integer. If passed an integer as its argument, `copy-marker` returns a new marker that points to position marker-or-integer in the current buffer.

If passed an argument that is an integer whose value is less than 1, `copy-marker` returns a new marker that points to the beginning of the current buffer. If passed an argument that is an integer whose value is greater than the length of the buffer, then `copy-marker` returns a new marker that points to the end of the buffer.

An error is signaled if marker is neither a marker nor an integer.

```
(setq p (point-marker))
     => #<marker at 2139 in markers.texi>

(setq q (copy-marker p))
     => #<marker at 2139 in markers.texi>

(eq p q)
     => nil

(equal p q)
     => t

(copy-marker 0)
     => #<marker at 1 in markers.texi>

(copy-marker 20000)
     => #<marker at 7572 in markers.texi>
```

Information from Markers

This section describes the functions for accessing the components of a marker object.

Function: **marker-position** *marker*

This function returns the position that marker points to, or `nil` if it points nowhere.

Function: **marker-buffer** *marker*

This function returns the buffer that marker points into, or `nil` if it points nowhere.

```
(setq m (make-marker))
     => #<marker in no buffer>
(marker-position m)
     => nil
(marker-buffer m)
     => nil

(set-marker m 3770 (current-buffer))
     => #<marker at 3770 in markers.texi>
```


```
(marker-buffer m)
     => #<buffer markers.texi>
(marker-position m)
     => 3770
```

Two distinct markers will be found `equal` (even though not `eq`) to each other if they have the same position and buffer, or if they both point nowhere.

Changing Markers

This section describes how to change the position of an existing marker. When you do this, be sure you know whether the marker is used outside of your program, and, if so, what effects will result from moving it--otherwise, confusing things may happen in other parts of Emacs.

Function: **set-marker** *marker position &optional buffer*

This function moves marker to position in buffer. If buffer is not provided, it defaults to the current buffer.

If position is less than 1, `set-marker` moves marker to the beginning of the buffer. If the value of position is greater than the size of the buffer, `set-marker` moves marker to the end of the buffer. If position is `nil` or a marker that points nowhere, then marker is set to point nowhere.

The value returned is marker.

```
(setq m (point-marker))
     => #<marker at 4714 in markers.texi>
(set-marker m 55)
     => #<marker at 55 in markers.texi>
(setq b (get-buffer "foo"))
     => #<buffer foo>
(set-marker m 0 b)
     => #<marker at 1 in foo>
```

Function: **move-marker** *marker position &optional buffer*

This is another name for `set-marker`.

The Mark

A special marker in each buffer is designated the mark. It records a position for the user for the sake of commands such as C-w and C-x TAB. Lisp programs should set the mark only to values that have a potential use to the user, and never for their own internal purposes. For example, the `replace-regexp` command sets the mark to the value of point before doing any replacements, because this enables the user to move back there conveniently after the replace is finished.

Many commands are designed so that when called interactively they operate on the text between point

and the mark. If you are writing such a command, don't examine the mark directly; instead, use `interactive` with the `r' specification. This will provide the values of point and the mark as arguments to the command in an interactive call, but will permit other Lisp programs to specify arguments explicitly. See section Code Characters for `interactive`.

Each buffer has its own value of the mark that is independent of the value of the mark in other buffers. When a buffer is created, the mark exists but does not point anywhere. We consider this state as "the absence of a mark in that buffer".

Once the mark "exists" in a buffer, it normally never ceases to exist. However, it may become inactive, if Transient Mark mode is enabled. The variable `mark-active`, which is always local in all buffers, indicates whether the mark is active: non-`nil` means yes. A command can request deactivation of the mark upon return to the editor command loop by setting `deactivate-mark` to a non-`nil` value (but this deactivation only follows if Transient Mark mode is enabled).

The main motivation for using Transient Mark mode is that this mode also enables highlighting of the region when the mark is active. See section Emacs Display.

In addition to the mark, each buffer has a mark ring which is a list of markers that are the previous values of the mark. When editing commands change the mark, they should normally save the old value of the mark on the mark ring. The mark ring may contain no more than the maximum number of entries specified by the variable `mark-ring-max`; excess entries are discarded on a first-in-first-out basis.

Function: **mark** *&optional force*

This function returns the position of the current buffer's mark as an integer.

Normally, if the mark is inactive `mark` signals an error. However, if force is non-`nil`, then it returns the mark position anyway--or `nil`, if the mark is not yet set for this buffer.

Function: **mark-marker**

This function returns the current buffer's mark. This is the very marker which records the mark location inside Emacs, not a copy. Therefore, changing this marker's position will directly affect the position of the mark. Don't do it unless that is the effect you want.

```
(setq m (mark-marker))
     => #<marker at 3420 in markers.texi>
(set-marker m 100)
     => #<marker at 100 in markers.texi>
(mark-marker)
     => #<marker at 100 in markers.texi>
```

Like any marker, this marker can be set to point at any buffer you like. We don't recommend that you make it point at any buffer other than the one of which it is the mark. If you do, it will yield perfectly consistent, if rather odd, results.

Function: **set-mark** *position*

This function sets the mark to position, and activates the mark. The old value of the mark is *not* pushed onto the mark ring.

Please note: use this function only if you want the user to see that the mark has moved, and you want the previous mark position to be lost. Normally, when a new mark is set, the old one should go on the `mark-ring`. For this reason, most applications should use `push-mark` and `pop-mark`, not `set-mark`.

Novice Emacs Lisp programmers often try to use the mark for the wrong purposes. The mark saves a location for the user's convenience. An editing command should not alter the mark unless altering the mark is part of the user-level functionality of the command. (And, in that case, this effect should be documented.) To remember a location for internal use in the Lisp program, store it in a Lisp variable. For example:

```
(let ((beg (point)))
  (forward-line 1)
  (delete-region beg (point))).
```

Variable: **mark-ring**

The value of this buffer-local variable is the list of saved former marks of the current buffer, most recent first.

```
mark-ring
=> (#<marker at 11050 in markers.texi>
    #<marker at 10832 in markers.texi>
    ...)
```

User Option: **mark-ring-max**

The value of this variable is the maximum size of `mark-ring`. If more marks than this are pushed onto the `mark-ring`, it discards marks on a first-in, first-out basis.

Function: **push-mark** *&optional position nomsg activate*

This function sets the current buffer's mark to position, and pushes a copy of the previous mark onto `mark-ring`. If position is `nil`, then the value of point is used. `push-mark` returns `nil`.

The function `push-mark` normally *does not* activate the mark. To do that, specify `t` for the argument activate.

A `Mark set' message is displayed unless nomsg is non-`nil`.

Function: **pop-mark**

This function pops off the top element of `mark-ring` and makes that mark become the buffer's actual mark. This does not change the buffer's point, and does nothing if `mark-ring` is empty. It deactivates the mark.

The return value is not useful.

User Option: **transient-mark-mode**

This variable enables Transient Mark mode, in which every buffer-modifying primitive sets `deactivate-mark`. The consequence of this is that commands that modify the buffer normally cause the mark to become inactive.

Variable: **deactivate-mark**

If an editor command sets this variable non-`nil`, then the editor command loop deactivates the mark after the command returns.

Variable: **mark-active**

The mark is active when this variable is non-`nil`. This variable is always local in each buffer.

Variable: **activate-mark-hook**

Variable: **deactivate-mark-hook**

These normal hooks are run, respectively, when the mark becomes active and when it becomes inactive. The hook `activate-mark-hook` is also run at the end of a command if the mark is active and the region may have changed.

The Region

The text between point and the mark is known as the region. Various functions operate on text delimited by point and the mark, but only those functions specifically related to the region itself are described here.

Function: **region-beginning**

This function returns the position of the beginning of the region (as an integer). This is the position of either point or the mark, whichever is smaller.

If the mark does not point anywhere, an error is signaled.

Function: **region-end**

This function returns the position of the end of the region (as an integer). This is the position of either point or the mark, whichever is larger.

If the mark does not point anywhere, an error is signaled.

Few programs need to use the `region-beginning` and `region-end` functions. A command designed to operate on a region should instead use `interactive` with the `r' specification, so that the same function can be called with explicit bounds arguments from programs. (See section Code Characters for `interactive`.)

Go to the previous, next section.

Go to the previous, next section.

Text

This chapter describes the functions that deal with the text in a buffer. Most examine, insert or delete text in the current buffer, often in the vicinity of point. Many are interactive. All the functions that change the text provide for undoing the changes (see section Undo).

Many text-related functions operate on a region of text defined by two buffer positions passed in arguments named start and end. These arguments should be either markers (see section Markers) or or numeric character positions (see section Positions). The order of these arguments does not matter; it is all right for start to be the end of the region and end the beginning. For example, `(delete-region 1 10)` and `(delete-region 10 1)` perform identically. An `args-out-of-range` error is signaled if either start or end is outside the accessible portion of the buffer. In an interactive call, point and the mark are used for these arguments.

Throughout this chapter, "text" refers to the characters in the buffer.

Examining Text Near Point

Many functions are provided to look at the characters around point. Several simple functions are described here. See also `looking-at` in section Regular Expression Searching.

Function: **char-after** *position*

This function returns the character in the current buffer at (i.e., immediately after) position position. If position is out of range for this purpose, either before the beginning of the buffer, or at or beyond the end, then the value is `nil`.

Remember that point is always between characters, and the terminal cursor normally appears over the character following point. Therefore, the character returned by `char-after` is the character the cursor is over.

In the following example, assume that the first character in the buffer is `@':

```
(char-to-string (char-after 1))
     => "@"
```

Function: **following-char**

This function returns the character following point in the current buffer. This is similar to `(char-after (point))`. However, if point is at the end of the buffer, then the result of `following-char` is 0.

In this example, point is between the `a' and the `c'.


```
---------- Buffer: foo ----------
Gentlemen may cry "Pea-!-ce! Peace!,"
but there is no peace.
---------- Buffer: foo ----------

(char-to-string (preceding-char))
     => "a"
(char-to-string (following-char))
     => "c"
```

Function: **preceding-char**

This function returns the character preceding point in the current buffer. See above, under `following-char`, for an example. If point is at the beginning of the buffer, then the result of `preceding-char` is 0.

Function: **bobp**

This function returns `t` if point is at the beginning of the buffer. If narrowing is in effect, this means the beginning of the accessible portion of the text. See also `point-min` in section Point.

Function: **eobp**

This function returns `t` if point is at the end of the buffer. If narrowing is in effect, this means the end of accessible portion of the text. See also `point-max` in See section Point.

Function: **bolp**

This function returns `t` if point is at the beginning of a line. See section Motion by Text Lines.

Function: **eolp**

This function returns `t` if point is at the end of a line. The end of the buffer is always considered the end of a line.

Examining Buffer Contents

This section describes two functions that allow a Lisp program to convert any portion of the text in the buffer into a string.

Function: **buffer-substring** *start end*

This function returns a string containing a copy of the text of the region defined by positions start and end in the current buffer. If the arguments are not positions in the accessible portion of the buffer, Emacs signals an `args-out-of-range` error.

It is not necessary for start to be less than end; the arguments can be given in either order. But most often the smaller argument is written first.


```
---------- Buffer: foo ----------
This is the contents of buffer foo

---------- Buffer: foo ----------

(buffer-substring 1 10)
=> "This is t"
(buffer-substring (point-max) 10)
=> "he contents of buffer foo
"
```

Function: **buffer-string**

This function returns the contents of the accessible portion of the current buffer as a string. This is the portion between `(point-min)` and `(point-max)` (see section Narrowing).

```
---------- Buffer: foo ----------
This is the contents of buffer foo

---------- Buffer: foo ----------

(buffer-string)
     => "This is the contents of buffer foo
"
```

Comparing Text

This function lets you compare portions of the text in a buffer, without copying them into strings first.

Function: **compare-buffer-substrings** *buffer1 start1 end1 buffer2 start2 end2*

This function lets you compare two substrings of the same buffer or two different buffers. The first three arguments specify one substring, giving a buffer and two positions within the buffer. The last three arguments specify the other substring in the same way. You can use `nil` for buffer1, buffer2 or both to stand for the current buffer.

The value is negative if the first substring is less, positive if the first is greater, and zero if they are equal. The absolute value of the result is one plus the index of the first differing characters within the substrings.

This function ignores case when comparing characters if `case-fold-search` is non-`nil`.

Suppose the current buffer contains the text `foobarbar haha!rara!'; then in this example the two substrings are `rbar ' and `rara!'. The value is 2 because the first substring is greater at the second character.

```
(compare-buffer-substring nil 6 11 nil 16 21)
```


```
=> 2
```

This function does not exist in Emacs version 18 and earlier.

Insertion

Insertion takes place at point. Markers pointing at positions after the insertion point are relocated with the surrounding text (see section Markers). When a marker points at the place of insertion, it is normally not relocated, so that it points to the beginning of the inserted text; however, when `insert-before-markers` is used, all such markers are relocated to point after the inserted text.

Point may end up either before or after inserted text, depending on the function used. If point is left after the inserted text, we speak of insertion before point.

Each of these functions signals an error if the current buffer is read-only.

Function: **insert** *&rest args*

This function inserts the strings and/or characters args into the current buffer, at point, moving point forward. An error is signaled unless all args are either strings or characters. The value is `nil`.

Function: **insert-before-markers** *&rest args*

This function inserts the strings and/or characters args into the current buffer, at point, moving point forward. An error is signaled unless all args are either strings or characters. The value is `nil`.

This function is unlike the other insertion functions in that a marker whose position initially equals point is relocated to come after the newly inserted text.

Function: **insert-char** *character count*

This function inserts count instances of character into the current buffer before point. count must be a number, and character must be a character. The value is `nil`.

Function: **insert-buffer-substring** *from-buffer-or-name &optional start end*

This function inserts a substring of the contents of buffer from-buffer-or-name (which must already exist) into the current buffer before point. The text inserted consists of the characters in the region defined by start and end (These arguments default to the beginning and end of the accessible portion of that buffer). The function returns `nil`.

In this example, the form is executed with buffer `bar' as the current buffer. We assume that buffer `bar' is initially empty.

```
---------- Buffer: foo ----------
We hold these truths to be self-evident, that all
---------- Buffer: foo ----------

(insert-buffer-substring "foo" 1 20)
```


```
     => nil

---------- Buffer: bar ----------
We hold these truth
---------- Buffer: bar ----------
```

User-Level Insertion Commands

This section describes higher-level commands for inserting text, commands intended primarily for the user but useful also in Lisp programs.

Command: **insert-buffer** *from-buffer-or-name*

This function inserts the entire contents of from-buffer-or-name (which must exist) into the current buffer after point. It leaves the mark after the inserted text. The value is `nil`.

Command: **self-insert-command** *count*

This function inserts the last character typed count times and returns `nil`. Most printing characters are bound to this command. In routine use, `self-insert-command` is the most frequently called function in Emacs, but programs rarely use it except to install it on a keymap.

In an interactive call, count is the numeric prefix argument.

This function calls `auto-fill-function` if the current column number is greater than the value of `fill-column` and the character inserted is a space (see section Auto Filling).

This function performs abbrev expansion if Abbrev mode is enabled and the inserted character does not have word-constituent syntax. (See section Abbrevs And Abbrev Expansion, and section Table of Syntax Classes.)

This function is also responsible for calling `blink-paren-function` when the inserted character has close parenthesis syntax (see section Blinking).

Command: **newline** *&optional number-of-newlines*

This function inserts newlines into the current buffer before point. If number-of-newlines is supplied, that many newline characters are inserted.

In Auto Fill mode, `newline` can break the preceding line if number-of-newlines is not supplied. When this happens, it actually inserts two newlines at different places: one at point, and another earlier in the line. `newline` does not auto-fill if number-of-newlines is non-`nil`.

The value returned is `nil`. In an interactive call, count is the numeric prefix argument.

Command: **split-line**

This function splits the current line, moving the portion of the line after point down vertically, so that it is on the next line directly below where it was before. Whitespace is inserted as needed at the beginning of

the lower line, using the `indent-to` function. `split-line` returns the position of point.

Programs hardly ever use this function.

Variable: **overwrite-mode**

This variable controls whether overwrite mode is in effect: a non-`nil` value enables the mode. It is automatically made buffer-local when set in any fashion.

Deletion of Text

All of the deletion functions operate on the current buffer, and all return a value of `nil`. In addition to these functions, you can also delete text using the "kill" functions that save it in the kill ring; some of these functions save text in the kill ring in some cases but not in the usual case. See section The Kill Ring.

Function: **erase-buffer**

This function deletes the entire text of the current buffer, leaving it empty. If the buffer is read-only, it signals a `buffer-read-only` error. Otherwise, it deletes the text without asking for any confirmation. The value is always `nil`.

Normally, deleting a large amount of text from a buffer inhibits further auto-saving of that buffer "because it has shrunk". However, `erase-buffer` does not do this, the idea being that the future text is not really related to the former text, and its size should not be compared with that of the former text.

Command: **delete-region** *start end*

This function deletes the text in the current buffer in the region defined by start and end. The value is `nil`.

Command: **delete-char** *count &optional killp*

This function deletes count characters directly after point, or before point if count is negative. If killp is non-`nil`, then it saves the deleted characters in the kill ring.

In an interactive call, count is the numeric prefix argument, and killp is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

Command: **delete-backward-char** *count &optional killp*

This function deletes count characters directly before point, or after point if count is negative. If killp is non-`nil`, then it saves the deleted characters in the kill ring.

In an interactive call, count is the numeric prefix argument, and killp is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

Command: **backward-delete-char-untabify** *count &optional killp*

This function deletes count characters backward, changing tabs into spaces. When the next character to be deleted is a tab, it is first replaced with the proper number of spaces to preserve alignment and then one of those spaces is deleted instead of the tab. If killp is non-`nil`, then the command saves the deleted characters in the kill ring.

If count is negative, then tabs are not changed to spaces, and the characters are deleted by calling `delete-backward-char` with count.

In an interactive call, count is the numeric prefix argument, and killp is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

User-Level Deletion Commands

This section describes higher-level commands for deleting text, commands intended primarily for the user but useful also in Lisp programs.

Command: **delete-horizontal-space**

This function deletes all spaces and tabs around point. It returns `nil`.

In the following examples, assume that `delete-horizontal-space` is called four times, once on each line, with point between the second and third characters on the line.

```
---------- Buffer: foo ----------
I -!-thought
I -!-     thought
We-!- thought
Yo-!-u thought
---------- Buffer: foo ----------

(delete-horizontal-space)   ; Four times.
     => nil

---------- Buffer: foo ----------
Ithought
Ithought
Wethought
You thought
---------- Buffer: foo ----------
```

Command: **delete-indentation** *&optional join-following-p*

This function joins the line point is on to the previous line, deleting any whitespace at the join and in some cases replacing it with one space. If join-following-p is non-`nil`, `delete-indentation` joins this line to the following line instead. The value is `nil`.

If there is a fill prefix, and the second of the lines being joined starts with the prefix, then `delete-indentation` deletes the fill prefix before joining the lines.

In the example below, point is located on the line starting `events', and it makes no difference if there are trailing spaces in the preceding line.

```
---------- Buffer: foo ----------
When in the course of human
-!-    events, it becomes necessary
---------- Buffer: foo ----------

(delete-indentation)
     => nil

---------- Buffer: foo ----------
When in the course of human-!- events, it becomes necessary
---------- Buffer: foo ----------
```

After the lines are joined, the function `fixup-whitespace` is responsible for deciding whether to leave a space at the junction.

Function: **fixup-whitespace**

This function replaces white space between the objects on either side of point with either one space or no space as appropriate. It returns `nil`.

The appropriate amount of space is none at the beginning or end of the line. Otherwise, it is one space except when point is before a character with close parenthesis syntax or after a character with open parenthesis or expression-prefix syntax. See section Table of Syntax Classes.

In the example below, when `fixup-whitespace` is called the first time, point is before the word `spaces' in the first line. It is located directly after the `(' for the second invocation.

```
---------- Buffer: foo ----------
This has too many     -!-spaces
This has too many spaces at the start of (-!-   this list)
---------- Buffer: foo ----------

(fixup-whitespace)
     => nil
(fixup-whitespace)
     => nil
```


```
---------- Buffer: foo ----------
This has too many spaces
This has too many spaces at the start of (this list)
---------- Buffer: foo ----------
```

Command: **just-one-space**

This command replaces any spaces and tabs around point with a single space. It returns `nil`.

Command: **delete-blank-lines**

This function deletes blank lines surrounding point. If point is on a blank line with one or more blank lines before or after it, then all but one of them are deleted. If point is on an isolated blank line, then it is deleted. If point is on a nonblank line, the command deletes all blank lines following it.

A blank line is defined as a line containing only tabs and spaces.

`delete-blank-lines` returns `nil`.

The Kill Ring

Kill functions delete text like the deletion functions, but save it so that the user can reinsert it by yanking. Most of these functions have `kill-' in their name. By contrast, the functions whose names start with `delete-' normally do not save text for yanking (though they can still be undone); these are "deletion" functions.

Most of the kill commands are primarily for interactive use, and are not described here. What we do describe are the functions provided for use in writing such commands. When deleting text for internal purposes within a Lisp function, you should normally use deletion functions, so as not to disturb the kill ring contents. See section Deletion of Text.

Emacs saves the last several batches of killed text in a list. We call it the kill ring because, in yanking, the elements are considered to be in a cyclic order. The list is kept in the variable `kill-ring`, and can be operated on with the usual functions for lists; there are also specialized functions, described in this section, which treat it as a ring.

Some people think use of the word "kill" in Emacs is unfortunate, since it refers to processes which specifically *do not* destroy the entities "killed". This is in sharp contrast to ordinary life, in which death is permanent and "killed" entities do not come back to life. Therefore, other metaphors have been proposed. For example, the term "cut ring" makes sense to people who, in pre-computer days, used scissors and paste to cut up and rearrange manuscripts. However, it would be difficult to change now.

Kill Ring Concepts

The kill ring records killed text as strings in a list. A short kill ring, for example, might look like this:

```
("some text" "a different piece of text" "yet more text")
```

New entries in the kill ring go at the front of the list. When the list reaches `kill-ring-max` entries in length, adding a new entry automatically deletes the last entry.

When kill commands are interwoven with other commands, the killed portions of text are put into separate entries in the kill ring. But when two or more kill commands are executed in succession, the text they kill forms a single entry, because the second and subsequent consecutive kill commands append to the entry made by the first one.

The user can reinsert or yank text from any element in the kill ring. One of the entries in the ring is considered the "front", and the simplest yank command yanks that entry. Other yank commands "rotate" the ring by designating other entries as the "front".

Functions for Killing

`kill-region` is the usual subroutine for killing text. Any command that calls this function is a "kill command" (and should probably have `kill' in its name). `kill-region` puts the newly killed text in a new element at the beginning of the kill ring or adds it to the most recent element. It uses the `last-command` variable to keep track of whether the previous was a kill command, and in such cases appends the killed text to the most recent entry.

Command: **kill-region** *start end*

This function kills the text in the region defined by start and end. The text is deleted but saved in the kill ring. The value is always `nil`.

In an interactive call, start and end are point and the mark.

If the buffer is read-only, `kill-region` modifies the kill ring just the same, then signals an error without modifying the buffer. This is convenient because it lets the user use all the kill commands to copy text into the kill ring from a read-only buffer.

Command: **copy-region-as-kill** *start end*

This function saves the region defined by start and end on the kill ring, but does not delete the text from the buffer. It returns `nil`. It also indicates the extent of the text copied by moving the cursor momentarily, or by displaying a message in the echo area.

Don't use this command in Lisp programs; use `kill-new` or `kill-append` instead. See section Low Level Kill Ring.

In an interactive call, start and end are point and the mark.

Functions for Yanking

Command: **yank** *&optional arg*

This function inserts the text in the first entry in the kill ring directly before point. After the yank, the mark is positioned at the beginning and point is positioned after the end of the inserted text.

If arg is a list (which occurs interactively when the user types C-u with no digits), then `yank` inserts text as described above, but puts point before the yanked text and puts the mark after it. If arg is a number, then `yank` inserts the argth most recently killed text.

`yank` does not alter the contents of the kill ring or rotate it. It returns `nil`.

Command: **yank-pop** *arg*

This function replaces the just-yanked text with another batch of killed text--another element of the kill ring.

This command is allowed only immediately after a `yank` or a `yank-pop`. At such a time, the region contains text that was just inserted by the previous `yank`. `yank-pop` deletes that text and inserts in its place a different stretch of killed text. The text that is deleted is not inserted into the kill ring, since it is already in the kill ring somewhere.

If arg is `nil`, then the existing region contents are replaced with the previous element of the kill ring. If arg is numeric, then the argth previous kill is the replacement. If arg is negative, a more recent kill is the replacement.

The sequence of kills in the kill ring wraps around, so that after the oldest one comes the newest one, and before the newest one goes the oldest.

The value is always `nil`.

Low Level Kill Ring

These functions and variables provide access to the kill ring at a lower level, but still convenient for use in Lisp programs. They take care of interaction with X Window selections. They do not exist in Emacs version 18.

Function: **current-kill** *n &optional do-not-move*

The function `current-kill` rotates the yanking pointer in the kill ring by n places, and returns the text at that place in the ring.

If the optional second argument do-not-move is non-`nil`, then `current-kill` doesn't alter the yanking pointer; it just returns the nth kill forward from the current yanking pointer.

If n is zero, indicating a request for the latest kill, `current-kill` calls the value of `interprogram-paste-function` (documented below) before consulting the kill ring.

Function: **kill-new** *string*

This function puts the text string into the kill ring as a new entry at the front of the ring. It also discards the oldest entry if appropriate. It also invokes the value of `interprogram-cut-function` (see below).

Function: **kill-append** *string before-p*

This function appends the text string to the first entry in the kill ring. Normally string goes at the end of

the entry, but if before-p is non-`nil`, it goes at the beginning. This function also invokes the value of `interprogram-cut-function` (see below).

Variable: **interprogram-paste-function**

This variable provides a way of transferring killed text from other programs, when you are using a window system. Its value should be `nil` or a function of no arguments.

If the value is a function, it is called when the "most recent kill" value is called for. If the function returns a non-`nil` values, then that value is used as the "most recent kill". If it returns `nil`, then the first element of the kill ring is used.

Variable: **interprogram-cut-function**

This variable provides a way of communicating killed text to and from other programs, when you are using a window system. Its value should be `nil` or a function of one argument.

If the value is a function, it is called whenever the "most recent kill" is changed, with the new string of killed text as an argument.

Internals of the Kill Ring

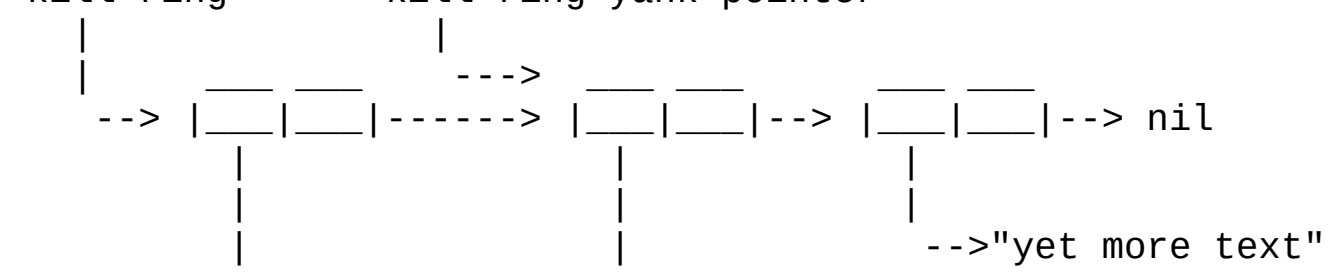
The variable `kill-ring` holds the kill ring contents, in the form of a list of strings. The most recent kill is always at the front of the list.

The `kill-ring-yank-pointer` variable points to a link in the kill ring list, whose CAR is the text that yank functions should copy. Moving `kill-ring-yank-pointer` to a different link is called rotating the kill ring. We call the kill ring a "ring" because the functions that move the yank pointer wrap around from the end of the list to the beginning, or vice-versa. Rotating the ring does not change the value of `kill-ring`.

Both `kill-ring` and `kill-ring-yank-pointer` are Lisp variables whose values are normally lists. The word "pointer" in the name of the `kill-ring-yank-pointer` indicates that the variable's purpose is to identify one element of the list for use by the next yank command.

The value of `kill-ring-yank-pointer` is always `eq` to one of the links in the kill ring list. The element it identifies is the CAR of that link. Commands which change the text in the kill ring also set this variable from `kill-ring`. The effect is to rotate the ring so that the newly killed text is at front.

Here is a diagram that shows the variable `kill-ring-yank-pointer` pointing to the second entry in the kill ring `("some text" "a different piece of text" "yet more text")`.




```
        |                  |
        |                   --> "a different piece of text"
        |
         --> "some text"
```

This circumstance might occur after C-y (`yank`) immediately followed by M-y (`yank-pop`).

Variable: **kill-ring**

List of killed text sequences, most recently killed first.

Variable: **kill-ring-yank-pointer**

This variable's value indicates which element of the kill ring is at the "front" of the ring for yanking. More precisely, the value is a sublist of the value of `kill-ring`, and its CAR is the kill string that C-y should yank.

User Option: **kill-ring-max**

The value of this variable is the maximum length to which the kill ring can grow, before elements are thrown away at the end. The default value for `kill-ring-max` is 30.

Undo

Most buffers have an undo list which records all changes made to the buffer's text so that they can be undone. (The buffers which don't have one are usually special-purpose buffers for which Emacs assumes that undoing is not useful.) All the primitives which modify the text in the buffer automatically add elements to the front of the undo list, which you can find in the variable `buffer-undo-list`.

Variable: **buffer-undo-list**

This variable's value is the undo list of the current buffer. A value of `t` disables the recording of undo information.

Here are the kinds of elements an undo list can have:

`integer`
: This kind of element records a previous value of point. Ordinary cursor motion does not get any sort of undo record, but these entries are used to record where point was before a deletion.

`(beg . end)`
: This kind of element indicates how to delete text that was inserted. Upon insertion, the text occupied the range beg--end in the buffer.

`(pos . deleted)`
: This kind of element indicates how to reinsert text that was deleted. The deleted text itself is the string deleted. The place to reinsert it is pos.

`(t high . low)`
: This kind of element indicates that an unmodified buffer became modified. The elements high and

low are two integers, each recording 16 bits of the visited file's modification time as of when it previously visited or saved. `primitive-undo` uses those values to determine whether to ma the buffer as unmodified once again; it does so only if the file's modification time matches those numbers.

`(nil property value beg . end)`

This kind of element records a change in a text property. Here's how you might undo the change:

```
(put-text-property beg end
                   property value)
```

`nil`

This element is a boundary. The function `undo-boundary` adds these elements. The elements between two boundaries are called a change group; normally, each change group corresponds to one keyboard command, and undo commands normally undo an entire group as a unit.

Function: **undo-boundary**

This function places a boundary element in the undo list. The undo command stops at such a boundary, and successive undo commands undo to earlier and earlier boundaries. This function returns `nil`.

The editor command loop automatically creates an undo boundary between keystroke commands. Thus, each undo normally undoes the effects of one command. Calling this function explicitly is useful for splitting the effects of a command into more than one unit. For example, `query-replace` calls this function after each replacement so that the user can undo individual replacements one by one.

Function: **primitive-undo** *count list*

This is the basic function for undoing elements of an undo list. It undoes the first count elements of list, returning the rest of list. You could write this function in Lisp, but it is convenient to have it in C.

`primitive-undo` adds elements to the buffer's undo list. Undo commands avoid confusion by saving the undo list value at the beginning of a sequence of undo operations. Then the undo operations use and update the saved value. The new elements added by undoing never get into the saved value, so they don't cause any trouble.

Maintaining Undo Lists

This section describes how to enable and disable undo information for a given buffer. It also explains how data from the undo list is discarded automatically so it doesn't get too big.

Recording of undo information in a newly created buffer is normally enabled to start with; but if the buffer name starts with a space, the undo recording is initially disabled. You can explicitly enable or disable undo recording with the following two functions, or by setting `buffer-undo-list` yourself.

Command: **buffer-enable-undo** *&optional buffer-or-name*

This function enables recording undo information for buffer buffer-or-name, so that subsequent changes can be undone. If no argument is supplied, then the current buffer is used. This function does nothing if

undo recording is already enabled in the buffer. It returns `nil`.

In an interactive call, buffer-or-name is the current buffer. You cannot specify any other buffer.

Function: **buffer-disable-undo** *buffer*

Function: **buffer-flush-undo** *buffer*

This function discards the undo list of buffer, and disables further recording of undo information. As a result, it is no longer possible to undo either previous changes or any subsequent changes. If the undo list of buffer is already disabled, this function has no effect.

This function returns `nil`. It cannot be called interactively.

The name `buffer-flush-undo` is not considered obsolete, but the preferred name `buffer-disable-undo` was not provided in Emacs versions 18 and earlier.

As editing continues, undo lists get longer and longer. To prevent them from using up all available memory space, garbage collection trims them back to size limits you can set. (For this purpose, the "size" of an undo list measures the cons cells that make up the list, plus the strings of deleted text.) Two variables control the range of acceptable sizes: `undo-limit` and `undo-strong-limit`.

Variable: **undo-limit**

This is the soft limit for the acceptable size of an undo list. The change group at which this size is exceeded is the last one kept.

Variable: **undo-strong-limit**

The upper limit for the acceptable size of an undo list. The change group at which this size is exceeded is discarded itself (along with all subsequent changes). There is one exception: garbage collection always keeps the very last change group no matter how big it is.

Filling

Filling means adjusting the lengths of lines (by moving words between them) so that they are nearly (but no greater than) a specified maximum width. Additionally, lines can be justified, which means that spaces are inserted between words to make the line exactly the specified width. The width is controlled by the variable `fill-column`. For ease of reading, lines should be no longer than 70 or so columns.

You can use Auto Fill mode (see section Auto Filling) to fill text automatically as you insert it, but changes to existing text may leave it improperly filled. Then you must fill the text explicitly.

Most of the functions in this section return values that are not meaningful.

Command: **fill-paragraph** *justify-flag*

This function fills the paragraph at or after point. If justify-flag is non-`nil`, each line is justified as well. It uses the ordinary paragraph motion commands to find paragraph boundaries.

Command: **fill-region** *start end &optional justify-flag*

This function fills each of the paragraphs in the region from start to end. It justifies as well if justify-flag is non-`nil`. (In an interactive call, this is true if there is a prefix argument.)

The variable `paragraph-separate` controls how to distinguish paragraphs.

Command: **fill-individual-paragraphs** *start end &optional justify-flag mail-flag*

This function fills each paragraph in the region according to its individual fill prefix. Thus, if the lines of a paragraph are indented with spaces, the filled paragraph will continue to be indented in the same fashion.

The first two arguments, start and end, are the beginning and end of the region that will be filled. The third and fourth arguments, justify-flag and mail-flag, are optional. If justify-flag is non-`nil`, the paragraphs are justified as well as filled. If mail-flag is non-`nil`, the function is told that it is operating on a mail message and therefore should not fill the header lines.

Ordinarily, `fill-individual-paragraphs` regards each change in indentation as starting a new paragraph. If `fill-individual-varying-indent` is non-`nil`, then only separator lines separate paragraphs. That mode can handle paragraphs with extra indentation on the first line.

User Option: **fill-individual-varying-indent**

This variable alters the action of `fill-individual-paragraphs` as described above.

Command: **fill-region-as-paragraph** *start end &optional justify-flag*

This function considers a region of text as a paragraph and fills it. If the region was made up of many paragraphs, the blank lines between paragraphs are removed. This function justifies as well as filling when justify-flag is non-`nil`. In an interactive call, any prefix argument requests justification.

In Adaptive Fill mode, which is enabled by default, `fill-region-as-paragraph` on an indented paragraph when there is no fill prefix uses the indentation of the second line of the paragraph as the fill prefix.

Command: **justify-current-line**

This function inserts spaces between the words of the current line so that the line ends exactly at `fill-column`. It returns `nil`.

User Option: **fill-column**

This buffer-local variable specifies the maximum width of filled lines. Its value should be an integer, which is a number of columns. All the filling, justification and centering commands are affected by this variable, including Auto Fill mode (see section Auto Filling).

As a practical matter, if you are writing text for other people to read, you should set `fill-column` to no more than 70. Otherwise the line will be too long for people to read comfortably, and this can make the text seem clumsy.

Variable: **default-fill-column**

The value of this variable is the default value for `fill-column` in buffers that do not override it. This is the same as `(default-value 'fill-column)`.

The default value for `default-fill-column` is 70.

Auto Filling

Filling breaks text into lines that are no more than a specified number of columns wide. Filled lines end between words, and therefore may have to be shorter than the maximum width.

Auto Fill mode is a minor mode in which Emacs fills lines automatically as text as inserted. This section describes the hook and the two variables used by Auto Fill mode. For a description of functions that you can call manually to fill and justify text, see section Filling.

Variable: **auto-fill-function**

The value of this variable should be a function (of no arguments) to be called after self-inserting a space at a column beyond `fill-column`. It may be `nil`, in which case nothing special is done.

The default value for `auto-fill-function` is `do-auto-fill`, a function whose sole purpose is to implement the usual strategy for breaking a line.

> In older Emacs versions, this variable was named `auto-fill-hook`, but since it is not called with the standard convention for hooks, it was renamed to `auto-fill-function` in version 19.

Sorting Text

The sorting commands described in this section all rearrange text in a buffer. This is in contrast to the function `sort`, which rearranges the order of the elements of a list (see section Functions that Rearrange Lists). The values returned by these commands are not meaningful.

Command: **sort-regexp-fields** *reverse record-regexp key-regexp start end*

This command sorts the region between start and end alphabetically as specified by record-regexp and key-regexp. If reverse is a negative integer, then sorting is in reverse order.

Alphabetical sorting means that two sort keys are compared by comparing the first characters of each, the second characters of each, and so on. If a mismatch is found, it means that the sort keys are unequal; the sort key whose character is less at the point of first mismatch is the lesser sort key. The individual characters are compared according to their numerical values. Since Emacs uses the ASCII character set, the ordering in that set determines alphabetical order.

The value of the record-regexp argument specifies the textual units or records that should be sorted. At the end of each record, a search is done for this regular expression, and the text that matches it is the next record. For example, the regular expression `\^.+$', which matches lines with at least one character besides a newline, would make each such line into a sort record. See section Regular Expressions, for a description of the syntax and meaning of regular expressions.

The value of the key-regexp argument specifies what part of each record is to be compared against the other records. The key-regexp could match the whole record, or only a part. In the latter case, the rest of the record has no effect on the sorted order of records, but it is carried along when the record moves to its new position.

The key-regexp argument can refer to the text matched by a subexpression of record-regexp, or it can be a regular expression on its own.

If key-regexp is:

`\digit'

then the text matched by the digitth `\(...\)' parenthesis grouping in record-regexp is used for sorting.

`\&'

then the whole record is used for sorting.

a regular expression

then the function searches for a match for the regular expression within the record. If such a match is found, it is used for sorting. If a match for key-regexp is not found within a record then that record is ignored, which means its position in the buffer is not changed. (The other records may move around it.)

For example, if you plan to sort all the lines in the region by the first word on each line starting with the letter `f', you should set record-regexp to `^.*$' and set key-regexp to `\<f\w*\>'. The resulting expression looks like this:

```
(sort-regexp-fields nil "^.*$" "\\<f\\w*\\>"
                    (region-beginning)
                    (region-end))
```

If you call `sort-regexp-fields` interactively, you are prompted for record-regexp and key-regexp in the minibuffer.

Command: **sort-subr** *reverse nextrecfun endrecfun &optional startkeyfun endkeyfun*

This command is the general text sorting routine that divides a buffer into records and sorts them. The functions `sort-lines`, `sort-paragraphs`, `sort-pages`, `sort-fields`, `sort-regexp-fields` and `sort-numeric-fields` all use `sort-subr`.

To understand how `sort-subr` works, consider the whole accessible portion of the buffer as being divided into disjoint pieces called sort records. A portion of each sort record (perhaps all of it) is designated as the sort key. The records are rearranged in the buffer in order by their sort keys. The records may or may not be contiguous.

Usually, the records are rearranged in order of ascending sort key. If the first argument to the `sort-subr` function, reverse, is non-`nil`, the sort records are rearranged in order of descending sort key.

The next four arguments to `sort-subr` are functions that are called to move point across a sort re
They are called many times from within `sort-subr`.

1. nextrecfun is called with point at the end of a record. This function moves point to the start of the next record. The first record is assumed to start at the position of point when `sort-subr` is called. (Therefore, you should usually move point to the beginning of the buffer before calling `sort-subr`.)

   This function can indicate there are no more sort records by leaving point at the end of the buffer.
2. endrecfun is called with point within a record. It moves point to the end of the record.
3. startkeyfun is called to move point from the start of a record to the start of the sort key. This argument is optional. If supplied, the function should either return a non-`nil` value to be used as the sort key, or return `nil` to indicate that the sort key is in the buffer starting at point. In the latter case, endkeyfun is called to find the end of the sort key.
4. endkeyfun is called to move point from the start of the sort key to the end of the sort key. This argument is optional. If startkeyfun returns `nil` and this argument is omitted (or `nil`), then the sort key extends to the end of the record. There is no need for endkeyfun if startkeyfun returns a non-`nil` value.

As an example of `sort-subr`, here is the complete function definition for `sort-lines`:

```
;; Note that the first two lines of doc string
;; are effectively one line when viewed by a user.
(defun sort-lines (reverse beg end)
  "Sort lines in region alphabetically;\
 argument means descending order.
Called from a program, there are three arguments:
REVERSE (non-nil means reverse order),
and BEG and END (the region to sort)."
  (interactive "P\nr")
  (save-restriction
    (narrow-to-region beg end)
    (goto-char (point-min))
    (sort-subr reverse
               'forward-line
               'end-of-line)))
```

Here `forward-line` moves point to the start of the next record, and `end-of-line` moves point to the end of record. We do not pass the arguments startkeyfun and endkeyfun, because the entire record is used as the sort key.

The `sort-paragraphs` function is very much the same, except that its `sort-subr` call looks like this:

```
(sort-subr reverse
           (function
```


```
     (lambda ()
       (skip-chars-forward "\n \t\f")))
   'forward-paragraph)
```

Command: **sort-lines** *reverse start end*

This command sorts lines in the region between start and end alphabetically. If reverse is non-`nil`, the sort is in reverse order.

Command: **sort-paragraphs** *reverse start end*

This command sorts paragraphs in the region between start and end alphabetically. If reverse is non-`nil`, the sort is in reverse order.

Command: **sort-pages** *reverse start end*

This command sorts pages in the region between start and end alphabetically. If reverse is non-`nil`, the sort is in reverse order.

Command: **sort-fields** *field start end*

This command sorts lines in the region between start and end, comparing them alphabetically by the fieldth field of each line. Fields are separated by whitespace and numbered starting from 1. If field is negative, sorting is by the -fieldth field from the end of the line. This command is useful for sorting tables.

Command: **sort-numeric-fields** *field start end*

This command sorts lines in the region between start and end, comparing them numerically by the fieldth field of each line. Fields are separated by whitespace and numbered starting from 1. The specified field must contain a number in each line of the region. If field is negative, sorting is by the -fieldth field from the end of the line. This command is useful for sorting tables.

Command: **sort-columns** *reverse &optional beg end*

This command sorts the lines in the region between beg and end, comparing them alphabetically by a certain range of columns. The column positions of beg and end bound the range of columns to sort on.

If reverse is non-`nil`, the sort is in reverse order.

One unusual thing about this command is that the entire line containing position beg, and the entire line containing position end, are included in the region sorted.

Note that `sort-columns` uses the `sort` utility program, and so cannot work properly on text containing tab characters. Use M-x `untabify` to convert tabs to spaces before sorting.

The `sort-columns` function did not work on VMS prior to Emacs 19.

Indentation

The indentation functions are used to examine, move to, and change whitespace that is at the beginning of a line. Some of the functions can also change whitespace elsewhere on a line. Indentation always counts from zero at the left margin.

Indentation Primitives

This section describes the primitive functions used to count and insert indentation. The functions in the following sections use these primitives.

Function: **current-indentation**

This function returns the indentation of the current line, which is the horizontal position of the first nonblank character. If the contents are entirely blank, then this is the horizontal position of the end of the line.

Command: **indent-to** *column &optional minimum*

This function indents from point with tabs and spaces until column is reached. If minimum is specified and non-`nil`, then at least that many spaces are inserted even if this requires going beyond column. The value is the column at which the inserted indentation ends.

User Option: **indent-tabs-mode**

If this variable is non-`nil`, indentation functions can insert tabs as well as spaces. Otherwise, they insert only spaces. Setting this variable automatically makes it local to the current buffer.

Indentation Controlled by Major Mode

An important function of each major mode is to customize the TAB key to indent properly for the language being edited. This section describes the mechanism of the TAB key and how to control it. The functions in this section return unpredictable values.

Variable: **indent-line-function**

This variable's value is the function to be used by TAB (and various commands) to indent the current line. The command `indent-according-to-mode` does no more than call this function.

In Lisp mode, the value is the symbol `lisp-indent-line`; in C mode, `c-indent-line`; in Fortran mode, `fortran-indent-line`. In Fundamental mode, Text mode, and many other modes with no standard for indentation, the value is `indent-to-left-margin` (which is the default value).

Command: **indent-according-to-mode**

This command calls the function in `indent-line-function` to indent the current line in a way appropriate for the current major mode.

Command: **indent-for-tab-command**

This command calls the function in `indent-line-function` to indent the current line, except that that function is `indent-to-left-margin`, `insert-tab` is called instead. (That is a trivial command which inserts a tab character.)

Variable: **left-margin**

This variable is the column to which the default `indent-line-function` will indent. (That function is `indent-to-left-margin`.) In Fundamental mode, LFD indents to this column. This variable automatically becomes buffer-local when set in any fashion.

Function: **indent-to-left-margin**

This is the default `indent-line-function`, used in Fundamental mode, Text mode, etc. Its effect is to adjust the indentation at the beginning of the current line to the value specified by the variable `left-margin`. This may involve either inserting or deleting whitespace.

Command: **newline-and-indent**

This function inserts a newline, then indents the new line (the one following the newline just inserted) according to the major mode.

Indentation is done using the current `indent-line-function`. In programming language modes, this is the same thing TAB does, but in some text modes, where TAB inserts a tab, `newline-and-indent` indents to the column specified by `left-margin`.

Command: **reindent-then-newline-and-indent**

This command reindents the current line, inserts a newline at point, and then reindents the new line (the one following the newline just inserted).

Indentation of both lines is done according to the current major mode; this means that the current value of `indent-line-function` is called. In programming language modes, this is the same thing TAB does, but in some text modes, where TAB inserts a tab, `reindent-then-newline-and-indent` indents to the column specified by `left-margin`.

Indenting an Entire Region

This section describes commands which indent all the lines in the region. They return unpredictable values.

Command: **indent-region** *start end to-column*

This command indents each nonblank line starting between start (inclusive) and end (exclusive). If to-column is `nil`, `indent-region` indents each nonblank line by calling the current mode's indentation function, the value of `indent-line-function`.

If to-column is non-`nil`, it should be an integer specifying the number of columns of indentation; then this function gives each line exactly that much indentation, by either adding or deleting whitespace.

If there is a fill prefix, `indent-region` indents each line by making it start with the fill prefix.

Variable: **indent-region-function**

The value of this variable is a function that can be used by `indent-region` as a short cut. You should design the function so that it will produce the same results as indenting the lines of the region one by one (but presumably faster).

If the value is `nil`, there is no short cut, and `indent-region` actually works line by line.

A short cut function is useful in modes such as C mode and Lisp mode, where the `indent-line-function` must scan from the beginning of the function: applying it to each line would be quadratic in time. The short cut can update the scan information as it moves through the lines indenting them; this takes linear time. If indenting a line individually is fast, there is no need for a short cut.

`indent-region` with a non-`nil` argument has a different definition and does not use this variable.

Command: **indent-rigidly** *start end count*

This command indents all lines starting between start (inclusive) and end (exclusive) sideways by `count` columns. This "preserves the shape" of the affected region, moving it as a rigid unit. Consequently, this command is useful not only for indenting regions of unindented text, but also for indenting regions of formatted code.

For example, if count is 3, this command adds 3 columns of indentation to each of the lines beginning in the region specified.

In Mail mode, C-c C-y (`mail-yank-original`) uses `indent-rigidly` to indent the text copied from the message being replied to.

Function: **indent-code-rigidly** *start end columns &optional nochange-regexp*

This is like `indent-rigidly`, except that it doesn't alter lines that start within strings or comments.

In addition, it doesn't alter a line if nochange-regexp matches at the beginning of the line (if nochange-regexp is non-`nil`).

Indentation Relative to Previous Lines

This section describes two commands which indent the current line based on the contents of previous lines.

Command: **indent-relative** *&optional unindented-ok*

This function inserts whitespace at point, extending to the same column as the next indent point of the previous nonblank line. An indent point is a non-whitespace character following whitespace. The next indent point is the first one at a column greater than the current column of point. For example, if point is underneath and to the left of the first non-blank character of a line of text, it moves to that column by inserting whitespace.

If the previous nonblank line has no next indent point (i.e., none at a great enough column position), this function either does nothing (if unindented-ok is non-`nil`) or calls `tab-to-tab-stop`. Thus, if point

is underneath and to the right of the last column of a short line of text, this function moves point to th next tab stop by inserting whitespace.

This command returns an unpredictable value.

In the following example, point is at the beginning of the second line:

```
            This line is indented twelve spaces.
-!-The quick brown fox jumped.
```

Evaluation of the expression `(indent-relative nil)` produces the following:

```
            This line is indented twelve spaces.
            -!-The quick brown fox jumped.
```

In this example, point is between the `m' and `p' of `jumped':

```
            This line is indented twelve spaces.
The quick brown fox jum-!-ped.
```

Evaluation of the expression `(indent-relative nil)` produces the following:

```
            This line is indented twelve spaces.
The quick brown fox jum  -!-ped.
```

Command: **indent-relative-maybe**

This command indents the current line like the previous nonblank line. The function consists of a call to `indent-relative` with a non-`nil` value passed to the unindented-ok optional argument. The value is unpredictable.

If the previous line has no indentation, the current line is given no indentation (any existing indentation is deleted); if the previous nonblank line has no indent points beyond the column at which point starts, nothing is changed.

Adjustable "Tab Stops"

This section explains the mechanism for user-specified "tab stops" and the mechanisms which use and set them. The name "tab stops" is used because the feature is similar to that of the tab stops on a typewriter. The feature works by inserting an appropriate number of spaces and tab characters to reach the designated position, like the other indentation functions; it does not affect the display of tab characters in the buffer (see section Usual Display Conventions). Note that the TAB character as input uses this tab stop feature only in a few major modes, such as Text mode.

Function: **tab-to-tab-stop**

This function inserts spaces or tabs up to the next tab stop column defined by `tab-stop-list`. It searches the list for an element greater than the current column number, and uses that element as the

column to indent to. If no such element is found, then nothing is done.

User Option: **tab-stop-list**

This variable is the list of tab stop columns used by `tab-to-tab-stops`. The elements should be integers in increasing order. The tab stop columns need not be evenly spaced.

Use M-x edit-tab-stops to edit the location of tab stops interactively.

Indentation-Based Motion Commands

These commands, primarily for interactive use, act based on the indentation in the text.

Command: **back-to-indentation**

This command moves point to the first non-whitespace character in the current line (which is the line in which point is located). It returns `nil`.

Command: **backward-to-indentation** *arg*

This command moves point backward arg lines and then to the first nonblank character on that line. It returns `nil`.

Command: **forward-to-indentation** *arg*

This command moves point forward arg lines and then to the first nonblank character on that line. It returns `nil`.

Counting Columns

The column functions convert between a character position (counting characters from the beginning of the buffer) and a column position (counting screen characters from the beginning of a line).

Column number computations ignore the width of the window and the amount of horizontal scrolling. Consequently, a column value can be arbitrarily high. The first (or leftmost) column is numbered 0.

A character counts according to the number of columns it occupies on the screen. This means control characters count as occupying 2 or 4 columns, depending upon the value of `ctl-arrow`, and tabs count as occupying a number of columns that depends on the value of `tab-width` and on the column where the tab begins. See section Usual Display Conventions.

Function: **current-column**

This function returns the horizontal position of point, measured in columns, counting from 0 at the left margin. The column count is calculated by adding together the widths of all the displayed representations of the characters between the start of the current line and point.

For a more complicated example of the use of `current-column`, see the description of `count-lines` in section Motion by Text Lines.

Function: **move-to-column** *column &optional force*

This function moves point to column in the current line. The calculation of column takes into account the widths of all the displayed representations of the characters between the start of the line and point.

If the argument column is greater than the column position of the end of the line, point moves to the end of the line. If column is negative, point moves to the beginning of the line.

If it is impossible to move to column column because that is in the middle of a multicolumn character such as a tab, point moves to the end of that character. However, if force is non-`nil`, and column is in the middle of a tab, then `move-to-column` converts the tab into spaces so that it can move precisely to column column.

The argument force also has an effect if the line isn't long enough to reach column column; in that case, it says to indent at the end of the line to reach that column.

If column is not an integer, an error is signaled.

The return value is the column number actually moved to.

Case Changes

The case change commands described here work on text in the current buffer. See section Character Case, for case conversion commands that work on strings and characters. See section The Case Table, for how to customize which characters are upper or lower case and how to convert them.

Command: **capitalize-region** *start end*

This function capitalizes all words in the region defined by start and end. To capitalize means to convert each word's first character to upper case and convert the rest of each word to lower case. The function returns `nil`.

If one end of the region is in the middle of a word, the part of the word within the region is treated as an entire word.

When `capitalize-region` is called interactively, start and end are point and the mark, with the smallest first.

```
---------- Buffer: foo ----------
This is the contents of the 5th foo.
---------- Buffer: foo ----------

(capitalize-region 1 44)
=> nil

---------- Buffer: foo ----------
This Is The Contents Of The 5th Foo.
---------- Buffer: foo ----------
```

Command: **downcase-region** *start end*

This function converts all of the letters in the region defined by start and end to lower case. The function returns `nil`.

When `downcase-region` is called interactively, start and end are point and the mark, with the smallest first.

Command: **upcase-region** *start end*

This function converts all of the letters in the region defined by start and end to upper case. The function returns `nil`.

When `upcase-region` is called interactively, start and end are point and the mark, with the smallest first.

Command: **capitalize-word** *count*

This function capitalizes count words after point, moving point over as it does. To capitalize means to convert each word's first character to upper case and convert the rest of each word to lower case. If count is negative, the function capitalizes the -count previous words but does not move point. The value is `nil`.

If point is in the middle of a word, the part of word the before point (if moving forward) or after point (if operating backward) is ignored. The rest is treated as an entire word.

When `capitalize-word` is called interactively, count is set to the numeric prefix argument.

Command: **downcase-word** *count*

This function converts the count words after point to all lower case, moving point over as it does. If count is negative, it converts the -count previous words but does not move point. The value is `nil`.

When `downcase-word` is called interactively, count is set to the numeric prefix argument.

Command: **upcase-word** *count*

This function converts the count words after point to all upper case, moving point over as it does. If count is negative, it converts the -count previous words but does not move point. The value is `nil`.

When `upcase-word` is called interactively, count is set to the numeric prefix argument.

Text Properties

Each character position in a buffer or a string can have a text property list, much like the property list of a symbol. The properties belong to a particular character at a particular place, such as, the letter `T' at the beginning of this sentence or the first `o' in `foo'---if the same character occurs in two different places, the two occurrences generally have different properties.

Each property has a name, which is usually a symbol, and an associated value, which can be any Lisp

object--just as for properties of symbols (see section Property Lists).

If a character has a `category` property, we call it the category of the character. It should be a symbol. The properties of the symbol serve as defaults for the properties of the character.

Copying text between strings and buffers preserves the properties along with the characters; this includes such diverse functions as `substring`, `insert`, and `buffer-substring`.

Examining Text Properties

The simplest way to examine text properties is to ask for the value of a particular property of a particular character. For that, use `get-text-property`. Use `text-properties-at` to get the entire property list of a character. See section Property Search Functions, for functions to examine the properties of a number of characters at once.

These functions handle both strings and buffers. Keep in mind that positions in a string start from 0, whereas positions in a buffer start from 1.

Function: **get-text-property** *pos prop &optional object*

This function returns the value of the prop property of the character after position pos in object (a buffer or string). The argument object is optional and defaults to the current buffer.

If there is no prop property strictly speaking, but the character has a category which is a symbol, then `get-text-property` returns the prop property of that symbol.

Function: **text-properties-at** *position &optional object*

This function returns the list of properties held by the character at position in the string or buffer object. If object is `nil`, it defaults to the current buffer.

Changing Text Properties

The primitives for changing properties apply to a specified range of text. The function `set-text-properties` (see end of section) sets the entire property list of the text in that range; more often, it is useful to add, change, or delete just certain properties specified by name.

Since text properties are considered part of the buffer's contents, and can affect how the buffer looks on the screen, any change in the text properties is considered a buffer modification. Buffer text property changes are undoable.

Function: **add-text-properties** *start end props &optional object*

This function modifies the text properties for the text between start and end in the string or buffer object. If object is `nil`, it defaults to the current buffer.

The argument props specifies which properties to change. It should have the form of a property list (see section Property Lists): a list whose elements include the property names followed alternately by the corresponding values.

The return value is `t` if the function actually changed some property's value; `nil` otherwise (if props is `nil` or its values agree with those in the text).

For example, here is how to set the `comment` property to `t` for a range of text:

```
(add-text-properties (region-beginning)
                     (region-end)
                     (list 'comment t))
```

Function: **put-text-property** *start end prop value &optional object*

This function sets the prop property to value for the text between start and end in the string or buffer object. If object is `nil`, it defaults to the current buffer.

Function: **remove-text-properties** *start end props &optional object*

This function deletes specified text properties from the text between start and end in the string or buffer object. If object is `nil`, it defaults to the current buffer.

The argument props specifies which properties to delete. It should have the form of a property list (see section Property Lists): a list whose elements include the property names followed by the corresponding values. The property names mentioned in props are the ones deleted from the text. The values associated in props with these names do not matter.

The return value is `t` if the function actually changed some property's value; `nil` otherwise (if props is `nil` or if none of the text had any of those properties).

Function: **set-text-properties** *start end props &optional object*

This function completely replaces the text property list for the text between start and end in the string or buffer object. If object is `nil`, it defaults to the current buffer.

The argument props is the new property list. It should have the form of a list whose elements include the property names followed by the corresponding values.

After `set-text-properties` returns, all the characters in the specified range have identical properties.

If props is `nil`, the effect is to get rid of all properties from the specified range of text. Here's an example:

```
(set-text-properties (region-beginning)
                     (region-end)
                     nil)
```

Property Search Functions

In typical use of text properties, most of the time several or many consecutive characters have the same value for a property. Rather than writing your programs to examine characters one by one, it is much faster to process chunks of text that have the same property value.

Here are functions you can use to do this. In all cases, object defaults to the current buffer.

Function: **next-property-change** *pos &optional object*

The function scans the text forward from position pos in the string or buffer object till it finds a change in some text property, then returns the position of the change. In other words, it returns the position of the first character beyond pos whose properties are not identical to those of the character just after pos.

The value is `nil` if the properties remain unchanged all the way to the end of object. If the value is non-`nil`, it is a position greater than pos, never equal.

Here is an example of how to scan the buffer by chunks of text within which all properties are constant:

```
(while (not (eobp))
  (let ((plist (text-properties-at (point)))
        (next-change
         (or (next-property-change (point) (current-buffer))
             (point-max))))
    Process text from point to next-change...
    (goto-char next-change)))
```

Function: **next-single-property-change** *pos prop &optional object*

The function scans the text forward from position pos in the string or buffer object till it finds a change in the prop property, then returns the position of the change. In other words, it returns the position of the first character beyond pos whose prop property differs from that of the character just after pos.

The value is `nil` if the properties remain unchanged all the way to the end of object. If the value is non-`nil`, it is a position greater than pos, never equal.

Function: **previous-property-change** *pos &optional object*

This is like `next-property-change`, but scans back from pos instead of forward. If the value is non-`nil`, it is a position always strictly less than pos.

Function: **previous-single-property-change** *pos prop &optional object*

This is like `next-property-change`, but scans back from pos instead of forward. If the value is non-`nil`, it is a position always strictly less than pos.

Special Properties

If a character has a `category` property, we call it the category of the character. It should be a symbol. The properties of the symbol serve as defaults for the properties of the character.

You can use the property `face` to control the font and color of text. See section Faces, for more information. This feature is temporary; in the future, we may replace it with other ways of specifying how to display text.

The property `mouse-face` is used instead of `face` when the mouse is on or near the character. For this purpose, "near" means that all text between the character and where the mouse is have the same `mouse-face` property value.

You can specify a different keymap for a portion of the text by means of a `local-map` property. The property's value, for the character after point, replaces the buffer's local map. See section Active Keymaps.

If a character has the property `read-only`, then modifying that character is not allowed. Any command that would do so gets an error.

If a character has the property `modification-hooks`, then its value should be a list of functions; modifying that character calls all of those functions. Each function receives two arguments: the beginning and end of the part of the buffer being modified. Note that if a particular modification hook function appears on several characters being modified by a single primitive, you can't predict how many times the function will be called.

Insertion of text does not, strictly speaking, change any existing character, so there is a special rule for insertion. It compares the `read-only` properties of the two surrounding characters; if they are non-`nil` and `eq` to each other, then the insertion is not allowed. Assuming insertion is allowed, it then gets the `modification-hooks` properties of those characters and calls all the functions in each of them. (If a function appears on both characters, it may be called once or twice.)

See also section Change Hooks, for other hooks that are called when you change text in a buffer.

The special properties `point-entered` and `point-left` record hook functions that report motion of point. Each time point moves, Emacs compares these two property values:

- the `point-left` property of the character after the old location, and
- the `point-entered` property of the character after the new location.

If these two values differ, each of them is called (if not `nil`) with two arguments: the old value of point, and the new one.

The same comparison is made for the characters before the old and new locations. The result may be to execute two `point-left` functions (which may be the same function) and/or two `point-entered` functions (which may be the same function). The `point-left` functions are always called before the `point-entered` functions.

A primitive function may examine characters at various positions without moving point to those

positions. Only an actual change in the value of point runs these hook functions.

Why Text Properties are not Intervals

Some editors that support adding attributes to text in the buffer do so by letting the user specify "intervals" within the text, and adding the properties to the intervals. Those editors permit the user or the programmer to determine where individual intervals start and end. We deliberately provided a different sort of interface in Emacs Lisp to avoid certain paradoxical behavior associated with text modification.

If the actual subdivision into intervals is meaningful, that means you can distinguish between a buffer that is just one interval with a certain property, and a buffer containing the same text subdivided into two intervals, both of which have that property.

Suppose you take the buffer with just one interval and kill part of the text. The text remaining in the buffer is one interval, and the copy in the kill ring (and the undo list) becomes a separate interval. Then if you undo the kill, you get two intervals with the same properties. Thus, the distinction can't be preserved when editing happens.

But suppose we "fix" this problem by coalescing the two intervals when the text is inserted. That works fine if the buffer originally was a single interval. But if it was two intervals, and the killed text equals one of them, then undoing the kill yields just one interval. Again, the distinction can't be preserved.

Insertion of text at the border between intervals also raises questions that have no satisfactory answer.

However, it is easy to arrange for editing to behave consistently for questions of the form, "What are the properties of this character?" So we have decided these are the only questions that make sense; we have not implemented asking questions about where intervals start or end.

For practical purposes, the property search functions serve in place of explicit interval boundaries. You can think of them as finding the boundaries of intervals, assuming that intervals are always coalesced whenever possible. See section Property Search Functions.

Emacs also provides explicit intervals as a presentation feature; see section Overlays.

Substituting for a Character Code

The following functions replace characters within a specified region based on their character codes.

Function: **subst-char-in-region** *start end old-char new-char &optional noundo*

This function replaces all occurrences of the character old-char with the character new-char in the region of the current buffer defined by start and end.

If noundo is non-`nil`, then `subst-char-in-region` does not record the change for undo and does not mark the buffer as modified. This feature is useful for changes which are not considered significant, such as when Outline mode changes visible lines to invisible lines and vice versa.

`subst-char-in-region` does not move point and returns `nil`.


```
---------- Buffer: foo ----------
This is the contents of the buffer before.
---------- Buffer: foo ----------

(subst-char-in-region 1 20 ?i ?X)
     => nil

---------- Buffer: foo ----------
ThXs Xs the contents of the buffer before.
---------- Buffer: foo ----------
```

Function: **translate-region** *start end table*

This function applies a translation table to the characters in the buffer between positions start and end.

The translation table table is a string; `(aref table ochar)` gives the translated character corresponding to ochar. If the length of table is less than 256, any characters with codes larger than the length of table are not altered by the translation.

The return value of `translate-region` is the number of characters which were actually changed by the translation. This does not count characters which were mapped into themselves in the translation table.

This function is available in Emacs versions 19 and later.

Underlining

The underlining commands are somewhat obsolete. The `underline-region` function actually inserts `_^H' before each appropriate character in the region. This command provides a minimal text formatting feature that might work on your printer; however, we recommend instead that you use more powerful text formatting facilities, such as Texinfo.

Command: **underline-region** *start end*

This function underlines all nonblank characters in the region defined by start and end. That is, an underscore character and a backspace character are inserted just before each non-whitespace character in the region. The backspace characters are intended to cause overstriking, but in Emacs they display as either `\010' or `^H', depending on the setting of `ctl-arrow`. There is no way to see the effect of the overstriking within Emacs. The value is `nil`.

Command: **ununderline-region** *start end*

This function removes all underlining (overstruck underscores) in the region defined by start and end. The value is `nil`.

Registers

A register is a sort of variable used in Emacs editing that can hold a marker, a string, a rectangle, a window configuration (of one frame), or a frame configuration (of all frames). Each register is named by a single character. All characters, including control and meta characters (but with the exception of C-g), can be used to name registers. Thus, there are 255 possible registers. A register is designated in Emacs Lisp by a character which is its name.

The functions in this section return unpredictable values unless otherwise stated.

Variable: **register-alist**

This variable is an alist of elements of the form `(name . contents)`. Normally, there is one element for each Emacs register that has been used.

The object name is a character (an integer) identifying the register. The object contents is a string, marker, or list representing the register contents. A string represents text stored in the register. A marker represents a position. A list represents a rectangle; its elements are strings, one per line of the rectangle.

Command: **view-register** *reg*

This command displays what is contained in register reg.

Function: **get-register** *reg*

This function returns the contents of the register reg, or `nil` if it has no contents.

Function: **set-register** *reg value*

This function sets the contents of register reg to value. A register can be set to any value, but the other register functions expect only certain data types. The return value is value.

Command: **point-to-register** *reg*

This command stores both the current location of point and the current buffer in register reg as a marker.

Command: **jump-to-register** *reg*

Command: **register-to-point** *reg*

This command restores the status recorded in register reg.

If reg contains a marker, it moves point to the position stored in the marker. Since both the buffer and the location within the buffer are stored by the `point-to-register` function, this command can switch you to another buffer.

If reg contains a window configuration or a frame configuration. `jump-to-register` restores that configuration.

Command: **insert-register** *reg &optional beforep*

This command inserts contents of register reg into the current buffer.

Normally, this command puts point before the inserted text, and the mark after it. However, if the optional second argument beforep is non-`nil`, it puts the mark before and point after. You can pass a non-`nil` second argument beforep to this function interactively by supplying any prefix argument.

If the register contains a rectangle, then the rectangle is inserted with its upper left corner at point. This means that text is inserted in the current line and underneath it on successive lines.

If the register contains something other than saved text (a string) or a rectangle (a list), currently useless things happen. This may be changed in the future.

Command: **copy-to-register** *reg start end &optional delete-flag*

This command copies the region from start to end into register reg. If delete-flag is non-`nil`, it deletes the region from the buffer after copying it into the register.

Command: **prepend-to-register** *reg start end &optional delete-flag*

This command prepends the region from start to end into register reg. If delete-flag is non-`nil`, it deletes the region from the buffer after copying it to the register.

Command: **append-to-register** *reg start end &optional delete-flag*

This command appends the region from start to end to the text already in register reg. If delete-flag is non-`nil`, it deletes the region from the buffer after copying it to the register.

Command: **copy-rectangle-to-register** *reg start end &optional delete-flag*

This command copies a rectangular region from start to end into register reg. If delete-flag is non-`nil`, it deletes the region from the buffer after copying it to the register.

Command: **window-configuration-to-register** *reg*

This function stores the window configuration of the selected frame in register reg.

Command: **frame-configuration-to-register** *reg*

This function stores the current frame configuration in register reg.

Change Hooks

These hook variables let you arrange to take notice of all changes in all buffers (or in a particular buffer, if you make them buffer-local). See also section Special Properties, for how to detect changes to specific parts of the text.

Variable: **before-change-function**

If this variable is non-`nil`, then it should be a function; the function is called before any buffer modification. Its arguments are the beginning and end of the region that is going to change, represented as integers. The buffer that's about to change is always the current buffer.

Variable: **after-change-function**

If this variable is non-`nil`, then it should be a function; the function is called after any buffer modification. It receives three arguments: the beginning and end of the region just changed, and the length of the text that existed before the change. (To get the current length, subtract the region beginning from the region end.) All three arguments are integers. The buffer that's about to change is always the current buffer.

Both of these variables are temporarily bound to `nil` during the time that either of these hooks is running. This means that if one of these functions changes the buffer, that change won't run these functions. If you do want the hook function to be run recursively, write your hook functions to bind these variables back to their usual values.

Variable: **first-change-hook**

This variable is a normal hook; its hook functions are run using `run-hooks` whenever a buffer is changed that was previously in the unmodified state.

The variables described in this section are meaningful only starting with Emacs version 19.

Go to the previous, next section.

Go to the previous, next section.

Searching and Matching

GNU Emacs provides two ways to search through a buffer for specified text: exact string searches and regular expression searches. After a regular expression search, you can identify the text matched by parts of the regular expression by examining the match data.

Searching for Strings

These are the primitive functions for searching through the text in a buffer. They are meant for use in programs, but you may call them interactively. If you do so, they prompt for the search string; limit and noerror are set to `nil`, and repeat is set to 1.

Command: **search-forward** *string &optional limit noerror repeat*

This function searches forward from point for an exact match for string. If successful, it sets point to the end of the occurrence found, and returns the new value of point. If no match is found, the value and side effects depend on noerror (see below).

In the following example, point is positioned at the beginning of the line. Then `(search-forward "fox")` is evaluated in the minibuffer and point is left after the last letter of `fox':

```
---------- Buffer: foo ----------
-!-The quick brown fox jumped over the lazy dog.
---------- Buffer: foo ----------

(search-forward "fox")
     => t

---------- Buffer: foo ----------
The quick brown fox-!- jumped over the lazy dog.
---------- Buffer: foo ----------
```

The argument limit specifies the upper bound to the search. (It must be a position in the current buffer.) No match extending after that position is accepted. If limit is omitted or `nil`, it defaults to the end of the accessible portion of the buffer.

What happens when the search fails depends on the value of noerror. If noerror is `nil`, a `search-failed` error is signaled. If noerror is `t`, `search-forward` returns `nil` and does nothing. If noerror is neither `nil` nor `t`, then `search-forward` moves point to the upper bound and returns `nil`. (It would be more consistent now to return the new position of point in that case, but some programs may depend on a value of `nil`.)

If repeat is non-`nil`, then the search is repeated that many times. Point is positioned at the end of the last

match.

Command: **search-backward** *string &optional limit noerror repeat*

This function searches backward from point for string. It is just like `search-forward` except that it searches backwards and leaves point at the beginning of the match.

Command: **word-search-forward** *string &optional limit noerror repeat*

This function searches forward from point for a "word" match for string. If it finds a match, it sets point to the end of the match found, and returns the new value of point.

A word search differs from a simple string search in that a word search **requires** that the words it searches for are present as entire words (searching for the word `ball' does not match the word `balls'), and punctuation and spacing are ignored (searching for `ball boy' does match `ball. Boy!').

In this example, point is first placed at the beginning of the buffer; the search leaves it between the y and the !.

```
---------- Buffer: foo ----------
-!-He said "Please!  Find
the ball boy!"
---------- Buffer: foo ----------

(word-search-forward "Please find the ball, boy.")
     => t

---------- Buffer: foo ----------
He said "Please!  Find
the ball boy-!-!"
---------- Buffer: foo ----------
```

If limit is non-`nil` (it must be a position in the current buffer), then it is the upper bound to the search. The match found must not extend after that position.

If noerror is `t`, then `word-search-forward` returns `nil` when a search fails, instead of signaling an error. If noerror is neither `nil` nor `t`, then `word-search-forward` moves point to limit (or the end of the buffer) and returns `nil`.

If repeat is non-`nil`, then the search is repeated that many times. Point is positioned at the end of the last match.

Command: **word-search-backward** *string &optional limit noerror repeat*

This function searches backward from point for a word match to string. This function is just like `word-search-forward` except that it searches backward and normally leaves point at the beginning of the match.

Regular Expressions

A regular expression (regexp, for short) is a pattern that denotes a (possibly infinite) set of strings. Searching for matches for a regexp is a very powerful operation. This section explains how to write regexps; the following section says how to search for them.

Syntax of Regular Expressions

Regular expressions have a syntax in which a few characters are special constructs and the rest are ordinary. An ordinary character is a simple regular expression which matches that character and nothing else. The special characters are `$', `^', `.', `*', `+', `?', `[', `]' and `\'; no new special characters will be defined in the future. Any other character appearing in a regular expression is ordinary, unless a `\' precedes it.

For example, `f' is not a special character, so it is ordinary, and therefore `f' is a regular expression that matches the string `f' and no other string. (It does *not* match the string `ff'.) Likewise, `o' is a regular expression that matches only `o'.

Any two regular expressions a and b can be concatenated. The result is a regular expression which matches a string if a matches some amount of the beginning of that string and b matches the rest of the string.

As a simple example, we can concatenate the regular expressions `f' and `o' to get the regular expression `fo', which matches only the string `fo'. Still trivial. To do something more powerful, you need to use one of the special characters. Here is a list of them:

. (Period)

is a special character that matches any single character except a newline. Using concatenation, we can make regular expressions like `a.b' which matches any three-character string which begins with `a' and ends with `b'.

*

is not a construct by itself; it is a suffix that means the preceding regular expression is to be repeated as many times as possible. In `fo*', the `*' applies to the `o', so `fo*' matches one `f' followed by any number of `o's. The case of zero `o's is allowed: `fo*' does match `f'.

`*' always applies to the *smallest* possible preceding expression. Thus, `fo*' has a repeating `o', not a repeating `fo'.

The matcher processes a `*' construct by matching, immediately, as many repetitions as can be found. Then it continues with the rest of the pattern. If that fails, backtracking occurs, discarding some of the matches of the `*'-modified construct in case that makes it possible to match the rest of the pattern. For example, matching `ca*ar' against the string `caaar', the `a*' first tries to match all three `a's; but the rest of the pattern is `ar' and there is only `r' left to match, so this try fails. The next alternative is for `a*' to match only two `a's. With this choice, the rest of the regexp matches successfully.

+

is a suffix character similar to `*' except that it must match the preceding expression at least once. So, for example, `ca+r' will match the strings `car' and `caaaar' but not the string `cr', whereas `ca*r' would match all three strings.

?

is a suffix character similar to `*' except that it can match the preceding expression either once or not at all. For example, `ca?r' will match `car' or `cr'; nothing else.

[...]

`[' begins a character set, which is terminated by a `]'. In the simplest case, the characters between the two form the set. Thus, `[ad]' matches either one `a' or one `d', and `[ad]*' matches any string composed of just `a's and `d's (including the empty string), from which it follows that `c[ad]*r' matches `cr', `car', `cdr', `caddaar', etc.

Character ranges can also be included in a character set, by writing two characters with a `-' between them. Thus, `[a-z]' matches any lower case letter. Ranges may be intermixed freely with individual characters, as in `[a-z$%.]', which matches any lower case letter or `$', `%' or a period.

Note that the usual special characters are not special any more inside a character set. A completely different set of special characters exists inside character sets: `]', `-' and `^'.

To include a `]' in a character set, make it the first character. For example, `[]a]' matches `]' or `a'. To include a `-', write `-' as the first or last character in the range.

To include `^', make it other than the first character in the set.

[^ ...]

`[^' begins a complement character set, which matches any character except the ones specified. Thus, `[^a-z0-9A-Z]' matches all characters *except* letters and digits.

`^' is not special in a character set unless it is the first character. The character following the `^' is treated as if it were first (thus, `-' and `]' are not special there).

Note that a complement character set can match a newline, unless newline is mentioned as one of the characters not to match.

^

is a special character that matches the empty string, but only at the beginning of a line in the text being matched. Otherwise it fails to match anything. Thus, `^foo' matches a `foo' which occurs at the beginning of a line.

When matching a string, `^' matches at the beginning of the string or after a newline character `\n'.

$

is similar to `^' but matches only at the end of a line. Thus, `x+$' matches a string of one `x' or more at the end of a line.

When matching a string, `$' matches at the end of the string or before a newline character `\n'.

\

has two functions: it quotes the special characters (including `\'), and it introduces additional special constructs.

Because `\' quotes special characters, `\$' is a regular expression which matches only `$', and `\[' a regular expression which matches only `[', and so on.

Note that `\' also has special meaning in the read syntax of Lisp strings (see section String Type), and must be quoted with `\'. For example, the regular expression that matches the `\' character is `\\'. To write a Lisp string that contains the characters `\\', Lisp syntax requires you to quote each `\' with another `\'. Therefore, the read syntax for a regular expression matching `\' is `"\\\\"'.

Please note: for historical compatibility, special characters are treated as ordinary ones if they are in contexts where their special meanings make no sense. For example, `*foo' treats `*' as ordinary since there is no preceding expression on which the `*' can act. It is poor practice to depend on this behavior; better to quote the special character anyway, regardless of where it appears.

For the most part, `\' followed by any character matches only that character. However, there are several exceptions: characters which, when preceded by `\', are special constructs. Such characters are always ordinary when encountered on their own. Here is a table of `\' constructs:

\|

specifies an alternative. Two regular expressions a and b with `\|' in between form an expression that matches anything that either a or b matches.

Thus, `foo\|bar' matches either `foo' or `bar' but no other string.

`\|' applies to the largest possible surrounding expressions. Only a surrounding `\(... \)' grouping can limit the grouping power of `\|'.

Full backtracking capability exists to handle multiple uses of `\|'.

\(... \)

is a grouping construct that serves three purposes:

To enclose a set of `\|' alternatives for other operations. Thus, `\(foo\|bar\)x' matches either `foox' or `barx'.

To enclose a complicated expression for a suffix character such as `*' to operate on. Thus, `ba\(na\)*' matches `bananana', etc., with any (zero or more) number of `na' strings.

To record a matched substring for future reference.

This last application is not a consequence of the idea of a parenthetical grouping; it is a separate feature which happens to be assigned as a second meaning to the same `\(... \)' construct because there is no conflict in practice between the two meanings. Here is an explanation of this feature:

\digit

matches the same text which is matched the digitth time by a previous `\(... \)' construct.

In other words, after the end of a `\( ... \)' construct. the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use `\' followed by digit to mean "match the same text matched the digitth time by the `\(... \)' construct."

The strings matching the first nine `\(... \)' constructs appearing in a regular expression are assigned numbers 1 through 9 in the order that the open parentheses appear in the regular

expression. So you can use `\1' through `\9' to refer to the text matched by the corresponding `\(\)' constructs.

For example, `\(.*\)\1' matches any newline-free string that is composed of two identical halves. The `\(.*\)' matches the first half, which may be anything, but the `\1' that follows must match the same exact text.

\`

matches the empty string, provided it is at the beginning of the buffer.

\'

matches the empty string, provided it is at the end of the buffer.

\=

matches the empty string, provided it is at point.

\b

matches the empty string, provided it is at the beginning or end of a word. Thus, `\bfoo\b' matches any occurrence of `foo' as a separate word. `\bballs?\b' matches `ball' or `balls' as a separate word.

\B

matches the empty string, provided it is *not* at the beginning or end of a word.

\<

matches the empty string, provided it is at the beginning of a word.

\>

matches the empty string, provided it is at the end of a word.

\w

matches any word-constituent character. The editor syntax table determines which characters these are. See section Syntax Tables.

\W

matches any character that is not a word-constituent.

\scode

matches any character whose syntax is code. Here code is a character which represents a syntax code: thus, `w' for word constituent, `-' for whitespace, `(' for open parenthesis, etc. See section Syntax Tables, for a list of the codes.

\Scode

matches any character whose syntax is not code.

Not every string is a valid regular expression. For example, any string with unbalanced square brackets is invalid, and so is a string that ends with a single `\'. If an invalid regular expression is passed to any of the search functions, an `invalid-regexp` error is signaled.

Function: **regexp-quote** *string*

This function returns a regular expression string which matches exactly string and nothing else. This allows you to request an exact string match when calling a function that wants a regular expression.


```
(regexp-quote "^The cat$")
     => "\\^The cat\\$"
```

One use of `regexp-quote` is to combine an exact string match with context described as a regular expression. For example, this searches for the string which is the value of `string`, surrounded by whitespace:

```
(re-search-forward
 (concat "\\s " (regexp-quote string) "\\s "))
```

Complex Regexp Example

Here is a complicated regexp, used by Emacs to recognize the end of a sentence together with any whitespace that follows. It is the value of the variable `sentence-end`.

First, we show the regexp as a string in Lisp syntax to enable you to distinguish the spaces from the tab characters. The string constant begins and ends with a double-quote. `\"' stands for a double-quote as part of the string, `\\' for a backslash as part of the string, `\t' for a tab and `\n' for a newline.

```
"[.?!][]\"')}]*\\($\\|\t\\|  \\)[ \t\n]*"
```

In contrast, if you evaluate the variable `sentence-end`, you will see the following:

```
sentence-end
=>
"[.?!][]\"')}]*\\($\\|  \\|  \\)[
]*"
```

In this case, the tab and carriage return are the actual characters.

This regular expression contains four parts in succession and can be deciphered as follows:

`[.?!]`

The first part of the pattern consists of three characters, a period, a question mark and an exclamation mark, within square brackets. The match must begin with one of these three characters.

`[]\"')}]*`

The second part of the pattern matches any closing braces and quotation marks, zero or more of them, that may follow the period, question mark or exclamation mark. The `\"` is Lisp syntax for a double-quote in a string. The `*' at the end indicates that the immediately preceding regular expression (a character set, in this case) may be repeated zero or more times.

`\\($\\|\t\\| \\)`

The third part of the pattern matches the whitespace that follows the end of a sentence: the end of a line, or a tab, or two spaces. The double backslashes are needed to prevent Emacs from reading the parentheses and vertical bars as part of the search pattern; the parentheses are used to mark the

group and the vertical bars are used to indicated that the patterns to either side of them are alternatives. The dollar sign is used to match the end of a line. The tab character is written using `\t' and the two spaces are written as themselves.

`[ \t\n]*`

Finally, the last part of the pattern indicates that the end of the line or the whitespace following the period, question mark or exclamation mark may, but need not, be followed by additional whitespace.

Regular Expression Searching

In GNU Emacs, you can search for the next match for a regexp either incrementally or not. Incremental search commands are described in the The GNU Emacs Manual. See section 'Regular Expression Search' in The GNU Emacs Manual. Here we describe only the search functions useful in programs. The principal one is `re-search-forward`.

Command: **re-search-forward** *regexp &optional limit noerror repeat*

This function searches forward in the current buffer for a string of text that is matched by the regular expression regexp. The function skips over any amount of text that is not matched by regexp, and leaves point at the end of the first string found that does match.

If the search is successful (i.e., if text matching regexp is found), then point moves to the end of that text, and the function returns the new value of point.

What happens when the search fails depends on the value of noerror. If noerror is `nil`, a `search-failed` error is signaled. If noerror is `t`, `re-search-forward` does nothing and returns `nil`. If noerror is neither `nil` nor `t`, then `re-search-forward` moves point to limit (or the end of the buffer) and returns `nil`.

If limit is non-`nil` (it must be a position in the current buffer), then it is the upper bound to the search. No match extending after that position is accepted.

If repeat is supplied (it must be a positive number), then the search is repeated that many times (each time starting at the end of the previous time's match). The call succeeds if all these searches succeeded, and point is left at the end of the match found by the last search. Otherwise the search fails.

In the following example, point is initially located directly before the `T'. After evaluating the form, point is located at the end of that line (between the `t' of `hat' and before the newline).

```
---------- Buffer: foo ----------
I read "-!-The cat in the hat
comes back" twice.
---------- Buffer: foo ----------

(re-search-forward "[a-z]+" nil t 5)
     => t
```


```
---------- Buffer: foo ----------
I read "The cat in the hat-!-
comes back" twice.
---------- Buffer: foo ----------
```

Command: **re-search-backward** *regexp &optional limit noerror repeat*

This function searches backward in the current buffer for a string of text that is matched by the regular expression regexp, leaving point at the beginning of the first text found.

This function is analogous to `re-search-forward`, but they are not simple mirror images. `re-search-forward` finds the match whose beginning is as close as possible. If `re-search-backward` were a perfect mirror image, it would find the match whose end is as close as possible. However, in fact it finds the match whose beginning is as close as possible. The reason is that matching a regular expression at a given spot always works from beginning to end, and is done at a specified beginning position. Thus, true mirror-image behavior would require a special feature for matching regexps from end to beginning.

Function: **string-match** *regexp string &optional start*

This function returns the index of the start of the first match for the regular expression regexp in string, or `nil` if there is no match. If start is non-`nil`, the search starts at that index in string.

For example,

```
(string-match
 "quick" "The quick brown fox jumped quickly.")
     => 4
(string-match
 "quick" "The quick brown fox jumped quickly." 8)
     => 27
```

The index of the first character of the string is 0, the index of the second character is 1, and so on.

After this function returns, the index of the first character beyond the match is available as `(match-end 0)`. See section The Match Data.

```
(string-match
 "quick" "The quick brown fox jumped quickly." 8)
     => 27

(match-end 0)
     => 32
```

The `match-beginning` and `match-end` functions are described together; see section The Match Data.

Function: **looking-at** *regexp*

This function determines whether the text in the current buffer directly following point matches the regular expression regexp. "Directly following" means precisely that: the search is "anchored" and it must succeed starting with the first character following point. The result is `t` if so, `nil` otherwise.

This function does not move point, but it updates the match data, which you can access using `match-beginning` or `match-end`. See section The Match Data.

In this example, point is located directly before the `T'. If it were anywhere else, the result would be `nil`.

```
---------- Buffer: foo ----------
I read "-!-The cat in the hat
comes back" twice.
---------- Buffer: foo ----------

(looking-at "The cat in the hat$")
     => t
```

Replacement

Function: **perform-replace** *from-string replacements query-flag regexp-flag delimited-flag &optional repeat-count map*

This function is the guts of `query-replace` and related commands. It searches for occurrences of from-string and replaces some or all of them. If query-flag is `nil`, it replaces all occurrences; otherwise, it asks the user what to do about each one.

If regexp-flag is non-`nil`, then from-string is considered a regular expression; otherwise, it must match literally. If delimited-flag is non-`nil`, then only replacements surrounded by word boundaries are considered.

The argument replacements specifies what to replace occurrences with. If it is a string, that string is used. It can also be a list of strings, to be used in cyclic order.

If repeat-count is non-`nil`, it should be an integer, the number of occurrences to consider. In this case, `perform-replace` returns after considering that many occurrences.

Normally, the keymap `query-replace-map` defines the possible user responses. The argument map, if non-`nil`, is a keymap to use instead of `query-replace-map`.

Variable: **query-replace-map**

This variable holds a special keymap that defines the valid user responses for `query-replace` and related functions, as well as `y-or-n-p` and `map-y-or-n-p`. It is special in two ways:

- The "key bindings" are not commands, just symbols that are meaningful to the functions that use this map.
- Prefix keys are not supported; each key binding must be for a single event key sequence. This is because the functions don't use read key sequence to get the input; instead, they read a single event

and look it up "by hand."

Here are the meaningful "bindings" for `query-replace-map`. Several of them are meaningful only for `query-replace` and friends.

`act`
: Do take the action. The action being considered--in other words, "yes."

`skip`
: Do not take action for this question--in other words, "no."

`exit`
: Answer this question "no," and don't ask any more.

`act-and-exit`
: Answer this question "yes," and don't ask any more.

`act-and-show`
: Answer this question "yes," but show the results--don't advance yet.

`automatic`
: Answer this question and all subsequent questions in the series with "yes," without further user interaction.

`backup`
: Move back to the previous place that a question was asked about.

`edit`
: Enter a recursive edit to deal with this item--instead of any other answer.

`delete-and-edit`
: Delete the text being considered, then enter a recursive edit to replace it.

`recenter`
: Redisplay and center the window, then ask the same question again.

`quit`
: Perform a quit right away. Only the `y-or-n-p` functions use this answer.

`help`
: Display some help, then ask again.

The Match Data

Emacs keeps track of the positions of the start and end of segments of text found during a regular expression search. This means, for example, that you can search for a complex pattern, such as a date in an Rmail message, and extract parts of it.

Because the match data normally describe the most recent search only, you must be careful not to do another search inadvertently between the search you wish to refer back to and the use of the match data. If you can't avoid another intervening search, you must save and restore the match data around it, to

prevent it from being overwritten.

Simple Match Data Access

This section explains how to use the match data to find the starting point or ending point of the text that was matched by a particular search, or by a particular parenthetical subexpression of a regular expression.

Function: **match-beginning** *count*

This function returns the position of the start of text matched by the last regular expression searched for. count, a number, specifies a subexpression whose start position is the value. If count is zero, then the value is the position of the text matched by the whole regexp. If count is greater than zero, then the value is the position of the beginning of the text matched by the countth subexpression, regardless of whether it was used in the final match.

Subexpressions of a regular expression are those expressions grouped inside of parentheses, `\(...\)'. The countth subexpression is found by counting occurrences of `\(' from the beginning of the whole regular expression. The first subexpression is numbered 1, the second 2, and so on.

The value is `nil` for a parenthetical grouping inside of a `\|' alternative that wasn't used in the match.

The `match-end` function is similar to the `match-beginning` function except that it returns the position of the end of the matched text.

Here is an example, with a comment showing the numbers of the positions in the text:

```
(string-match
 "\\(qu\\)\\(ick\\)" "The quick fox jumped quickly.")
     => 4            ;^^^^^^^^^^
                     ;0123456789

(match-beginning 1)              ; The beginning of the match
     => 4                        ;   with `qu' is at index 4.

(match-beginning 2)              ; The beginning of the match
     => 6                        ;   with `ick' is at index 6.

(match-end 1)                    ; The end of the match
     => 6                        ;   with `qu' is at index 6.

(match-end 2)                    ; The end of the match
     => 9                        ;   with `ick' is at index 9.
```

Here is another example. Before the form is evaluated, point is located at the beginning of the line. After evaluating the search form, point is located on the line between the space and the word `in'. The beginning of the entire match is at the 9th character of the buffer (`T'), and the beginning of the match for the first subexpression is at the 13th character (`c').


```
(list
  (re-search-forward "The \\(cat \\)")
  (match-beginning 0)
  (match-beginning 1))
    => (t 9 13)

---------- Buffer: foo ----------
I read "The cat -!-in the hat comes back" twice.
        ^   ^
        9  13
---------- Buffer: foo ----------
```

(Note that in this case, the index returned is a buffer position; the first character of the buffer counts as 1.)

Function: **match-end** *count*

This function returns the position of the end of text matched by the last regular expression searched for. This function is otherwise similar to `match-beginning`.

Replacing the Text That Matched

Function: **replace-match** *replacement &optional fixedcase literal*

This function replaces the text matched by the last search with replacement.

If fixedcase is non-`nil`, then the case of the replacement text is not changed; otherwise, the replacement text is converted to a different case depending upon the capitalization of the text to be replaced. If the original text is all upper case, the replacement text is converted to upper case, except when all of the words in the original text are only one character long. In that event, the replacement text is capitalized. If *all* of the words in the original text are capitalized, then all of the words in the replacement text are capitalized.

If literal is non-`nil`, then replacement is inserted exactly as it is, the only alterations being case changes as needed. If it is `nil` (the default), then the character `\' is treated specially. If a `\' appears in replacement, then it must be part of one of the following sequences:

`\&'

: `\&' stands for the entire text being replaced.

`\n'

: `\n' stands for the nth subexpression in the original regexp. Subexpressions are those expressions grouped inside of `\(...\)'. n is a digit.

`\\'

: `\\' stands for a single `\' in the replacement text.

`replace-match` leaves point at the end of the replacement text, and returns `t`.

Accessing the Entire Match Data

The functions `match-data` and `store-match-data` let you read or write the entire match data, all at once.

Function: **match-data**

This function returns a new list containing all the information on what text the last search matched. Element zero is the position of the beginning of the match for the whole expression; element one is the position of the end of the match for the expression. The next two elements are the positions of the beginning and end of the match for the first subexpression. In general, element corresponds to `(match-beginning n)`; and element corresponds to `(match-end n)`.

All the elements are markers or `nil` if matching was done on a buffer, and all are integers or `nil` if matching was done on a string with `string-match`. (In Emacs 18 and earlier versions, markers were used even for matching on a string, except in the case of the integer 0.)

As always, there must be no possibility of intervening searches between the call to a search function and the call to `match-data` that is intended to access the match-data for that search.

```
(match-data)
     =>  (#<marker at 9 in foo>
          #<marker at 17 in foo>
          #<marker at 13 in foo>
          #<marker at 17 in foo>)
```

Function: **store-match-data** *match-list*

This function sets the match data from the elements of match-list, which should be a list that was the value of a previous call to `match-data`.

If match-list refers to a buffer that doesn't exist, you don't get an error; that sets the match data in a meaningless but harmless way.

Saving and Restoring the Match Data

All asynchronous process functions (filters and sentinels) and functions that use `recursive-edit` should save and restore the match data if they do a search or if they let the user type arbitrary commands. Saving the match data is useful in other cases as well--whenever you want to access the match data resulting from an earlier search, notwithstanding another intervening search.

This example shows the problem that can arise if you fail to attend to this requirement:

```
(re-search-forward "The \\(cat \\)")
     => 48
(foo)                   ; Perhaps foo does
                        ;   more searching.
```


```
(match-end 0)
     => 61               ; Unexpected result--not 48!
```

In Emacs versions 19 and later, you can save and restore the match data with `save-match-data`:

Special Form: **save-match-data** *body...*

This special form executes body, saving and restoring the match data around it. This is useful if you wish to do a search without altering the match data that resulted from an earlier search.

You can use `store-match-data` together with `match-data` to imitate the effect of the special form `save-match-data`. This is useful for writing code that can run in Emacs 18. Here is how:

```
(let ((data (match-data)))
  (unwind-protect
      ...   ; May change the original match data.
    (store-match-data data)))
```

Standard Regular Expressions Used in Editing

Here are the regular expressions standardly used in editing:

Variable: **page-delimiter**

This is the regexp describing line-beginnings that separate pages. The default value is `"^\014"` (i.e., `"^^L"` or `"^\C-l"`).

Variable: **paragraph-separate**

This is the regular expression for recognizing the beginning of a line that separates paragraphs. (If you change this, you may have to change `paragraph-start` also.) The default value is `"^[ \t\f]*$"`, which is a line that consists entirely of spaces, tabs, and form feeds.

Variable: **paragraph-start**

This is the regular expression for recognizing the beginning of a line that starts *or* separates paragraphs. The default value is `"^[ \t\n\f]"`, which matches a line starting with a space, tab, newline, or form feed.

Variable: **sentence-end**

This is the regular expression describing the end of a sentence. (All paragraph boundaries also end sentences, regardless.) The default value is:

```
"[.?!][]\"')}]*\\($\\|\t\\| \\)[ \t\n]*"
```

This means a period, question mark or exclamation mark, followed by a closing brace, followed by tabs, spaces or new lines.

For a detailed explanation of this regular expression, see section Complex Regexp Example.

Searching and Case

By default, searches in Emacs ignore the case of the text they are searching through; if you specify searching for `FOO', then `Foo' or `foo' is also considered a match. Regexps, and in particular character sets, are included: thus, `[aB]' would match `a' or `A' or `b' or `B'.

If you do not want this feature, set the variable `case-fold-search` to `nil`. Then all letters must match exactly, including case. This is a per-buffer-local variable; altering the variable affects only the current buffer. (See section Introduction to Buffer-Local Variables.) Alternatively, you may change the value of `default-case-fold-search`, which is the default value of `case-fold-search` for buffers that do not override it.

User Option: **case-replace**

This variable determines whether `query-replace` should preserve case in replacements. If the variable is `nil`, then case need not be preserved.

User Option: **case-fold-search**

This buffer-local variable determines whether searches should ignore case. If the variable is `nil` they do not ignore case; otherwise they do ignore case.

Variable: **default-case-fold-search**

The value of this variable is the default value for `case-fold-search` in buffers that do not override it. This is the same as `(default-value 'case-fold-search)`.

Go to the previous, next section.

Go to the previous, next section.

Syntax Tables

A syntax table provides Emacs with the information that determines the syntactic use of each character in a buffer. This information is used by the parsing commands, the complex movement commands, and others to determine where words, symbols, and other syntactic constructs begin and end. The current syntax table controls the meaning of the word motion functions (see section Motion by Words) and the list motion functions (see section Moving over Balanced Expressions) as well as the functions in this chapter.

A syntax table is a vector of 256 elements; it contains one entry for each of the 256 ASCII characters of an 8-bit byte. Each element is an integer that encodes the syntax of the character in question.

Syntax tables are used only for moving across text, not for the GNU Emacs Lisp reader. GNU Emacs Lisp uses built-in syntactic rules when reading Lisp expressions, and these rules cannot be changed.

Each buffer has its own major mode, and each major mode has its own idea of the syntactic class of various characters. For example, in Lisp mode, the character `;' begins a comment, but in C mode, it terminates a statement. To support these variations, Emacs makes the choice of syntax table local to each buffer. Typically, each major mode has its own syntax table and installs that table in each buffer which uses that mode. Changing this table alters the syntax in all those buffers as well as in any buffers subsequently put in that mode. Occasionally several similar modes share one syntax table. See section Major Mode Examples, for an example of how to set up a syntax table.

Function: **syntax-table-p** *object*

This function returns `t` if object is a vector of length 256 elements. This means that the vector may be a syntax table. However, according to this test, any vector of length 256 is considered to be a syntax table, no matter what its contents.

Syntax Descriptors

This section describes the syntax classes and flags that denote the syntax of a character, and how they are represented as a syntax descriptor, which is a Lisp string that you pass to `modify-syntax-entry` to specify the desired syntax.

Emacs defines twelve syntax classes. Each syntax table puts each character into one class. There is no necessary relationship between the class of a character in one syntax table and its class in any other table.

Each class is designated by a mnemonic character which serves as the name of the class when you need to specify a class. Usually the designator character is one which is frequently put in that class; however, its meaning as a designator is unvarying and independent of how it is actually classified.

A syntax descriptor is a Lisp string which specifies a syntax class, a matching character (unused except for parenthesis classes) and flags. The first character is the designator for a syntax class. The second

character is the character to match; if it is unused, put a space there. Then come the characters for any desired flags. If no matching character or flags are needed, one character is sufficient.

Thus, the descriptor for the character `*' in C mode is `. 23' (i.e., punctuation, matching character slot unused, second character of a comment-starter, first character of an comment-ender), and the entry for `/' is `. 14' (i.e., punctuation, matching character slot unused, first character of a comment-starter, second character of a comment-ender).

Table of Syntax Classes

Here is a summary of the classes, the characters that stand for them, their meanings, and examples of their use.

Syntax class: **whitespace character**

Whitespace characters (designated with ` ' or `-') separate symbols and words from each other. Typically, whitespace characters have no other syntactic use, and multiple whitespace characters are syntactically equivalent to a single one. Space, tab, newline and formfeed are almost always considered whitespace.

Syntax class: **word constituent**

Word constituents (designated with `w') are parts of normal English words and are typically used in variable and command names in programs. All upper and lower case letters and the digits are typically word constituents.

Syntax class: **symbol constituent**

Symbol constituents (designated with `_') are the extra characters that are used in variable and command names along with word constituents. For example, the symbol constituents class is used in Lisp mode to indicate that certain characters may be part of symbol names even though they are not part of English words. These characters are `$&*+-_<>'. In standard C, the only non-word-constituent character that is valid in symbols is underscore (`_').

Syntax class: **punctuation character**

Punctuation characters (`.') are those characters that are used as punctuation in English, or are used in some way in a programming language to separate symbols from one another. Most programming language modes, including Emacs Lisp mode, have no characters in this class since the few characters that are not symbol or word constituents all have other uses.

Syntax class: **open parenthesis character**

Syntax class: **close parenthesis character**

Open and close parenthesis characters are characters used in dissimilar pairs to surround sentences or expressions. Such a grouping is begun with an open parenthesis character and terminated with a close. Each open parenthesis character matches a particular close parenthesis character, and vice versa. Normally, Emacs indicates momentarily the matching open parenthesis when you insert a close parenthesis. See section Blinking.

The class of open parentheses is designated with `(', and that of close parentheses with `)'.

In English text, and in C code, the parenthesis pairs are `()', `[]', and `{}'. In Emacs Lisp, the delimiters for lists and vectors (`()' and `[]') are classified as parenthesis characters.

Syntax class: **string quote**

String quote characters (designated with `"') is used to delimit string constants in many languages, including Lisp and C. The same string quote character appears at the beginning and the end of a string. Such quoted strings do not nest.

The parsing facilities of Emacs consider a string as a single token. The usual syntactic meanings of the characters in the string are suppressed.

The Lisp modes have two string quote characters: double-quote (`"') and vertical bar (`|'). `|' is not used in Emacs Lisp, but it is used in Common Lisp. C also has two string quote characters: double-quote for strings, and single-quote (`') for character constants.

English text has no string quote characters because English is not a programming language. Although quotation marks are used in English, we do not want them to turn off the usual syntactic properties of other characters in the quotation.

Syntax class: **escape**

An escape character (designated with `\') starts an escape sequence such as is used in C string and character constants. The character `\' belongs to this class in both C and Lisp. (In C, it is used thus only inside strings, but it turns out to cause no trouble to treat it this way throughout C code.)

Characters in this class count as part of words if `words-include-escapes` is non-`nil`. See section Motion by Words.

Syntax class: **character quote**

A character quote character (designated with `/') quotes the following character so that it loses its normal syntactic meaning. This differs from an escape character in that only the character immediately following is ever affected.

Characters in this class count as part of words if `words-include-escapes` is non-`nil`. See section Motion by Words.

This class is not currently used in any standard Emacs modes.

Syntax class: **paired delimiter**

Paired delimiter characters (designated with `$') are like string quote characters except that the syntactic properties of the characters between the delimiters are not suppressed. Only TeX mode uses a paired identical delimiter presently--the `$' that begins and ends math mode.

Syntax class: **expression prefix**

An expression prefix operator (designated with `') is used for syntactic operators that are part of an expression if they appear next to one but are not part of an adjoining symbol. These characters in Lisp

include the apostrophe, `'' (used for quoting), the comma, `,' (used in macros), and `#' (used in the rea syntax for certain data types).

Syntax class: **comment starter**

Syntax class: **comment ender**

The comment starter and comment ender characters are used in different languages to delimit comments. These classes are designated with `<' and `>', respectively.

English text has no comment characters. In Lisp, the semicolon (`;') starts a comment and a newline or formfeed ends one.

Syntax Flags

In addition to the classes, entries for characters in a syntax table can include flags. There are six possible flags, represented by the characters `1', `2', `3', `4', `b' and `p'.

All the flags except `p' are used to describe multi-character comment delimiters. The digit flags indicate that a character can *also* be part of a comment sequence, in addition to the syntactic properties associated with its character class. The flags are independent of the class and each other for the sake of characters such as `*' in C mode, which is a punctuation character, *and* the second character of a start-of-comment sequence (`/*'), *and* the first character of an end-of-comment sequence (`*/').

The flags for a character c are:

- `1' means c is the start of a two-character comment start sequence.
- `2' means c is the second character of such a sequence.
- `3' means c is the start of a two-character comment end sequence.
- `4' means c is the second character of such a sequence.
- `b' means that c as a comment delimiter belongs to the alternative "b" comment style.

  Emacs can now supports two comment styles simultaneously. (This is for the sake of C++.) More specifically, it can recognize two different comment-start sequences. Both must share the same first character; only the second character may differ. Mark the second character of the "b"-style comment start sequence with the `b' flag.

  The two styles of comment can have different comment-end sequences. A comment-end sequence (one or two characters) applies to the "b" style if its first character has the `b' flag set; otherwise, it applies to the "a" style.

  The appropriate comment syntax settings for C++ are as follows:
  `/'
  `124b'
  `*'
  `23'
  newline

`>b'

Thus `/*' is a comment-start sequence for "a" style, `//' is a comment-start sequence for "b" style, `*/' is a comment-end sequence for "a" style, and newline is a comment-end sequence for "b" style.

- `p' identifies an additional "prefix character" for Lisp syntax. These characters are treated as whitespace when they appear between expressions. When they appear within an expression, they are handled according to their usual syntax codes.

  The function `backward-prefix-chars` moves back over these characters, as well as over characters whose primary syntax class is prefix (`').

Syntax Table Functions

In this section we describe functions for creating, accessing and altering syntax tables.

Function: **make-syntax-table** *&optional table*

This function constructs a copy of table and returns it. If table is not supplied (or is `nil`), it returns a copy of the current syntax table. Otherwise, an error is signaled if table is not a syntax table.

Function: **copy-syntax-table** *&optional table*

This function is identical to `make-syntax-table`.

Command: **modify-syntax-entry** *char syntax-descriptor &optional table*

This function sets the syntax entry for char according to syntax-descriptor. The syntax is changed only for table, which defaults to the current buffer's syntax table, and not in any other syntax table. The argument syntax-descriptor specifies the desired syntax; this is a string beginning with a class designator character, and optionally containing a matching character and flags as well. See section Syntax Descriptors.

This function always returns `nil`. The old syntax information in the table for this character is discarded.

An error is signaled if the first character of the syntax descriptor is not one of the twelve syntax class designator characters. An error is also signaled if char is not a character.

```
Examples:

;; Put the space character in class whitespace.
(modify-syntax-entry ?\  " ")
     => nil

;; Make `$' an open parenthesis character,
;;   with `^' as its matching close.
(modify-syntax-entry ?$ "(^")
     => nil
```


```
;; Make `^' a close parenthesis character,
;;   with `$' as its matching open.
(modify-syntax-entry ?^ ")$")
     => nil

;; Make `/' a punctuation character,
;;   the first character of a start-comment sequence,
;;   and the second character of an end-comment sequence.
;;   This is used in C mode.
(modify-syntax-entry ?/ ".13")
     => nil
```

Function: **char-syntax** *character*

This function returns the syntax class of character, represented by its mnemonic designator character. This *only* returns the class, not any matching parenthesis or flags.

An error is signaled if char is not a character.

The first example shows that the syntax class of space is whitespace (represented by a space). The second example shows that the syntax of `/' is punctuation in C-mode. This does not show the fact that it is also a comment sequence character. The third example shows that open parenthesis is in the class of open parentheses. This does not show the fact that it has a matching character, `)'.

```
(char-to-string (char-syntax ?\ ))
     => " "

(char-to-string (char-syntax ?/))
     => "."

(char-to-string (char-syntax ?\())
     => "("
```

Function: **set-syntax-table** *table*

This function makes table the syntax table for the current buffer. It returns table.

Function: **syntax-table**

This function returns the current syntax table, which is the table for the current buffer.

Motion and Syntax

This section describes functions for moving across characters in certain syntax classes. None of these functions exists in Emacs version 18 or earlier.

Function: **skip-syntax-forward** *syntaxes &optional limit*

This function moves point forward across characters whose syntax classes are mentioned in syntaxes stops when it encounters the end of the buffer, or position lim (if specified), or a character it is not supposed to skip.

The return value is the distance traveled, which is a nonnegative integer.

Function: **skip-syntax-backward** *syntaxes &optional limit*

This function moves point backward across characters whose syntax classes are mentioned in syntaxes. It stops when it encounters the beginning of the buffer, or position lim (if specified), or a character it is not supposed to skip.

The return value indicates the distance traveled. It is an integer that is zero or less.

Function: **backward-prefix-chars**

This function moves point backward over any number of chars with expression prefix syntax. This includes both characters in the expression prefix syntax class, and characters with the `p' flag.

Parsing Balanced Expressions

Here are several functions for parsing and scanning balanced expressions. The syntax table controls the interpretation of characters, so these functions can be used for Lisp expressions when in Lisp mode and for C expressions when in C mode. See section Moving over Balanced Expressions, for convenient higher-level functions for moving over balanced expressions.

Function: **parse-partial-sexp** *start limit &optional target-depth stop-before state stop-comment*

This function parses an expression in the current buffer starting at start, not scanning past limit. Parsing stops at limit or when certain criteria described below are met; point is set to the location where parsing stops. The value returned is a description of the status of the parse at the point where it stops.

Normally, start is assumed to be the top level of an expression to be parsed, such as the beginning of a function definition. Alternatively, you might wish to resume parsing in the middle of an expression. To do this, you must provide a state argument that describes the initial status of parsing. If state is omitted (or `nil`), parsing assumes that start is the beginning of a new parse at level 0.

If the third argument target-depth is non-`nil`, parsing stops if the depth in parentheses becomes equal to target-depth. The depth starts at 0, or at whatever is given in state.

If the fourth argument stop-before is non-`nil`, parsing stops when it comes to any character that starts a sexp. If stop-comment is non-`nil`, parsing stops when it comes to the start of a comment.

The fifth argument state is a seven-element list of the same form as the value of this function, described below. The return value of one call may be used to initialize the state of the parse on another call to `parse-partial-sexp`.

The result is a list of seven elements describing the final state of the parse:

1. The depth in parentheses, starting at 0.

2. The character position of the start of the innermost containing parenthetical grouping; `nil` if r
3. The character position of the start of the last complete subexpression terminated; `nil` if none.
4. Non-`nil` if inside a string. (It is the character that will terminate the string.)
5. `t` if inside a comment.
6. `t` if point is just after a quote character.
7. The minimum parenthesis depth encountered during this scan.

Elements 1, 4, 5, and 6 are significant in the argument state.

This function is used to determine how to indent lines in programs written in languages that have nested parentheses.

Function: **scan-lists** *from count depth*

This function scans forward count balanced parenthetical groupings from character number from. It returns the character number of the position thus found.

If depth is nonzero, parenthesis depth counting begins from that value. The only candidates for stopping are places where the depth in parentheses becomes zero; `scan-lists` counts count such places and then stops. Thus, a positive value for depth means go out levels of parenthesis.

Comments are ignored if `parse-sexp-ignore-comments` is non-`nil`.

If the beginning or end of the buffer (or its accessible portion) is reached and the depth is not zero, an error is signaled. If the depth is zero but the count is not used up, `nil` is returned.

Function: **scan-sexps** *from count*

Scan from character number from by count balanced expressions. It returns the character number of the position thus found.

Comments are ignored if `parse-sexp-ignore-comments` is non-`nil`.

If the beginning or end of (the accessible part of) the buffer is reached in the middle of a parenthetical grouping, an error is signaled. If the beginning or end is reached between groupings but before count is used up, `nil` is returned.

Variable: **parse-sexp-ignore-comments**

If the value is non-`nil`, then comments are treated as whitespace by the functions in this section and by `forward-sexp`.

In older Emacs versions, this feature worked only when the comment terminator is something like `*/', and appears only to end a comment. In languages where newlines terminate comments, it was necessary make this variable `nil`, since not every newline is the end of a comment. This limitation no longer exists.

You can use `forward-comment` to move forward or backward over one comment or several comments.

Function: **forward-comment** *count*

This function moves point forward across count comments (backward, if count is negative). If it finds anything other than a comment or whitespace, it stops, leaving point at the place where it stopped. It also stops after satisfying count.

To move forward over all comments and whitespace following point, use `(forward-comment (buffer-size))`. `(buffer-size)` is a good argument to use, because the number of comments to skip cannot exceed that many.

Some Standard Syntax Tables

Each of the major modes in Emacs has its own syntax table. Here are several of them:

Function: **standard-syntax-table**

This function returns the standard syntax table, which is the syntax table used in Fundamental mode.

Variable: **text-mode-syntax-table**

The value of this variable is the syntax table used in Text mode.

Variable: **c-mode-syntax-table**

The value of this variable is the syntax table in use in C-mode buffers.

Variable: **emacs-lisp-mode-syntax-table**

The value of this variable is the syntax table used in Emacs Lisp mode by editing commands. (It has no effect on the Lisp `read` function.)

Syntax Table Internals

Each element of a syntax table is an integer that translates into the full meaning of the entry: class, possible matching character, and flags. However, it is not common for a programmer to work with the entries directly in this form since the Lisp-level syntax table functions usually work with syntax descriptors (see section Syntax Descriptors).

The low 8 bits of each element of a syntax table indicates the syntax class.

Integer
: *Class*

0
: whitespace

1
: punctuation

2

word

3

symbol

4

open parenthesis

5

close parenthesis

6

expression prefix

7

string quote

8

paired delimiter

9

escape

10

character quote

11

comment-start

12

comment-end

The next 8 bits are the matching opposite parenthesis (if the character has parenthesis syntax); otherwise, they are not meaningful. The next 6 bits are the flags.

Go to the previous, next section.

Go to the previous, next section.

Abbrevs And Abbrev Expansion

An abbreviation or abbrev is a string of characters that may be expanded to a longer string. The user can insert the abbrev string and find it replaced automatically with the expansion of the abbrev. This saves typing.

The set of abbrevs currently in effect is recorded in an abbrev table. Each buffer has a local abbrev table, but normally all buffers in the same major mode share one abbrev table. There is also a global abbrev table. Normally both are used.

An abbrev table is represented as an obarray containing a symbol for each abbreviation. The symbol's name is the abbreviation. Its value is the expansion; its function definition is the hook; its property list cell contains the use count, the number of times the abbreviation has been expanded. Because these symbols are not interned in the usual obarray, they will never appear as the result of reading a Lisp expression; in fact, they will never be used except by the code that handles abbrevs. Therefore, it is safe to use them in an extremely nonstandard way. See section Creating and Interning Symbols.

For the user-level commands for abbrevs, see section 'Abbrev Mode' in The GNU Emacs Manual.

Setting Up Abbrev Mode

Abbrev mode is a minor mode controlled by the value of the variable `abbrev-mode`.

Variable: **abbrev-mode**

A non-`nil` value of this variable turns on the automatic expansion of abbrevs when their abbreviations are inserted into a buffer. If the value is `nil`, abbrevs may be defined, but they are not expanded automatically.

This variable automatically becomes local when set in any fashion.

Variable: **default-abbrev-mode**

This is the value `abbrev-mode` for buffers that do not override it. This is the same as `(default-value 'abbrev-mode)`.

Abbrev Tables

This section describes how to create and manipulate abbrev tables.

Function: **make-abbrev-table**

This function creates and returns a new, empty abbrev table--an obarray containing no symbols. It is a vector filled with `nil`s.

Function: **clear-abbrev-table** *table*

This function undefines all the abbrevs in abbrev table table, leaving it empty. The function returns `nil`.

Function: **define-abbrev-table** *tabname definitions*

This function defines tabname (a symbol) as an abbrev table name, i.e., as a variable whose value is an abbrev table. It defines abbrevs in the table according to definitions, a list of elements of the form `(abbrevname expansion hook usecount)`. The value is always `nil`.

Variable: **abbrev-table-name-list**

This is a list of symbols whose values are abbrev tables. `define-abbrev-table` adds the new abbrev table name to this list.

Function: **insert-abbrev-table-description** *name &optional human*

This function inserts before point a description of the abbrev table named name. The argument name is a symbol whose value is an abbrev table. The value is always `nil`.

If human is non-`nil`, a human-oriented description is inserted. Otherwise the description is a Lisp expression--a call to `define-abbrev-table` which would define name exactly as it is currently defined.

Defining Abbrevs

These functions define an abbrev in a specified abbrev table. `define-abbrev` is the low-level basic function, while `add-abbrev` is used by commands that ask for information from the user.

Function: **add-abbrev** *table type arg*

This function adds an abbreviation to abbrev table table. The argument type is a string describing in English the kind of abbrev this will be (typically, `"global"` or `"mode-specific"`); this is used in prompting the user. The argument arg is the number of words in the expansion.

The return value is the symbol which internally represents the new abbrev, or `nil` if the user declines to redefine an existing abbrev.

Function: **define-abbrev** *table name expansion hook*

This function defines an abbrev in table named name, to expand to expansion, and call hook. The return value is an uninterned symbol which represents the abbrev inside Emacs; its name is name.

The argument name should be a string. The argument expansion should be a string, or `nil`, to undefine the abbrev.

The argument hook is a function or `nil`. If hook is non-`nil`, then it is called with no arguments after the abbrev is replaced with expansion; point is located at the end of expansion.

The use count of the abbrev is initialized to zero.

User Option: **only-global-abbrevs**

If this variable is non-`nil`, it means that the user plans to use global abbrevs only. This tells the commands that define mode-specific abbrevs to define global ones instead. This variable does not alter the functioning of the functions in this section; it is examined by their callers.

Saving Abbrevs in Files

A file of saved abbrev definitions is actually a file of Lisp code. The abbrevs are saved in the form of a Lisp program to define the same abbrev tables with the same contents. Therefore, you can load the file with `load` (see section How Programs Do Loading). However, the function `quietly-read-abbrev-file` is provided as a more convenient interface.

User-level facilities such as `save-some-buffers` can save abbrevs in a file automatically, under the control of variables described here.

User Option: **abbrev-file-name**

This is the default file name for reading and saving abbrevs.

Function: **quietly-read-abbrev-file** *filename*

This function reads abbrev definitions from a file named filename, previously written with `write-abbrev-file`. If filename is `nil`, the file specified in `abbrev-file-name` is used. `save-abbrevs` is set to `t` so that changes will be saved.

This function does not display any messages. It returns `nil`.

User Option: **save-abbrevs**

A non-`nil` value for `save-abbrev` means that Emacs should save abbrevs when files are saved. `abbrev-file-name` specifies the file to save the abbrevs in.

Variable: **abbrevs-changed**

This variable is set non-`nil` by defining or altering any abbrevs. This serves as a flag for various Emacs commands to offer to save your abbrevs.

Command: **write-abbrev-file** *filename*

Save all abbrev definitions, in all abbrev tables, in the file filename, in the form of a Lisp program which when loaded will define the same abbrevs. This function returns `nil`.

Looking Up and Expanding Abbreviations

Abbrevs are usually expanded by commands for interactive use, including `self-insert-command`. This section describes the subroutines used in writing such functions, as well as the variables they use for communication.

Function: **abbrev-symbol** *abbrev &optional table*

This function returns the symbol representing the abbrev named abbrev. The value returned is `nil` if that abbrev is not defined. The optional second argument table is the abbrev table to look it up in. By default, this function tries first the current buffer's local abbrev table, and second the global abbrev table.

User Option: **abbrev-all-caps**

When this is set non-`nil`, an abbrev entered entirely in upper case is expanded using all upper case. Otherwise, an abbrev entered entirely in upper case is expanded by capitalizing each word of the expansion.

Function: **abbrev-expansion** *abbrev &optional table*

This function returns the string that abbrev would expand into (as defined by the abbrev tables used for the current buffer). The optional argument table specifies the abbrev table to use; if it is specified, the abbrev is looked up in that table only.

Variable: **abbrev-start-location**

This is the buffer position for `expand-abbrev` to use as the start of the next abbrev to be expanded. (`nil` means use the word before point instead.) `abbrev-start-location` is set to `nil` each time `expand-abbrev` is called. This variable is also set by `abbrev-prefix-mark`.

Variable: **abbrev-start-location-buffer**

The value of this variable is the buffer for which `abbrev-start-location` has been set. Trying to expand an abbrev in any other buffer clears `abbrev-start-location`. This variable is set by `abbrev-prefix-mark`.

Variable: **last-abbrev**

This is the `abbrev-symbol` of the last abbrev expanded. This information is left by `expand-abbrev` for the sake of the `unexpand-abbrev` command.

Variable: **last-abbrev-location**

This is the location of the last abbrev expanded. This contains information left by `expand-abbrev` for the sake of the `unexpand-abbrev` command.

Variable: **last-abbrev-text**

This is the exact expansion text of the last abbrev expanded, as results from case conversion. Its value is `nil` if the abbrev has already been unexpanded. This contains information left by `expand-abbrev` for the sake of the `unexpand-abbrev` command.

Variable: **pre-abbrev-expand-hook**

This is a normal hook whose functions are executed, in sequence, just before any expansion of an abbrev. See section Hooks. Since it is a normal hook, the hook functions receive no arguments. However, they can find the abbrev to be expanded by looking in the buffer before point.

The following sample code shows a simple use of `pre-abbrev-expand-hook`. If the user termir an abbrev with a punctuation character, the function issues a prompt. Thus, this hook allows the user t decide whether the abbrev should be expanded, and to abort expansion if it is not desired.

```
(add-hook 'pre-abbrev-expand-hook 'query-if-not-space)

;; This is the function invoked by pre-abbrev-expand-hook.

;; If the user terminated the abbrev with a space, the function does
;; nothing (that is, it returns so that the abbrev can expand).  If the
;; user entered some other character, this function asks whether
;; expansion should continue.

;; If the user enters the prompt with y, the function returns
;; nil (because of the not function), but that is
;; acceptable; the return value has no effect on expansion.

(defun query-if-not-space ()
  (if (/= ?\  (preceding-char))
      (if (not (y-or-n-p "Do you want to expand this abbrev? "))
          (error "Not expanding this abbrev"))))
```

Standard Abbrev Tables

Here we list the variables that hold the abbrev tables for the preloaded major modes of Emacs.

Variable: **global-abbrev-table**

This is the abbrev table for mode-independent abbrevs. The abbrevs defined in it apply to all buffers. Each buffer may also have a local abbrev table, whose abbrev definitions take precedence over those in the global table.

Variable: **local-abbrev-table**

The value of this buffer-local variable is the (mode-specific) abbreviation table of the current buffer.

Variable: **fundamental-mode-abbrev-table**

This is the local abbrev table used in Fundamental mode. It is the local abbrev table in all buffers in Fundamental mode.

Variable: **text-mode-abbrev-table**

This is the local abbrev table used in Text mode.

Variable: **c-mode-abbrev-table**

This is the local abbrev table used in C mode.

Variable: **lisp-mode-abbrev-table**

This is the local abbrev table used in Lisp mode and Emacs Lisp mode.

Go to the previous, next section.

Go to the previous, next section.

Processes

In the terminology of operating systems, a process is a space in which a program can execute. Emacs runs in a process. Emacs Lisp programs can invoke other programs in processes of their own. These are called subprocesses or child processes of the Emacs process, which is their parent process.

A subprocess of Emacs may be synchronous or asynchronous, depending on how it is created. When you create a synchronous subprocess, the Lisp program waits for the subprocess to terminate before continuing execution. When you create an asynchronous subprocess, it can run in parallel with the Lisp program. This kind of subprocess is represented within Emacs by a Lisp object which is also called a "process". Lisp programs can use this object to communicate with the subprocess or to control it. For example, you can send signals, obtain status information, receive output from the process, or send input to it.

Function: **processp** *object*

This function returns `t` if object is a process, `nil` otherwise.

Functions that Create Subprocesses

There are three functions that create a new subprocess in which to run a program. One of them, `start-process`, creates an asynchronous process and returns a process object (see section Creating an Asynchronous Process). The other two, `call-process` and `call-process-region`, create a synchronous process and do not return a process object (see section Creating a Synchronous Process).

Synchronous and asynchronous processes are explained in following sections. Since the three functions are all called in a similar fashion, their common arguments are described here.

In all cases, the function's program argument specifies the program to be run. An error is signaled if the file is not found or cannot be executed. The actual file containing the program is found by following normal system rules: if the file name is absolute, then the program must be found in the specified file; if the name is relative, then the directories in `exec-path` are searched sequentially for a suitable file. The variable `exec-path` is initialized when Emacs is started, based on the value of the environment variable `PATH`. The standard file name constructs, `~', `.', and `..', are interpreted as usual in `exec-path`, but environment variable substitutions (`$HOME', etc.) are not recognized; use `substitute-in-file-name` to perform them (see section Functions that Expand Filenames).

Each of the subprocess-creating functions has a buffer-or-name argument which specifies where the standard output from the program will go. If buffer-or-name is `nil`, that says to discard the output unless a filter function handles it. (See section Process Filter Functions, and section Reading and Printing Lisp Objects.) Normally, you should avoid having multiple processes send output to the same buffer because their output would be intermixed randomly.

All three of the subprocess-creating functions have a `&rest` argument, args. The args must all be strings, and they are supplied to program as separate command line arguments. Wildcard characters and other shell constructs are not allowed in these strings, since they are passed directly to the specified program.

Please note: the argument program contains only the name of the program; it may not contain any command-line arguments. Such arguments must be provided via args.

The subprocess gets its current directory from the value of `default-directory` (see section Functions that Expand Filenames).

The subprocess inherits its environment from Emacs; but you can specify overrides for it with `process-environment`. See section Operating System Environment.

Variable: **exec-directory**

The value of this variable is the name of a directory (a string) that contains programs that come with GNU Emacs, that are intended for Emacs to invoke. The program `wakeup` is an example of such a program; the `display-time` command uses it to get a reminder once per minute.

The default value is the name of a directory whose name ends in `` `arch-lib' ``. We call the directory `` `emacs/arch-lib' ``, since its name usually ends that way. We sometimes refer to "the directory `` `emacs/arch-lib' ``," when strictly speaking we ought to say, "the directory named by the variable `exec-directory`." Most of the time, there is no difference.

(In earlier Emacs versions, prior to version 19, these files lived in the directory `` `emacs/etc' `` instead of in `` `emacs/arch-lib' ``.)

User Option: **exec-path**

The value of this variable is a list of directories to search for programs to run in subprocesses. Each element is either the name of a directory (i.e., a string), or `nil`, which stands for the default directory (which is the value of `default-directory`).

The value of `exec-path` is used by `call-process` and `start-process` when the program argument is not an absolute file name.

Creating a Synchronous Process

After a synchronous process is created, Emacs waits for the process to terminate before continuing. Starting Dired is an example of this: it runs `ls` in a synchronous process, then modifies the output slightly. Because the process is synchronous, the entire directory listing arrives in the buffer before Emacs tries to do anything with it.

While Emacs waits for the synchronous subprocess to terminate, the user can quit by typing C-g, and the process is killed by sending it a `SIGKILL` signal. See section Quitting.

The synchronous subprocess functions return `nil` in version 18. In version 19, they will return an indication of how the process terminated.

Function: **call-process** *program &optional infile buffer-or-name display &rest args*

This function calls program in a separate process and waits for it to finish.

The standard input for the process comes from file infile if infile is not `nil` and from `` `/dev/null' `` otherwise. The process output gets inserted in buffer buffer-or-name before point, if that argument names a buffer. If buffer-or-name is `t`, output is sent to the current buffer; if buffer-or-name is `nil`, output is discarded.

If buffer-or-name is the integer 0, `call-process` returns `nil` immediately and discards any output. In this case, the process is not truly synchronous, since it can run in parallel with Emacs; but you can think of it as synchronous in that Emacs is essentially finished with the subprocess as soon as this function returns.

If display is non-`nil`, then `call-process` redisplays the buffer as output is inserted. Otherwise the function does no redisplay, and the results become visible on the screen only when Emacs redisplays that buffer in the normal course of events.

The remaining arguments, args, are strings that are supplied as the command line arguments for the program.

The value returned by `call-process` (unless you told it not to wait) indicates the reason for process termination. A number gives the exit status of the subprocess; 0 means success, and any other value means failure. If the process terminated with a signal, `call-process` returns a string describing the signal.

The examples below are both run with the buffer `foo' current.

```
(call-process "pwd" nil t)
     => nil

---------- Buffer: foo ----------
/usr/user/lewis/manual
---------- Buffer: foo ----------

(call-process "grep" nil "bar" nil "lewis" "/etc/passwd")
     => nil

---------- Buffer: bar ----------
lewis:5LTsHm66CSWKg:398:21:Bil Lewis:/user/lewis:/bin/csh

---------- Buffer: bar ----------
```

The `dired-readin` function contains a good example of the use of `call-process`:

```
(call-process
 "ls" nil buffer nil dired-listing-switches dirname)
```

Function: **call-process-region** *start end program &optional delete buffer-or-name display &rest arg*

This function sends the text between start to end as standard input to a process running program. It deletes the text sent if delete is non-`nil`, which may be useful when the output is going to be inserted back in the current buffer.

If buffer-or-name names a buffer, the output is inserted in that buffer at point. If buffer-or-name is `t`, the output is sent to the current buffer. If buffer-or-name is `nil`, the output is discarded. If buffer-or-name is the integer 0, the output is discarded and `call-process` returns `nil` immediately, just as `call-process` would.

If display is non-`nil`, then `call-process-region` redisplays the buffer as output is inserted. Otherwise the function does no redisplay, and the results become visible on the screen only when Emacs redisplays that buffer in the normal course of events.

The remaining arguments, args, are strings that are supplied as the command line arguments for the program.

The return value of `call-process-region` is just like that of `call-process`: `nil` if you told it to return without waiting; otherwise, a number or string which indicates how the subprocess terminated.

In the following example, we use `call-process-region` to run the `cat` utility, with standard input being the first five characters in buffer `foo' (the word `input'). `cat` copies its standard input into its standard output. Since the argument buffer-or-name is `t`, this output is inserted in the current buffer.

```
---------- Buffer: foo ----------
input-!-
---------- Buffer: foo ----------

(call-process-region 1 6 "cat" nil t)
     => nil

---------- Buffer: foo ----------
inputinput-!-
---------- Buffer: foo ----------
```

The `shell-command-on-region` command uses `call-process-region` like this:

```
(call-process-region
 start end
 shell-file-name      ; Name of program.
 nil                  ; Do not delete region.
 buffer               ; Send output to buffer.
 nil                  ; No redisplay during output.
 "-c" command)        ; Arguments for the shell.
```

Creating an Asynchronous Process

After an asynchronous process is created, Emacs and the Lisp program can continue running immediately. The process may thereafter run in parallel with Emacs, and the two may communicate with each other using the functions described in following sections. Here we describe how to create an asynchronous process, with `start-process`.

Function: **start-process** *name buffer-or-name program &rest args*

This function creates a new asynchronous subprocess and starts the program program running in it. It returns a process object that stands for the new subprocess for Emacs Lisp programs. The argument name specifies the name for the process object; if a process with this name already exists, then name is modified (by adding `<1>', etc.) to be unique. The buffer buffer-or-name is the buffer to associate with the process.

The remaining arguments, args, are strings that are supplied as the command line arguments for the program.

In the example below, the first process is started and runs (rather, sleeps) for 100 seconds. Meanwhile, the second process is started, given the name `my-process<1>' for the sake of uniqueness. It inserts the directory listing at the end of the buffer `foo', before the first process finishes. Then it finishes, and a message to that effect is inserted in the buffer. Much later, the first process finishes, and another message is inserted in the buffer for it.

```
(start-process "my-process" "foo" "sleep" "100")
     => #<process my-process>

(start-process "my-process" "foo" "ls" "-l" "/user/lewis/bin")
     => #<process my-process<1>>

---------- Buffer: foo ----------
total 2
lrwxrwxrwx  1 lewis     14 Jul 22 10:12 gnuemacs --> /emacs
-rwxrwxrwx  1 lewis     19 Jul 30 21:02 lemon

Process my-process<1> finished

Process my-process finished
---------- Buffer: foo ----------
```

Function: **start-process-shell-command** *name buffer-or-name command &rest command-args*

This function is like `start-process` except that it uses a shell to execute the specified command. The argument command is a shell command name, and command-args are the arguments for the shell command.

Variable: **process-connection-type**

This variable controls the type of device used to communicate with asynchronous subprocesses. If it is `nil`, then pipes are used. If it is `t`, then PTYs are used (or pipes if PTYs are not supported).

PTYs are usually preferable for processes visible to the user, as in Shell mode, because they allow job control (C-c, C-z, etc.) to work between the process and its children whereas pipes do not. For subprocesses used for internal purposes by programs, it is often better to use a pipe, because they are more efficient. In addition, the total number of PTYs is limited on many systems and it is good not to waste them.

The value `process-connection-type` is used when `start-process` is called, so in order to change it for just one call of `start-process`, temporarily rebind it with `let`.

```
(let ((process-connection-type nil))  ; Use a pipe.
  (start-process ...))
```

Deleting Processes

Deleting a process disconnects Emacs immediately from the subprocess, and removes it from the list of active processes. It sends a signal to the subprocess to make the subprocess terminate, but this is not guaranteed to happen immediately. (The process object itself continues to exist as long as other Lisp objects point to it.)

You can delete a process explicitly at any time. Processes are deleted automatically after they terminate, but not necessarily right away. If you delete a terminated process explicitly before it is deleted automatically, no harm results.

Variable: **delete-exited-processes**

This variable controls automatic deletion of processes that have terminated (due to calling `exit` or to a signal). If it is `nil`, then they continue to exist until the user runs `list-processes`. Otherwise, they are deleted immediately after they exit.

Function: **delete-process** *name*

This function deletes the process associated with name. The argument name may be a process, the name of a process, a buffer, or the name of a buffer. The subprocess is killed with a `SIGHUP` signal.

```
(delete-process "*shell*")
     => nil
```

Function: **process-kill-without-query** *process*

This function declares that Emacs need not query the user if process is still running when Emacs is exited. The process will be deleted silently. The value is `t`.

```
(process-kill-without-query (get-process "shell"))
     => t
```

Process Information

Several functions return information about processes. `list-processes` is provided for interactive use.

Command: **list-processes**

This command displays a listing of all living processes. (Any processes listed as `Exited' or `Signaled' are actually eliminated after the listing is made.) This function returns `nil`.

Function: **process-list**

This function returns a list of all processes that have not been deleted.

```
(process-list)
     => (#<process display-time> #<process shell>)
```

Function: **get-process** *name*

This function returns the process named name, or `nil` if there is none. An error is signaled if name is not a string.

```
(get-process "shell")
     => #<process shell>
```

Function: **process-command** *process*

This function returns the command that was executed to start process. This is a list of strings, the first string being the program executed and the rest of the strings being the arguments that were given to the program.

```
(process-command (get-process "shell"))
     => ("/bin/csh" "-i")
```

Function: **process-exit-status** *process*

This function returns the exit status of process or the signal number that killed it. (Use the result of `process-status` to determine which of those it is.) If process has not yet terminated, the value is 0.

Function: **process-id** *process*

This function returns the PID of process. This is an integer which distinguishes the process process from all other processes running on the same computer at the current time. The PID of a process is chosen by the operating system kernel when the process is started and remains constant as long as the process exists.

Function: **process-name** *process*

This function returns the name of process.

Function: **process-status** *process-name*

This function returns the status of process-name as a symbol. The argument process-name must be either a process or a string. If it is a string, it need not name an actual process.

The possible values for an actual subprocess are:

`run`
: for a process that is running.

`stop`
: for a process that is stopped but continuable.

`exit`
: for a process that has exited.

`signal`
: for a process that has received a fatal signal.

`open`
: for a network connection that is open.

`closed`
: for a network connection that is closed. Once a connection is closed, you cannot reopen it, though you might be able to open a new connection to the same place.

`nil`
: if process-name is not the name of an existing process.

```
(process-status "shell")
     => run
(process-status "never-existed")
     => nil
x
     => #<process xx<1>>
(process-status x)
     => exit
```

For a network connection, `process-status` returns one of the symbols `open` or `closed`. The latter means that the other side closed the connection, or Emacs did `delete-process`.

In earlier Emacs versions (prior to version 19), the status of a network connection was `run` if open, and `exit` if closed.

Sending Input to Processes

Asynchronous subprocesses receive input when it is sent to them by Emacs, which is done with the functions in this section. You must specify the process to send input to, and the input data to send. The data appears on the "standard input" of the subprocess.

Some operating systems have limited space for buffered input in a PTY. On these systems, the subprocess will cease to read input correctly if you send an input line longer than the system can handle. You cannot avoid the problem by breaking the input into pieces and sending them separately, for the operating system will still have to put all the pieces together in the input buffer before it lets the subprocess read the line. The only solution is to put the input in a temporary file, and send the process a brief command to read that file.

Function: **process-send-string** *process-name string*

This function sends process-name the contents of string as standard input. The argument process-name must be a process or the name of a process.

The function returns `nil`.

```
(process-send-string "shell<1>" "ls\n")
     => nil


---------- Buffer: *shell* ----------
...
introduction.texi              syntax-tables.texi~
introduction.texi~             text.texi
introduction.txt               text.texi~
...
---------- Buffer: *shell* ----------
```

Command: **process-send-region** *process-name start end*

This function sends the text in the region defined by start and end as standard input to process-name, which is a process or a process name.

An error is signaled unless both start and end are integers or markers that indicate positions in the current buffer. (It is unimportant which number is larger.)

Function: **process-send-eof** *&optional process-name*

This function makes process-name see an end-of-file in its input. The EOF comes after any text already sent to it.

If process-name is not supplied, or if it is `nil`, then this function sends the EOF to the current buffer's process. An error is signaled if the current buffer has no process.

The function returns process-name.

```
(process-send-eof "shell")
     => "shell"
```

Sending Signals to Processes

Sending a signal to a subprocess is a way of interrupting its activities. There are several different signals, each with its own meaning. For example, the signal `SIGINT` means that the user has typed C-c, or that some analogous thing has happened.

Each signal has a standard effect on the subprocess. Most signals kill the subprocess, but some stop or resume execution instead. Most signals can optionally be handled by programs; if the program handles the signal, then we can say nothing in general about its effects.

The set of signals and their names is defined by the operating system; Emacs has facilities for sending only a few of the signals that are defined. Emacs can send signals only to its own subprocesses.

You can send signals explicitly by calling the functions in this section. Emacs also sends signals automatically at certain times: killing a buffer sends a `SIGHUP` signal to all its associated processes; killing Emacs sends a `SIGHUP` signal to all remaining processes. (`SIGHUP` is a signal that usually indicates that the user hung up the phone.)

Each of the signal-sending functions takes two optional arguments: process-name and current-group.

The argument process-name must be either a process, the name of one, or `nil`. If it is `nil`, the process defaults to the process associated with the current buffer. An error is signaled if process-name does not identify a process.

The argument current-group is a flag that makes a difference when you are running a job-control shell as an Emacs subprocess. If it is non-`nil`, then the signal is sent to the current process-group of the terminal which Emacs uses to communicate with the subprocess. If the process is a job-control shell, this means the shell's current subjob. If it is `nil`, the signal is sent to the process group of the immediate subprocess of Emacs. If the subprocess is a job-control shell, this is the shell itself.

The flag current-group has no effect when a pipe is used to communicate with the subprocess, because the operating system does not support the distinction in the case of pipes. For the same reason, job-control shells won't work when a pipe is used. See `process-connection-type` in section Creating an Asynchronous Process.

Function: **interrupt-process** *&optional process-name current-group*

This function interrupts the process process-name by sending the signal `SIGINT`. Outside of Emacs, typing the "interrupt character" (normally C-c on some systems, and `DEL` on others) sends this signal. When the argument current-group is non-`nil`, you can think of this function as "typing C-c" on the terminal by which Emacs talks to the subprocess.

Function: **kill-process** *&optional process-name current-group*

This function kills the process process-name by sending the signal `SIGKILL`. This signal kills the subprocess immediately, and cannot be handled by the subprocess.

Function: **quit-process** *&optional process-name current-group*

This function sends the signal `SIGQUIT` to the process process-name. This signal is the one sent by the "quit character" (usually C-b or C-\) when you are not inside Emacs.

Function: **stop-process** *&optional process-name current-group*

This function stops the process process-name by sending the signal `SIGTSTP`. Use `continue-process` to resume its execution.

On systems with job control, the "stop character" (usually C-z) sends this signal (outside of Emacs). When current-group is non-`nil`, you can think of this function as "typing C-z" on the terminal Emacs uses to communicate with the subprocess.

Function: **continue-process** *&optional process-name current-group*

This function resumes execution of the process process by sending it the signal `SIGCONT`. This presumes that process-name was stopped previously.

Function: **signal-process** *pid signal*

This function sends a signal to process pid, which need not be a child of Emacs. The argument signal specifies which signal to send; it should be an integer.

Receiving Output from Processes

There are two ways to receive the output that a subprocess writes to its standard output stream. The output can be inserted in a buffer, which is called the associated buffer of the process, or a function called the filter function can be called to act on the output.

Process Buffers

A process can (and usually does) have an associated buffer, which is an ordinary Emacs buffer that is used for two purposes: storing the output from the process, and deciding when to kill the process. You can also use the buffer to identify a process to operate on, since in normal practice only one process is associated with any given buffer. Many applications of processes also use the buffer for editing input to be sent to the process, but this is not built into Emacs Lisp.

Unless the process has a filter function (see section Process Filter Functions), its output is inserted in the associated buffer. The position to insert the output is determined by the `process-mark` (see section Process Information), which is then updated to point to the end of the text just inserted. Usually, but not always, the `process-mark` is at the end of the buffer. If the process has no buffer and no filter function, its output is discarded.

Function: **process-buffer** *process*

This function returns the associated buffer of the process process.

```
(process-buffer (get-process "shell"))
     => #<buffer *shell*>
```

Function: **process-mark** *process*

This function returns the marker which controls where additional output from the process will be inserted in the process buffer. When output is inserted, the marker is updated to point at the end of the output. This causes successive batches of output to be inserted consecutively.

If process does not insert its output into a buffer, then `process-mark` returns a marker that points nowhere.

Filter functions normally should use this marker in the same fashion as is done by direct insertion of output in the buffer. A good example of a filter function that uses `process-mark` is found at the end of the following section.

When the user is expected to enter input in the process buffer for transmission to the process, the process marker is useful for distinguishing the new input from previous output.

Function: **set-process-buffer** *process buffer*

This function sets the buffer associated with process to buffer. If buffer is `nil`, the process will not be associated with any buffer.

Function: **get-buffer-process** *buffer-or-name*

This function returns the process associated with buffer-or-name. If there are several processes associated with it, then one is chosen. (Presently, the one chosen is the one most recently created.) It is usually a bad idea to have more than one process associated with the same buffer.

```
(get-buffer-process "*shell*")
     => #<process shell>
```

If the process's buffer is killed, the actual child process is killed with a `SIGHUP` signal (see section Sending Signals to Processes).

Process Filter Functions

A process filter function is a function that receives the standard output from the associated process. If a process has a filter, then *all* output from that process, that would otherwise have been in a buffer, is passed to the filter. The process buffer is used for output from the process only when there is no filter.

A filter function must accept two arguments: the associated process and a string, which is the output. The function is then free to do whatever it chooses with the output.

A filter function runs only while Emacs is waiting (e.g., for terminal input, or for time to elapse, or for process output). This avoids the timing errors that could result from running filters at random places in the middle of other Lisp programs. You may explicitly cause Emacs to wait, so that filter functions will run, by calling `sit-for`, `sleep-for` or `accept-process-output` (see section Accepting Output from Processes). Emacs is also waiting when the command loop is reading input.

Quitting is normally inhibited within a filter function--otherwise, the effect of typing C-g at command

level or to quit a user command would be unpredictable. If you want to permit quitting inside a filter function, bind `inhibit-quit` to `nil`. See section Quitting.

Many filter functions sometimes or always insert the text in the process's buffer, mimicking the actions of Emacs when there is no filter. Such filter functions need to use `set-buffer` in order to be sure to insert in that buffer. To avoid setting the current buffer semipermanently, these filter functions must use `unwind-protect` to make sure to restore the previous current buffer. They should also update the process marker, and in some cases update the value of point. Here is how to do these things:

```
(defun ordinary-insertion-filter (proc string)
  (let ((old-buffer (current-buffer)))
    (unwind-protect
        (let (moving)
          (set-buffer (process-buffer proc))
          (setq moving (= (point) (process-mark proc)))
          (save-excursion
            ;; Insert the text, moving the process-marker.
            (goto-char (process-mark proc))
            (insert string)
            (set-marker (process-mark proc) (point)))
          (if moving (goto-char (process-mark proc))))
      (set-buffer old-buffer))))
```

The reason to use an explicit `unwind-protect` rather than letting `save-excursion` restore the current buffer is so as to preserve the change in point made by `goto-char`.

To make the filter force the process buffer to be visible whenever new text arrives, insert the following line just before the `unwind-protect`:

```
(display-buffer (process-buffer proc))
```

To force point to move to the end of the new output no matter where it was previously, eliminate the variable `moving` and call `goto-char` unconditionally.

All filter functions that do regexp searching or matching should save and restore the match data. Otherwise, a filter function that runs during a call to `sit-for` might clobber the match data of the program that called `sit-for`. See section The Match Data.

The output to the function may come in chunks of any size. A program that produces the same output twice in a row may send it as one batch of 200 characters one time, and five batches of 40 characters the next.

Function: **set-process-filter** *process filter*

This function gives process the filter function filter. If filter is `nil`, then the process will have no filter.

Function: **process-filter** *process*

This function returns the filter function of process, or `nil` if it has none.

Here is an example of use of a filter function:

```
(defun keep-output (process output)
   (setq kept (cons output kept)))
     => keep-output
(setq kept nil)
     => nil
(set-process-filter (get-process "shell") 'keep-output)
     => keep-output
(process-send-string "shell" "ls ~/other\n")
     => nil
kept
     => ("lewis@slug[8] % "
"FINAL-W87-SHORT.MSS    backup.otl              kolstad.mss~
address.txt             backup.psf              kolstad.psf
backup.bib~             david.mss               resume-Dec-86.mss~
backup.err              david.psf               resume-Dec.psf
backup.mss              dland                   syllabus.mss
"
"#backups.mss#          backup.mss~             kolstad.mss
")
```

Here is another, more realistic example, which demonstrates how to use the process mark to do insertion in the same fashion as is done when there is no filter function:

```
;; Insert input in the buffer specified by my-shell-buffer
;;   and make sure that buffer is shown in some window.
(defun my-process-filter (proc str)
    (let ((cur (selected-window))
          (pop-up-windows t))
      (pop-to-buffer my-shell-buffer)
      (goto-char (point-max))
      (insert str)
      (set-marker (process-mark proc) (point-max))
      (select-window cur)))
```

Accepting Output from Processes

Output from asynchronous subprocesses normally arrives only while Emacs is waiting for some sort of external event, such as elapsed time or terminal input. Occasionally it is useful in a Lisp program to explicitly permit output to arrive at a specific point, or even to wait until output arrives from a process.

Function: **accept-process-output** *&optional process seconds millisec*

This function allows Emacs to read pending output from processes. The output is inserted in the associated buffers or given to their filter functions. If process is non-`nil` then this function does not return until some output has been received from process.

The arguments seconds and millisec let you specify timeout periods. The former specifies a period measured in seconds and the latter specifies one measured in milliseconds. The two time periods thus specified are added together, and `accept-process-output` returns after that much time whether or not there has been any subprocess output.

Not all operating systems support waiting periods other than multiples of a second; on those that do not, you get an error if you specify nonzero millisec.

The function `accept-process-output` returns non-`nil` if it did get some output, or `nil` if the timeout expired before output arrived.

Sentinels: Detecting Process Status Changes

A process sentinel is a function that is called whenever the associated process changes status for any reason, including signals (whether sent by Emacs or caused by the process's own actions) that terminate, stop, or continue the process. The process sentinel is also called if the process exits. The sentinel receives two arguments: the process for which the event occurred, and a string describing the type of event.

The string describing the event looks like one of the following:

- `"finished\n"`.
- `"exited abnormally with code exitcode\n"`.
- `"name-of-signal\n"`.
- `"name-of-signal (core dumped)\n"`.

A sentinel runs only while Emacs is waiting (e.g., for terminal input, or for time to elapse, or for process output). This avoids the timing errors that could result from running them at random places in the middle of other Lisp programs. You may explicitly cause Emacs to wait, so that sentinels will run, by calling `sit-for`, `sleep-for` or `accept-process-output` (see section Accepting Output from Processes). Emacs is also waiting when the command loop is reading input.

Quitting is normally inhibited within a sentinel--otherwise, the effect of typing C-g at command level or to quit a user command would be unpredictable. If you want to permit quitting inside a sentinel, bind `inhibit-quit` to `nil`. See section Quitting.

All sentinels that do regexp searching or matching should save and restore the match data. Otherwise, a sentinel that runs during a call to `sit-for` might clobber the match data of the program that called `sit-for`. See section The Match Data.

Function: **set-process-sentinel** *process sentinel*

This function associates sentinel with process. If sentinel is `nil`, then the process will have no sentinel. The default behavior when there is no sentinel is to insert a message in the process's buffer when the process status changes.


```
(defun msg-me (process event)
   (princ
     (format "Process: %s had the event `%s'" process event)))
(set-process-sentinel (get-process "shell") 'msg-me)
     => msg-me
(kill-process (get-process "shell"))
     -| Process: #<process shell> had the event `killed'
     => #<process shell>
```

Function: **process-sentinel** *process*

This function returns the sentinel of process, or `nil` if it has none.

Function: **waiting-for-user-input-p**

While a sentinel or filter function is running, this function returns non-`nil` if Emacs was waiting for keyboard input from the user at the time the sentinel or filter function was called, `nil` if it was not.

Transaction Queues

You can use a transaction queue for more convenient communication with subprocesses using transactions. First use `tq-create` to create a transaction queue communicating with a specified process. Then you can call `tq-enqueue` to send a transaction.

Function: **tq-create** *process*

This function creates and returns a transaction queue communicating with process. The argument process should be a subprocess capable of sending and receiving streams of bytes. It may be a child process, or it may be a TCP connection to a server possibly on another machine.

Function: **tq-enqueue** *queue question regexp closure fn*

This function sends a transaction to queue queue. Specifying the queue has the effect of specifying the subprocess to talk to.

The argument question is the outgoing message which starts the transaction. The argument fn is the function to call when the corresponding answer comes back; it is called with two arguments: closure, and the answer received.

The argument regexp is a regular expression to match the entire answer; that's how `tq-enqueue` tells where the answer ends.

The return value of `tq-enqueue` itself is not meaningful.

Function: **tq-close** *queue*

Shut down transaction queue queue, waiting for all pending transactions to complete, and then terminate the connection or child process.

Transaction queues are implemented by means of a filter function. See section Process Filter Functio

TCP

Emacs Lisp programs can open TCP connections to other processes on the same machine or other machines. A network connection is handled by Lisp much like a subprocess, and is represented by a process object. However, the process you are communicating with is not a child of the Emacs process, so you can't kill it or send it signals. All you can do is send and receive data. `delete-process` closes the connection, but does not kill the process at the other end; that process must decide what to do about closure of the connection.

You can distinguish process objects representing network connections from those representing subprocesses with the `process-status` function. See section Process Information.

Function: **open-network-stream** *name buffer-or-name host service*

This function opens a TCP connection for a service to a host. It returns a process object to represent the connection.

The name argument specifies the name for the process object. It is modified as necessary to make it unique.

The buffer-or-name argument is the buffer to associate with the connection. Output from the connection is inserted in the buffer, unless you specify a filter function to handle the output. If buffer-or-name is `nil`, it means that the connection is not associated with any buffer.

The arguments host and service specify where to connect to; host is the host name (a string), and service is the name of a defined network service (a string) or a port number (an integer).

Go to the previous, next section.

Go to the previous, next section.

Operating System Interface

This chapter is about starting and getting out of Emacs, access to values in the operating system environment, and terminal input, output and flow control.

See section Building Emacs, for related information. See also section Emacs Display, for additional operating system status information pertaining to the terminal and the screen.

Starting Up Emacs

This section describes what Emacs does when it is started, and how you can customize these actions.

Summary: Sequence of Actions at Start Up

The order of operations performed (in `startup.el') by Emacs when it is started up is as follows:

1. It runs the normal hook `before-init-hook`.
2. It loads `.emacs' unless `-q' was specified on command line. (This is not done in `-batch' mode.) `.emacs' is found in the user's home directory; the `-u' option can specify the user name whose home directory should be used.
3. It loads `default.el' unless `inhibit-default-init` is non-`nil`. (This is not done in `-batch' mode or if `-q' was specified on command line.)
4. It runs the normal hook `after-init-hook`.
5. It loads the terminal-specific Lisp file, if any, except when in batch mode.
6. It runs `term-setup-hook`.
7. It runs `window-setup-hook`. See section Window Systems.
8. It displays copyleft and nonwarranty, plus basic use information, unless the value of the variable `inhibit-startup-message` is non-`nil`.

   This display is also inhibited in batch mode, and if the current buffer is not `*scratch*'.
9. It processes any remaining command line arguments.

User Option: **inhibit-startup-message**

This variable inhibits the initial startup messages (the nonwarranty, etc.). If it is non-`nil`, then the messages are not printed.

This variable exists so you can set it in your personal init file, once you are familiar with the contents of the startup message. Do not set this variable in the init file of a new user, or in a way that affects more than one user, because that would prevent new users from receiving the information they are supposed to see.

The Init File: `.emacs'

When you start Emacs, it normally attempts to load the file `.emacs' from your home directory. This file, if it exists, must contain Lisp code. It is called your init file. The command line switches `-q' and `-u' can be used to control the use of the init file; `-q' says not to load an init file, and `-u' says to load a specified user's init file instead of yours. See section 'Entering Emacs' in The GNU Emacs Manual.

Emacs may also have a default init file, which is the library named `default.el'. Emacs finds the `default.el' file through the standard search path for libraries (see section How Programs Do Loading). The Emacs distribution does not have any such file; you may create one at your site for local customizations. If the default init file exists, it is loaded whenever you start Emacs, except in batch mode or if `-q' is specified. But your own personal init file, if any, is loaded first; if it sets `inhibit-default-init` to a non-`nil` value, then Emacs will not subsequently load the `default.el' file.

If there is a great deal of code in your `.emacs' file, you should move it into another file named `something.el', byte-compile it (see section Byte Compilation), and make your `.emacs' file load the other file using `load` (see section Loading).

See section 'Init File Examples' in The GNU Emacs Manual, for examples of how to make various commonly desired customizations in your `.emacs' file.

User Option: **inhibit-default-init**

This variable prevents Emacs from loading the default initialization library file for your session of Emacs. If its value is non-`nil`, then the default library is not loaded. The default value is `nil`.

Variable: **before-init-hook**

Variable: **after-init-hook**

These two normal hooks are run just before, and just after, loading of the user's init file or `default.el'.

Terminal-Specific Initialization

Each terminal type can have its own Lisp library that Emacs loads when run on that type of terminal. For a terminal type named termtype, the library is called `term/termtype'. Emacs finds the file by searching the `load-path` directories as it does for other files, and trying the `.elc' and `.el' suffixes. Normally, terminal-specific Lisp library is located in `emacs/lisp/term', a subdirectory of the `emacs/lisp' directory in which most Emacs Lisp libraries are kept.

The library's name is constructed by concatenating the value of the variable `term-file-prefix` and the terminal type. Normally, `term-file-prefix` has the value `"term/"`; changing this is not recommended.

The usual function of a terminal-specific library is to enable special keys to send sequences that Emacs can recognize. It may also need to set or add to `function-key-map` if the Termcap entry does not

fully explain what should go in it. See section Terminal Input.

When the name of the terminal type contains a hyphen, only the part of the name before the first hyphen is significant in choosing the library name. Thus, terminal types `aaa-48' and `aaa-30-rv' both use the `term/aaa' library. If necessary, the library can evaluate `(getenv "TERM")` to find the full name of the terminal type.

Your `.emacs' file can prevent the loading of the terminal-specific library by setting the variable `term-file-prefix` to `nil`. This feature is very useful when experimenting with your own peculiar customizations.

You can also arrange to override some of the actions of the terminal-specific library by setting the variable `term-setup-hook`. This is a normal hook which Emacs runs using `run-hooks` at the end of Emacs initialization, after loading both your `.emacs' file and any terminal-specific libraries. You can use this variable to define initializations for terminals that do not have their own libraries. See section Hooks.

Variable: **term-file-prefix**

If the `term-file-prefix` variable is non-`nil`, Emacs loads a terminal-specific initialization file as follows:

```
(load (concat term-file-prefix (getenv "TERM")))
```

You may set the `term-file-prefix` variable to `nil` in your `.emacs' file if you do not wish to load the terminal-initialization file. To do this, put the following in your `.emacs' file: `(setq term-file-prefix nil)`.

Variable: **term-setup-hook**

This variable is a normal hook which Emacs runs after loading your `.emacs' file, the default initialization file (if any) and after loading terminal-specific Lisp files. arguments.

You can use `term-setup-hook` to override the definitions made by a terminal-specific file.

See `window-setup-hook` in section Window Systems, for a related feature.

Command Line Arguments

You can use command line arguments to request various actions when you start Emacs. Since you do not need to start Emacs more than once per day, and will often leave your Emacs session running longer than that, command line arguments are hardly ever used. As a practical matter, it is best to avoid making the habit of using them, since this habit would encourage you to kill and restart Emacs unnecessarily often. These options exist for two reasons: to be compatible with other editors (for invocation by other programs) and to enable shell scripts to run specific Lisp programs.

Function: **command-line**

This function parses the command line which Emacs was called with, processes it, loads the user's

`.emacs' file and displays the initial nonwarranty information, etc.

Variable: **command-line-processed**

The value of this variable is `t` once the command line has been processed.

If you redump Emacs by calling `dump-emacs`, you may wish to set this variable to `nil` first in order to cause the new dumped Emacs to process its new command line arguments.

Variable: **command-switch-alist**

The value of this variable is an alist of user-defined command-line options and associated handler functions. This variable exists so you can add elements to it.

A command line option is an argument on the command line of the form:

```
-option
```

The elements of the `command-switch-alist` look like this:

```
(option . handler-function)
```

The handler-function is called to handle option and receives the option name as its sole argument.

In some cases, the option is followed in the command line by an argument. In these cases, the handler-function can find all the remaining command-line arguments in the variable `command-line-args-left`. (The entire list of command-line arguments is in `command-line-args`.)

The command line arguments are parsed by the `command-line-1` function in the `startup.el' file. See also section 'Command Line Switches and Arguments' in The GNU Emacs Manual.

Variable: **command-line-args**

The value of this variable is the arguments passed by the shell to Emacs, as a list of strings.

Getting out of Emacs

There are two ways to get out of Emacs: you can kill the Emacs job, which exits permanently, or you can suspend it, which permits you to reenter the Emacs process later. As a practical matter, you seldom kill Emacs--only when you are about to log out. Suspending is much more common.

Killing Emacs

Killing Emacs means ending the execution of the Emacs process. The parent process normally resumes control.

All the information in the Emacs process, aside from files that have been saved, is lost when the Emacs is killed. Because killing Emacs inadvertently can lose a lot of work, Emacs queries for confirmation before

actually terminating if you have buffers that need saving or subprocesses that are running.

Function: **kill-emacs** *&optional no-query*

This function exits the Emacs process and kills it.

Normally, if there are modified files or if there are running processes, `kill-emacs` asks the user for confirmation before exiting. However, if no-query is supplied and non-`nil`, then Emacs exits without confirmation.

If no-query is an integer, then it is used as the exit status of the Emacs process. (This is useful primarily in batch operation; see section Batch Mode.)

If no-query is a string, its contents are stuffed into the terminal input buffer so that the shell (or whatever program next reads input) can read them.

Variable: **kill-emacs-hook**

This variable is a normal hook (a list of functions); the first thing `kill-emacs` does is to run this hook with `run-hooks`. That calls each of the functions in the list, with no arguments.

Suspending Emacs

Suspending Emacs means stopping Emacs temporarily and returning control to its superior process, which is usually the shell. This allows you to resume editing later in the same Emacs process, with the same buffers, the same kill ring, the same undo history, and so on. To resume Emacs, use the appropriate command in the parent shell--most likely `fg`.

Some operating systems do not support suspension of jobs; on these systems, "suspension" actually creates a new shell temporarily as a subprocess of Emacs. Then you would exit the shell to return to Emacs.

Suspension is not useful with window systems such as X, because the Emacs job may not have a parent that can resume it again, and in any case you can give input to some other job such as a shell merely by moving to a different window. Therefore, suspending is not allowed when Emacs is an X client.

Function: **suspend-emacs** *string*

This function stops Emacs and returns to the superior process. It returns `nil`.

If string is non-`nil`, its characters are sent to be read as terminal input by Emacs's superior shell. The characters in string are not echoed by the superior shell; only the results appear.

Before suspending, `suspend-emacs` runs the normal hook `suspend-hook`. In Emacs version 18, `suspend-hook` was not a normal hook; its value was a single function, and if its value was non-`nil`, then `suspend-emacs` returned immediately without actually suspending anything.

After the user resumes Emacs, it runs the normal hook `suspend-resume-hook` using `run-hooks`. See section Hooks.

The next redisplay after resumption will redraw the entire screen, unless the variable

`no-redraw-on-reenter` is non-`nil` (see section Refreshing the Screen).

In the following example, note that `pwd' is not echoed after Emacs is suspended. But it is read and executed by the shell.

```
(suspend-emacs)
     => nil

(add-hook 'suspend-hook
          (function (lambda ()
                      (or (y-or-n-p
                            "Really suspend? ")
                          (error "Suspend cancelled")))))
     => (lambda nil
          (or (y-or-n-p "Really suspend? ")
              (error "Suspend cancelled")))
(add-hook 'suspend-resume-hook
          (function (lambda () (message "Resumed!"))))
     => (lambda nil (message "Resumed!"))
(suspend-emacs "pwd")
     => nil
---------- Buffer: Minibuffer ----------
Really suspend? y
---------- Buffer: Minibuffer ----------

---------- Parent Shell ----------
lewis@slug[23] % /user/lewis/manual
lewis@slug[24] % fg

---------- Echo Area ----------
Resumed!
```

Variable: **suspend-hook**

This variable is a normal hook run before suspending.

Variable: **suspend-resume-hook**

This variable is a normal hook run after suspending.

Operating System Environment

Emacs provides access to variables in the operating system environment through various functions. These variables include the name of the system, the user's UID, and so on.

Variable: **system-type**

The value of this variable is a symbol indicating the type of operating system Emacs is operating on. Here is a table of the symbols for the operating systems that Emacs can run on up to version 19.1.

`aix-v3`
: AIX version 3.

`berkeley-unix`
: Berkeley BSD 4.1, 4.2, or 4.3.

`hpux`
: Hewlett-Packard operating system, version 5, 6, or 7.

`irix`
: Silicon Graphics Irix system.

`rtu`
: RTU 3.0, UCB universe.

`unisoft-unix`
: UniSoft's UniPlus 5.0 or 5.2.

`usg-unix-v`
: AT&T's System V.0, System V Release 2.0, 2.2, or 3.

`vax-vms`
: VAX VMS version 4 or 5.

`xenix`
: SCO Xenix 386 Release 2.2.

We do not wish to add new symbols to make finer distinctions unless it is absolutely necessary! In fact, it would be nice to eliminate some of these alternatives in the future.

Function: **system-name**

This function returns the name of the machine you are running on.

```
(system-name)
     => "prep.ai.mit.edu"
```

Function: **getenv** *var*

This function returns the value of the environment variable var, as a string. If the variable `process-environment` specifies a value for var, that overrides the actual environment.

```
(getenv "USER")
     => "lewis"

lewis@slug[10] % printenv
PATH=.:/user/lewis/bin:/usr/bin:/usr/local/bin
USER=lewis
```


```
TERM=ibmapa16
SHELL=/bin/csh
HOME=/user/lewis
```

Command: **setenv** *variable value*

This command sets the value of the environment variable named variable to value. Both arguments should be strings. This works by modifying `process-environment`; binding that variable with `let` is also reasonable practice.

Variable: **process-environment**

This variable is a list of strings to append to the environment of processes as they are created. Each string assigns a value to a shell environment variable. (This applies both to asynchronous and synchronous processes.) The function `getenv` also looks at this variable.

```
process-environment
=> ("l=/usr/stanford/lib/gnuemacs/lisp"
    "PATH=.:/user/lewis/bin:/usr/class:/nfsusr/local/bin"
    "USER=lewis"
    "TERM=ibmapa16"
    "SHELL=/bin/csh"
    "HOME=/user/lewis")
```

Function: **load-average**

This function returns the current 1 minute, 5 minute and 15 minute load averages in a list. The values are integers that are 100 times the system load averages. (The load averages indicate the number of processes trying to run.)

```
(load-average)
     => (169 48 36)

lewis@rocky[5] % uptime
 11:55am  up 1 day, 19:37,  3 users,
 load average: 1.69, 0.48, 0.36
```

Function: **setprv** *privilege-name &optional setp getprv*

This function sets or resets a VMS privilege. (It does not exist on Unix.) The first arg is the privilege name, as a string. The second argument, setp, is `t` or `nil`, indicating whether the privilege is to be turned on or off. Its default is `nil`. The function returns `t` if success, `nil` if not.

If the third argument, getprv, is non-`nil`, `setprv` does not change the privilege, but returns `t` or `nil` indicating whether the privilege is currently enabled.

User Identification

Function: **user-login-name**

This function returns the name under which the user is logged in. This is based on the effective UID, not the real UID.

```
(user-login-name)
     => "lewis"
```

Function: **user-real-login-name**

This function returns the name under which the user logged in. This is based on the real UID, not the effective UID. This differs from `user-login-name` only when running with the setuid bit.

Function: **user-full-name**

This function returns the full name of the user.

```
(user-full-name)
     => "Bil Lewis"
```

Function: **user-real-uid**

This function returns the real UID of the user.

```
(user-real-uid)
     => 19
```

Function: **user-uid**

This function returns the effective UID of the user.

Time of Day

This section explains how to determine the current time and the time zone.

Function: **current-time-string** *&optional time-value*

This function returns the current time and date as a humanly-readable string. The format of the string is unvarying; the number of characters used for each part is always the same, so you can reliably use `substring` to extract pieces of it. However, it would be wise to count the characters from the beginning of the string rather than from the end, as additional information may be added at the end.

The argument time-value, if given, specifies a time to format instead of the current time. The argument should be a cons cell containing two integers, or a list whose first two elements are integers. Thus, you can use times obtained from `current-time` (see below) and from `file-attributes` (see section Other Information about Files).


```
(current-time-string)
     => "Wed Oct 14 22:21:05 1987"
```

Function: **current-time**

This function returns the system's time value as a list of three integers: `(high low microsec)`. The integers high and low combine to give the number of seconds since 0:00 January 1, 1970, which is high * 2**16 + low.

The third element, microsec, gives the microseconds since the start of the current second (or 0 for systems that return time only on the resolution of a second).

The first two elements can be compared with file time values such as you get with the function `file-attributes`. See section Other Information about Files.

Function: **current-time-zone** *&optional time-value*

This function returns a list describing the time zone that the user is in.

The value has the form `(offset name)`. Here offset is an integer giving the number of seconds ahead of UTC (east of Greenwich). A negative value means west of Greenwich. The second element, name is a string giving the name of the time zone. Both elements change when daylight savings time begins or ends; if the user has specified a time zone that does not use a seasonal time adjustment, then the value is constant through time.

If the operating system doesn't supply all the information necessary to compute the value, both elements of the list are `nil`.

The argument time-value, if given, specifies a time to analyze instead of the current time. The argument should be a cons cell containing two integers, or a list whose first two elements are integers. Thus, you can use times obtained from `current-time` (see below) and from `file-attributes` (see section Other Information about Files).

Timers

You can set up a timer to call a function at a specified future time.

Function: **run-at-time** *time repeat function &rest args*

This function arranges to call function with arguments args at time time. The argument function is a function to call later, and args are the arguments to give it when it is called. The time time is specified as a string.

Absolute times may be specified in a wide variety of formats; The form `hour:min:sec timezone month/day/year', where all fields are numbers, works; the format that `current-time-string` returns is also allowed.

To specify a relative time, use numbers followed by units. For example:

`1 min'
: denotes 1 minute from now.

`1 min 5 sec'
: denotes 65 seconds from now.

`1 min 2 sec 3 hour 4 day 5 week 6 fortnight 7 month 8 year'
: denotes exactly 103 months, 123 days, and 10862 seconds from now.

If time is an integer, that specifies a relative time measured in seconds.

The argument repeat specifies how often to repeat the call. If repeat is `nil`, there are no repetitions; function is called just once, at time. If repeat is an integer, it specifies a repetition period measured in seconds.

Function: **cancel-timer** *timer*

Cancel the requested action for timer, which should be a value previously returned by `run-at-time`. This cancels the effect of that call to `run-at-time`; the arrival of the specified time will not cause anything special to happen.

Terminal Input

This section describes functions and variables for recording or manipulating terminal input. See section Emacs Display, for related functions.

Input Modes

Function: **set-input-mode** *interrupt flow meta quit-char*

This function sets the mode for reading keyboard input. If interrupt is non-null, then Emacs uses input interrupts. If it is `nil`, then it uses CBREAK mode.

If flow is non-`nil`, then Emacs uses XON/XOFF (C-q, C-s) flow control for output to terminal. This has no effect except in CBREAK mode. See section Flow Control.

The normal setting is system dependent. Some systems always use CBREAK mode regardless of what is specified.

The argument meta controls support for input character codes above 127. If meta is `t`, Emacs converts characters with the 8th bit set into Meta characters. If meta is `nil`, Emacs disregards the 8th bit; this is necessary when the terminal uses it as a parity bit. If meta is neither `t` nor `nil`, Emacs uses all 8 bits of input unchanged. This is good for terminals using European 8-bit character sets.

If quit-char is non-`nil`, it specifies the character to use for quitting. Normally this character is C-g. See section Quitting.

The `current-input-mode` function returns the input mode settings Emacs is currently using.

Function: **current-input-mode**

This function returns current mode for reading keyboard input. It returns a list, corresponding to the arguments of `set-input-mode`, of the form `(INTERRUPT FLOW META QUIT)` in which:

INTERRUPT
: is non-`nil` when Emacs is using interrupt-driven input. If `nil`, Emacs is using CBREAK mode.

FLOW
: is non-`nil` if Emacs uses XON/XOFF (C-q, C-s) flow control for output to the terminal. This value has no effect unless INTERRUPT is non-`nil`.

META
: is non-`nil` if Emacs is paying attention to the eighth bit of input characters; if nil, Emacs clears the eighth bit of every input character.

QUIT
: is the character Emacs currently uses for quitting, usually C-g.

Variable: **meta-flag**

This variable used to control whether to treat the 0200 bit in keyboard input as the Meta bit. `nil` meant no, and anything else meant yes. This variable existed in Emacs versions 18 and earlier but no longer exists in Emacs 19; use `set-input-mode` instead.

Translating Input Events

Variable: **extra-keyboard-modifiers**

This variable lets Lisp programs "press" the modifier keys on the keyboard. The value is a bit mask:

1
: The SHIFT key.

2
: The LOCK key.

4
: The CTL key.

8
: The META key.

Each time the user types a keyboard key, it is altered as if the modifier keys specified in the bit mask were held down.

When you use X windows, the program can "press" any of the modifier keys in this way. Otherwise, only the CTL and META keys can be virtually pressed.

Variable: **keyboard-translate-table**

This variable is the translate table for keyboard characters. It lets you reshuffle the keys on the keyboard

without changing any command bindings. Its value must be a string or `nil`.

If `keyboard-translate-table` is a string, then each character read from the keyboard is looked up in this string and the character in the string is used instead. If the string is of length n, character codes n and up are untranslated.

In the example below, we set `keyboard-translate-table` to a string of 128 characters. Then we fill it in to swap the characters C-s and C-\ and the characters C-q and C-^. Subsequently, typing C-\ has all the usual effects of typing C-s, and vice versa. (See section Flow Control for more information on this subject.)

```
(defun evade-flow-control ()
  "Replace C-s with C-\ and C-q with C-^."
  (interactive)
  (let ((the-table (make-string 128 0)))
    (let ((i 0))
      (while (< i 128)
        (aset the-table i i)
        (setq i (1+ i))))

    ;; Swap C-s and C-\.
    (aset the-table ?\034 ?\^s)
    (aset the-table ?\^s ?\034)
    ;; Swap C-q and C-^.
    (aset the-table ?\036 ?\^q)
    (aset the-table ?\^q ?\036)

    (setq keyboard-translate-table the-table)))
```

Note that this translation is the first thing that happens to a character after it is read from the terminal. Record-keeping features such as `recent-keys` and dribble files record the characters after translation.

Function: **keyboard-translate** *from to*

This function modifies `keyboard-translate-table` to translate character code from into character code to. It creates or enlarges the translate table if necessary.

Variable: **function-key-map**

This variable holds a keymap which describes the character sequences sent by function keys on an ordinary character terminal. This keymap uses the data structure as other keymaps, but is used differently: it specifies translations to make while reading events.

If `function-key-map` "binds" a key sequence k to a vector v, then when k appears as a subsequence *anywhere* in a key sequence, it is replaced with the events in v.

For example, VT100 terminals send ESC O P when the keypad PF1 key is pressed. Therefore, we want Emacs to translate that sequence of events into the single event `pf1`. We accomplish this by "binding"

ESC O P to `[pf1]` in `function-key-map`, when using a VT100.

Thus, typing C-c PF1 sends the character sequence C-c ESC O P; later the function `read-key-sequence` translates this back into C-c PF1, which it returns as the vector `[?\C-c pf1]`.

Entries in `function-key-map` are ignored if they conflict with bindings made in the minor mode, local, or global keymaps. The intent is that the character sequences that function keys send should not have command bindings in their own right.

The value of `function-key-map` is usually set up automatically according to the terminal's Terminfo or Termcap entry, but sometimes those need help from terminal-specific Lisp files. Emacs comes with a number of terminal-specific files for many common terminals; their main purpose is to make entries in `function-key-map` beyond those that can be deduced from Termcap and Terminfo. See section Terminal-Specific Initialization.

Emacs versions 18 and earlier used totally different means of detecting the character sequences that represent function keys.

Variable: **key-translation-map**

This variable is another keymap used just like `function-key-map` to translate input events into other events. It differs from `function-key-map` in two ways:

- `key-translation-map` goes to work after `function-key-map` is finished; it receives the results of translation by `function-key-map`.
- `key-translation-map` overrides actual key bindings.

The intent of `key-translation-map` is for users to map one character set to another, including ordinary characters normally bound to `self-insert-command`.

Recording Input

Function: **recent-keys**

This function returns a vector containing the last 100 input events from the keyboard or mouse. All input events are included, whether or not they were used as parts of key sequences. Thus, you always get the last 100 inputs, not counting keyboard macros. (Events from keyboard macros are excluded because they are less interesting for debugging; it should be enough to see the events which invoked the macros.)

Command: **open-dribble-file** *filename*

This function opens a dribble file named filename. When a dribble file is open, each input event from the keyboard or mouse (but not those from keyboard macros) are written in that file. A non-character event is expressed using its printed representation surrounded by `<...>'.

You close the dribble file by calling this function with an argument of `nil`. The function always returns `nil`.

This function is normally used to record the input necessary to trigger an Emacs bug, for the sake of a

bug report.

```
(open-dribble-file "~/dribble")
     => nil
```

See also the `open-termscript` function (see section Terminal Output).

Terminal Output

The terminal output functions send output to the terminal or keep track of output sent to the terminal. The variable `baud-rate` tells you what Emacs thinks is the output speed of the terminal.

Variable: **baud-rate**

This variable's value is the output speed of the terminal, as far as Emacs knows. Setting this variable does not change the speed of actual data transmission, but the value is used for calculations such as padding. It also affects decisions about whether to scroll part of the screen or repaint--even when using a window system, (We designed it this way despite the fact that a window system has no true "output speed", to give you a way to tune these decisions.)

The value is measured in baud.

If you are running across a network, and different parts of the network work at different baud rates, the value returned by Emacs may be different from the value used by your local terminal. Some network protocols communicate the local terminal speed to the remote machine, so that Emacs and other programs can get the proper value, but others do not. If Emacs has the wrong value, it makes decisions that are less than optimal. To fix the problem, set `baud-rate`.

Function: **baud-rate**

This function returns the value of the variable `baud-rate`. In Emacs versions 18 and earlier, this was the only way to find out the terminal speed.

Function: **send-string-to-terminal** *string*

This function sends string to the terminal without alteration. Control characters in string have terminal-dependent effects.

One use of this function is to define function keys on terminals that have downloadable function key definitions. For example, this is how on certain terminals to define function key 4 to move forward four characters (by transmitting the characters C-u C-f to the computer):

```
(send-string-to-terminal "\eF4\^U\^F")
     => nil
```

Command: **open-termscript** *filename*

This function is used to open a termscript file that will record all the characters sent by Emacs to the terminal. It returns `nil`. Termscript files are useful for investigating problems where Emacs garbles the

screen, problems which are due to incorrect Termcap entries or to undesirable settings of terminal options more often than actual Emacs bugs. Once you are certain which characters were actually output, you can determine reliably whether they correspond to the Termcap specifications in use.

See also `open-dribble-file` in section Terminal Input.

```
(open-termscript "../junk/termscript")
     => nil
```

Flow Control

This section attempts to answer the question "Why does Emacs choose to use flow-control characters in its command character set?" For a second view on this issue, read the comments on flow control in the `` `emacs/INSTALL' `` file from the distribution; for help with Termcap entries and DEC terminal concentrators, see `` `emacs/etc/TERMS' ``.

At one time, most terminals did not need flow control, and none used `C-s` and C-q for flow control. Therefore, the choice of C-s and C-q as command characters was unobjectionable. Emacs, for economy of keystrokes and portability, used nearly all the ASCII control characters, with mnemonic meanings when possible; thus, C-s for search and C-q for quote.

Later, some terminals were introduced which required these characters for flow control. They were not very good terminals for full-screen editing, so Emacs maintainers did not pay attention. In later years, flow control with C-s and C-q became widespread among terminals, but by this time it was usually an option. And the majority of users, who can turn flow control off, were unwilling to switch to less mnemonic key bindings for the sake of flow control.

So which usage is "right", Emacs's or that of some terminal and concentrator manufacturers? This is a rhetorical (or religious) question; it has no simple answer.

One reason why we are reluctant to cater to the problems caused by C-s and C-q is that they are gratuitous. There are other techniques (albeit less common in practice) for flow control that preserve transparency of the character stream. Note also that their use for flow control is not an official standard. Interestingly, on the model 33 teletype with a paper tape punch (which is very old), C-s and C-q were sent by the computer to turn the punch on and off!

GNU Emacs version 19 provides a convenient way of enabling flow control if you want it: call the function `enable-flow-control`.

Function: **enable-flow-control**

This function enables use of C-s and C-q for output flow control, and provides the characters C-\ and C-^ as aliases for them using `keyboard-translate-table` (see section Translating Input Events).

You can use the function `enable-flow-control-on` in your `` `.emacs' `` file to enable flow control automatically on certain terminal types.

Function: **enable-flow-control-on** *&rest termtypes*

This function enables flow control, and the aliases C-\ and C-^, if the terminal type is one of termtyp
For example:

```
(enable-flow-control-on "vt200" "vt300" "vt101" "vt131")
```

Here is how `enable-flow-control` does its job:

1. It sets CBREAK mode for terminal input, and tells the kernel to handle flow control, with `(set-input-mode nil t)`.
2. It sets up `keyboard-translate-table` to translate C-\ and C-^ into C-s and C-q were typed. Except at its very lowest level, Emacs never knows that the characters typed were anything but C-s and C-q, so you can in effect type them as C-\ and C-^ even when they are input for other commands. For example:

```
(setq keyboard-translate-table (make-string 128 0))
(let ((i 0))
  ;; Map most characters into themselves.
  (while (< i 128)
    (aset keyboard-translate-table i i)
    (setq i (1+ i))))
  ;; Map C-\ to C-s.
  (aset the-table ?\034 ?\^s)
  ;; Map C-^ to C-q.
  (aset the-table ?\036 ?\^q)))
```

If the terminal is the source of the flow control characters, then once you enable kernel flow control handling, you probably can make do with less padding than normal for that terminal. You can reduce the amount of padding by customizing the Termcap entry. You can also reduce it by setting `baud-rate` to a smaller value so that Emacs uses a smaller speed when calculating the padding needed. See section Terminal Output.

Batch Mode

The command line option `-batch' causes Emacs to run noninteractively. In this mode, Emacs does not read commands from the terminal, it does not alter the terminal modes, and it does not expect to be outputting to an erasable screen. The idea is that you specify Lisp programs to run; when they are finished, Emacs should exit. The way to specify the programs to run is with `-l file', which loads the library named file, and `-f function', which calls function with no arguments.

Any Lisp program output that would normally go to the echo area, either using `message` or using `prin1`, etc., with `t` as the stream, goes instead to Emacs's standard output descriptor when in batch mode. Thus, Emacs behaves much like a noninteractive application program. (The echo area output that Emacs itself normally generates, such as command echoing, is suppressed entirely.)

Variable: **noninteractive**

This variable is non-`nil` when Emacs is running in batch mode.

Go to the previous, next section.

Go to the previous, next section.

Emacs Display

This chapter describes a number of features related to the display that Emacs presents to the user.

Refreshing the Screen

The function `redraw-frame` redisplays the entire contents of a given frame. See section Frames.

Function: **redraw-frame** *frame*

This function clears and redisplays frame frame.

Even more powerful is `redraw-display`.

Command: **redraw-display**

This function clears and redisplays all visible frames.

Normally, suspending and resuming Emacs also refreshes the screen. Some terminal emulators record separate contents for display-oriented programs such as Emacs and for ordinary sequential display. If you are using such a terminal, you might want to inhibit the redisplay on resumption. See section Suspending Emacs.

Variable: **no-redraw-on-reenter**

This variable controls whether Emacs redraws the entire screen after it has been suspended and resumed. Non-`nil` means yes, `nil` means no.

Processing user input takes absolute priority over redisplay. If you call these functions when input is available, they do nothing immediately, but a full redisplay does happen eventually--after all the input has been processed.

Screen Size

The screen size functions report or tell Emacs the height or width of the terminal. When you are using multiple frames, they apply to the selected frame (see section Frames).

Function: **screen-height**

This function returns the number of lines on the screen that are available for display.

```
(screen-height)
     => 50
```

Function: **screen-width**

This function returns the number of columns on the screen that are available for display.

```
(screen-width)
     => 80
```

Function: **set-screen-height** *lines &optional not-actual-size*

This function declares that the terminal can display lines lines. The sizes of existing windows are altered proportionally to fit.

If not-actual-size is non-`nil`, then Emacs displays lines lines of output, but does not change its value for the actual height of the screen. (Knowing the correct actual size may be necessary for correct cursor positioning.) Using a smaller height than the terminal actually implements may be useful to reproduce behavior observed on a smaller screen, or if the terminal malfunctions when using its whole screen.

If lines is different from what it was previously, then the entire screen is cleared and redisplayed using the new size.

This function returns `nil`.

Function: **set-screen-width** *columns &optional not-actual-size*

This function declares that the terminal can display columns columns. The details are as in `set-screen-height`.

Truncation

When a line of text extends beyond the right edge of a window, the line can either be truncated or continued on the next line. When a line is truncated, this is shown with a `$' in the rightmost column of the window. When a line is continued or "wrapped" onto the next line, this is shown with a `\' on the rightmost column of the window. The additional screen lines used to display a long text line are called continuation lines. (Note that wrapped lines are not filled; filling has nothing to do with continuation and truncation. See section Filling.)

User Option: **truncate-lines**

This buffer-local variable controls how Emacs displays lines that extend beyond the right edge of the window. If it is non-`nil`, then Emacs does not display continuation lines; rather each line of text occupies exactly one screen line, and a dollar sign appears at the edge of any line that extends to or beyond the edge of the window. The default is `nil`.

If the variable `truncate-partial-width-windows` is non-`nil`, then truncation is used for windows that are not the full width of the screen, regardless of the value of `truncate-lines`.

Variable: **default-truncate-lines**

This variable is the default value for `truncate-lines` in buffers that do not have local values for it.

User Option: **truncate-partial-width-windows**

This variable determines how lines that are too wide to fit on the screen are displayed in side-by-side windows (see section Splitting Windows). If it is non-`nil`, then wide lines are truncated (with a `$' at the end of the line); otherwise they wrap to the next screen line (with a `\' at the end of the line).

You can override the images that indicate continuation or truncation with the display table; see section Display Tables.

The Echo Area

The echo area is used for displaying messages made with the `message` primitive, and for echoing keystrokes. It is not the same as the minibuffer, despite the fact that the minibuffer appears (when active) in the same place on the screen as the echo area. The GNU Emacs Manual specifies the rules for resolving conflicts between the echo area and the minibuffer for use of that screen space (see section 'The Minibuffer' in The GNU Emacs Manual). Error messages appear in the echo area; see section Errors.

You can write output in the echo area by using the Lisp printing functions with `t` as the stream (see section Output Functions), or as follows:

Function: **message** *string &rest arguments*

This function prints a one-line message in the echo area. The argument string is similar to a C language `printf` control string. See `format` in section Conversion of Characters and Strings, for the details on the conversion specifications. `message` returns the constructed string.

If string is `nil`, `message` clears the echo area. If the minibuffer is active, this brings the minibuffer contents back onto the screen immediately.

```
(message
 "Minibuffer depth is %d."
 (minibuffer-depth))
=> "Minibuffer depth is 0."

---------- Echo Area ----------
Minibuffer depth is 0.
---------- Echo Area ----------
```

Variable: **cursor-in-echo-area**

This variable controls where the cursor appears when a message is displayed in the echo area. If it is non-`nil`, then the cursor appears at the end of the message. Otherwise, the cursor appears at point--not in the echo area at all.

The value is normally `nil`; Lisp programs bind it to `t` for brief periods of time.

Selective Display

Selective display is a class of minor modes in which specially marked lines do not appear on the screen, or in which highly indented lines do not appear.

The first variant, explicit selective display, is designed for use in a Lisp program. The program controls which lines are hidden by altering the text. Outline mode uses this variant. In the second variant, the choice of lines to hide is made automatically based on indentation. This variant is designed as a user-level feature.

The way you control explicit selective display is by replacing a newline (control-j) with a control-m. The text which was formerly a line following that newline is now invisible. Strictly speaking, it is temporarily no longer a line at all, since only newlines can separate lines; it is now part of the previous line.

Selective display does not directly affect editing commands. For example, C-f (`forward-char`) moves point unhesitatingly into invisible space. However, the replacement of newline characters with carriage return characters affects some editing commands. For example, `next-line` skips invisible lines, since it searches only for newlines. Modes that use selective display can also define commands that take account of the newlines, or which make parts of the text visible or invisible.

When you write a selectively displayed buffer into a file, all the control-m's are replaced by their original newlines. This means that when you next read in the file, it looks OK, with nothing invisible. The selective display effect is seen only within Emacs.

Variable: **selective-display**

This buffer-local variable enables selective display. This means that lines, or portions of lines, may be made invisible.

- If the value of `selective-display` is `t`, then any portion of a line that follows a control-m is not displayed.
- If the value of `selective-display` is a positive integer, then lines that start with more than `selective-display` columns of indentation are not displayed.

When some portion of a buffer is invisible, the vertical movement commands operate as if that portion did not exist, allowing a single `next-line` command to skip any number of invisible lines. However, character movement commands (such as `forward-char`) do not skip the invisible portion, and it is possible (if tricky) to insert or delete text in an invisible portion.

In the examples below, what is shown is the *display* of the buffer `foo`, which changes with the value of `selective-display`. The *contents* of the buffer do not change.

```
(setq selective-display nil)
     => nil

---------- Buffer: foo ----------
1 on this column
 2on this column
```


```
  3n this column
  3n this column
 2on this column
1 on this column
---------- Buffer: foo ----------

(setq selective-display 2)
     => 2

---------- Buffer: foo ----------
1 on this column
 2on this column
 2on this column
1 on this column
---------- Buffer: foo ----------
```

Variable: **selective-display-ellipses**

If this buffer-local variable is non-`nil`, then Emacs displays `...' at the end of a line that is followed by invisible text. This example is a continuation of the previous one.

```
(setq selective-display-ellipses t)
     => t

---------- Buffer: foo ----------
1 on this column
 2on this column ...
 2on this column
1 on this column
---------- Buffer: foo ----------
```

You can use a display table to substitute other text for the ellipsis (`...'). See section Display Tables.

Overlay Arrow

The overlay arrow is useful for directing the user's attention to a particular line in a buffer. For example, in the modes used for interface to debuggers, the overlay arrow indicates the line of code about to be executed.

Variable: **overlay-arrow-string**

This variable holds the string to display as an arrow, or `nil` if the arrow feature is not in use.

Variable: **overlay-arrow-position**

This variable holds a marker which indicates where to display the arrow. It should point at the beginning of a line. The arrow text is displayed at the beginning of that line, overlaying any text that would

otherwise appear. Since the arrow is usually short, and the line usually begins with indentation, norm nothing significant is overwritten.

The overlay string is displayed only in the buffer which this marker points into. Thus, only one buffer can have an overlay arrow at any given time.

Temporary Displays

Temporary displays are used by commands to put output into a buffer and then present it to the user for perusal rather than for editing. Many of the help commands use this feature.

Special Form: **with-output-to-temp-buffer** *buffer-name forms...*

This function executes forms while arranging to insert any output they print into the buffer named buffer-name. The buffer is then shown in some window for viewing, displayed but not selected.

The string buffer-name specifies the temporary buffer, which need not already exist. The argument must be a string, not a buffer. The buffer is erased initially (with no questions asked), and it is marked as unmodified after `with-output-to-temp-buffer` exits.

`with-output-to-temp-buffer` binds `standard-output` to the temporary buffer, then it evaluates the forms in forms. Output using the Lisp output functions within forms goes by default to that buffer (but screen display and messages in the echo area, although output in the general sense of the word, are not affected). See section Output Functions.

The value of the last form in forms is returned.

```
---------- Buffer: foo ----------
 This is the contents of foo.
---------- Buffer: foo ----------

(with-output-to-temp-buffer "foo"
    (print 20)
    (print standard-output))
=> #<buffer foo>

---------- Buffer: foo ----------
20

#<buffer foo>

---------- Buffer: foo ----------
```

Variable: **temp-buffer-show-function**

The value of this variable, if non-`nil`, is called as a function to display a help buffer. This variable is used by `with-output-to-temp-buffer`.

In Emacs versions 18 and earlier, this variable was called `temp-buffer-show-hook`.

Function: **momentary-string-display** *string position &optional char message*

This function momentarily displays string in the current buffer at position (which is a character offset from the beginning of the buffer). The display remains until the next character is typed.

If the next character the user types is char, Emacs ignores it. Otherwise, that character remains buffered for subsequent use as input. Thus, typing char will simply remove the string from the display, while typing (say) C-f will remove the string from the display and later (presumably) move point forward. The argument char is a space by default.

The return value of `momentary-string-display` is not meaningful.

If message is non-`nil`, it is displayed in the echo area while string is displayed in the buffer. If it is `nil`, then instructions to type char are displayed there, e.g., `Type RET to continue editing'.

In this example, point is initially located at the beginning of the second line:

```
---------- Buffer: foo ----------
This is the contents of foo.
-!-Second line.
---------- Buffer: foo ----------

(momentary-string-display
   "**** Important Message! ****" (point) ?\r
   "Type RET when done reading")
=> t

---------- Buffer: foo ----------
This is the contents of foo.
**** Important Message! ****Second line.
---------- Buffer: foo ----------

---------- Echo Area ----------
Type RET when done reading
---------- Echo Area ----------
```

This function works by actually changing the text in the buffer. As a result, if you later undo in this buffer, you will see the message come and go.

Overlays

You can use overlays to alter the appearance of a buffer's text on the screen. An overlay is an object which belongs to a particular buffer, and has a specified beginning and end. It also has properties which you can examine and set; these affect the display of the text within the overlay.

Overlay Properties

Overlay properties are like text properties in some respects, but the differences are more important than the similarities. Text properties are considered a part of the text; overlays are specifically considered not to be part of the text. Thus, copying text between various buffers and strings preserves text properties, but does not try to preserve overlays. Changing a buffer's text properties marks the buffer as modified, while moving an overlay or changing its properties does not.

`face`

This property controls the font and color of text. See section Faces, for more information. This feature is temporary; in the future, we may replace it with other ways of specifying how to display text.

`mouse-face`

This property is used instead of `face` when the mouse is within the range of the overlay. This feature is not yet implemented, and may be temporary. It is documented here because we are likely to implement it this way at least for a while.

`priority`

This property's value (which should be a nonnegative number) determines the priority of the overlay. The priority matters when two or more overlays cover the same character and both specify a face for display; the one whose `priority` value is larger takes priority over the other, and its face attributes override the face attributes of the lower priority overlay.

Currently, all overlays take priority over text properties. Please avoid using negative priority values, as we have not yet decided just what they should mean.

`window`

If the `window` property is non-`nil`, then the overlay applies only on that window.

`before-string`

This property's value is a string to add to the display at the beginning of the overlay. The string does not appear in the buffer in any sense--only on the screen. This is not yet implemented, but will be.

`after-string`

This property's value is a string to add to the display at the end of the overlay. The string does not appear in the buffer in any sense--only on the screen. This is not yet implemented, but will be.

These are the functions for reading and writing the properties of an overlay.

Function: **overlay-get** *overlay prop*

This function returns the value of property prop recorded in overlay. If overlay does not record any value for that property, then the value is `nil`.

Function: **overlay-put** *overlay prop value*

This function set the value of property prop recorded in overlay to value. It returns value.

Managing Overlays

Function: **make-overlay** *start end &optional buffer*

This function creates and returns an overlay which belongs to buffer and ranges from start to end. Both start and end must specify buffer positions; they may be integers or markers. If buffer is omitted, the overlay is created in the current buffer.

The return value is the overlay itself.

Function: **overlay-start** *overlay*

This function returns the position at which overlay starts.

Function: **overlay-end** *overlay*

This function returns the position at which overlay ends.

Function: **overlay-buffer** *overlay*

This function returns the buffer that overlay belongs to.

Function: **delete-overlay** *overlay*

This function deletes overlay. The overlay continues to exist as a Lisp object, but ceases to be part of the buffer it belonged to, and ceases to have any effect on display.

Function: **move-overlay** *overlay start end &optional buffer*

This function moves overlay to buffer, and places its bounds at start and end. Both arguments start and end must specify buffer positions; they may be integers or markers. If buffer is omitted, the overlay stays in the same buffer.

The return value is overlay.

This is the only valid way to change the endpoints of an overlay. Do not try modifying the markers in the overlay by hand, as that fails to update other vital data structures and can cause some overlays to be "lost".

Function: **overlays-at** *pos*

This function returns a list of all the overlays that contain position pos in the current buffer. The list is in no particular order. An overlay contains position pos if it begins at or before pos, and ends after pos.

Function: **next-overlay-change** *pos*

This function returns the buffer position of the next beginning or end of an overlay, after pos.

Faces

A face is a named collection of graphical attributes: font, foreground color, background color and optional underlining. Faces control the display of text on the screen.

Each face has its own face id number which distinguishes faces at low levels within Emacs. However, for most purposes, you can refer to faces in Lisp programs by their names.

Each face name is meaningful for all frames, and by default it has the same meaning in all frames. But you can arrange to give a particular face name a special meaning in one frame if you wish.

The face named `default` is used for ordinary text. The face named `modeline` is used for displaying the mode line and menu bars. The face named `region` is used for highlighting the region (in Transient Mark mode only).

Merging Faces for Display

Here are all the ways to specify which face to use for display of text:

- With defaults. Each frame has a default face, whose id number is zero, which is used for all text that doesn't somehow specify another face.
- With text properties. A character may have a `face` property; if so, it's displayed with that face. If the character has a `mouse-face` property, that is used instead of the `face` property when the mouse is "near enough" to the character. See section Special Properties.
- With overlays. An overlay may have `face` and `mouse-face` properties too; they apply to all the text covered by the overlay.
- With special glyphs. Each glyph can specify a particular face id number. See section Glyphs.

If these various sources together specify more than one face for a particular character, Emacs merges the attributes of the various faces specified. The attributes of the faces of special glyphs come first; then come attributes of faces from overlays, followed by those from text properties, and last the default face.

When multiple overlays cover one character, an overlay with higher priority overrides those with lower priority. See section Overlays.

If an attribute such as the font or a color is not specified in any of the above ways, the frame's own font or color is used.

Functions for Working with Faces

The attributes a face can specify include the font, the foreground color, the background color, and underlining. The face can also leave these unspecified by giving the value `nil` for them.

Here are the primitives for creating and changing faces.

Function: **make-face** *name*

This function defines a new face named name, initially with all attributes `nil`. It does nothing if there already a face named name.

Function: **face-list**

This function returns a list of all defined face names.

Function: **copy-face** *old-face new-name &optional frame*

This function defines a new face named new which is a copy of the existing face named old. If there is already a face named new, then it alters the face to have the same attributes as old.

If the optional argument frame is given, this function applies only to that frame. Otherwise it applies to each frame individually.

You can modify the attributes of an existing face with the following functions. If you specify frame, they affect just that frame; otherwise, they affect all frames as well as the defaults that apply to new frames.

Function: **set-face-foreground** *face color &optional frame*

Function: **set-face-background** *face color &optional frame*

These functions set the foreground (respectively, background) color of face face to color. The argument color color should be a string, the name of a color.

Function: **set-face-font** *face font &optional frame*

This function sets the font of face face. The argument font should be a string.

Function: **set-face-underline-p** *face underline-p &optional frame*

This function sets the underline attribute of face face.

Function: **invert-face** *face &optional frame*

Swap the foreground and background colors of face face. If the face doesn't specify both foreground and background, then its foreground and background are set to the default background and foreground.

These functions examine the attributes of a face. If you don't specify frame, they refer to the default data for new frames.

Function: **face-foreground** *face &optional frame*

Function: **face-background** *face &optional frame*

These functions return the foreground (respectively, background) color of face face. The argument color color should be a string, the name of a color.

Function: **face-font** *face &optional frame*

This function returns the name of the font of face face.

Function: **face-underline-p** *face &optional frame*

This function returns the underline attribute of face face.

Function: **face-id-number** *face*

This function returns the id number of face face.

Function: **face-equal** *face1 face2 &optional frame*

This returns `t` if the faces face1 and face2 have the same attributes for display.

Function: **face-differs-from-default-p** *face &optional frame*

This returns `t` if the face face displays differently from the default face. A face is considered to be "the same" as the normal face if each attribute is either the same as that of the default face or `nil` (meaning to inherit from the default).

Variable: **region-face**

This variable's value specifies the face id to use to display characters in the region when it is active (in Transient Mark mode only). The face thus specified takes precedence over all faces that come from text properties and overlays, for characters in the region. See section The Mark, for more information about Transient Mark mode.

Normally, the value is the id number of the face named `region`.

Blinking

This section describes the mechanism by which Emacs shows a matching open parenthesis when the user inserts a close parenthesis.

Variable: **blink-paren-function**

The value of this variable should be a function (of no arguments) to be called whenever a char with close parenthesis syntax is inserted. The value of `blink-paren-function` may be `nil`, in which case nothing is done.

> **Please note:** this variable was named `blink-paren-hook` in older Emacs versions, but since it is not called with the standard convention for hooks, it was renamed to `blink-paren-function` in version 19.

Variable: **blink-matching-paren**

If this variable is `nil`, then `blink-matching-open` does nothing.

Variable: **blink-matching-paren-distance**

This variable specifies the maximum distance to scan for a matching parenthesis before giving up.

Function: **blink-matching-open**

This function is the default value of `blink-paren-function`. It assumes that point follows a character with close parenthesis syntax and moves the cursor momentarily to the matching opening

character. If that character is not already on the screen, then its context is shown by displaying it in the echo area. To avoid long delays, this function does not search farther than `blink-matching-paren-distance` characters.

Here is an example of calling this function explicitly.

```
(defun interactive-blink-matching-open ()
  "Indicate momentarily the start of sexp before point."
  (interactive)
  (let ((blink-matching-paren-distance
         (buffer-size))
        (blink-matching-paren t))
    (blink-matching-open)))
```

Inverse Video

User Option: **inverse-video**

This variable controls whether Emacs uses inverse video for all text on the screen. Non-`nil` means yes, `nil` means no. The default is `nil`.

User Option: **mode-line-inverse-video**

This variable controls the use of inverse video for mode lines. If it is non-`nil`, then mode lines are displayed in inverse video (under X, this uses the face named `modeline`, which you can set as you wish). Otherwise, mode lines are displayed normally, just like text. The default is `t`.

Usual Display Conventions

The usual display conventions define how to display each character code. You can override these conventions by setting up a display table (see section Display Tables). Here are the usual display conventions:

- Character codes 32 through 126 map to glyph codes 32 through 126. Normally this means they display as themselves.
- Character code 9 is a horizontal tab. It displays as whitespace up to a position determined by `tab-width`.
- Character code 10 is a newline.
- All other codes in the range 0 through 31, and code 127, display in one of two ways according to the value of `ctl-arrow`. If it is is non-`nil`, these codes map to sequences of two glyphs, where the first glyph is the ASCII code for `^'. Otherwise, these codes map just like the codes in the range 128 to 255.
- Character codes 128 through 255 map to sequences of four glyphs, where the first glyph is the ASCII code for `\', and the others are digit characters representing the code in octal.

The usual display conventions apply even when there is a display table, for any character whose entry in the active display table is `nil`. Thus, when you set up a display table, you need only specify the the characters for which you want unusual behavior.

These variables affect the way certain characters are displayed on the screen. Since they change the number of columns the characters occupy, they also affect the indentation functions.

User Option: **ctl-arrow**

This buffer-local variable controls how control characters are displayed. If it is non-`nil`, they are displayed as a caret followed by the character: `^A'. If it is `nil`, they are displayed as a backslash followed by three octal digits: `\001'.

Variable: **default-ctl-arrow**

The value of this variable is the default value for `ctl-arrow` in buffers that do not override it. This is the same as executing the following expression:

```
(default-value 'ctl-arrow)
```

See section The Default Value of a Buffer-Local Variable.

User Option: **tab-width**

The value of this variable is the spacing between tab stops used for displaying tab characters in Emacs buffers. The default is 8. Note that this feature is completely independent from the user-settable tab stops used by the command `tab-to-tab-stop`. See section Adjustable "Tab Stops".

Display Tables

You can use the display table feature to control how all 256 possible character codes display on the screen. This is useful for displaying European languages that have letters not in the ASCII character set.

The display table maps each character code into a sequence of glyphs, each glyph being an image that takes up one character position on the screen. You can also define how to display each glyph on your terminal, using the glyph table.

Display Table Format

A display table is actually an array of 261 elements.

Function: **make-display-table**

This creates and returns a display table. The table initially has `nil` in all elements.

The first 256 elements correspond to character codes; the nth element says how to display the character code n. The value should be `nil` or a vector of glyph values (see section Glyphs). If an element is `nil`, it says to display that character according to the usual display conventions (see section Usual Display

Conventions).

The remaining five elements of a display table serve special purposes, and `nil` means use the default stated below.

256
: The glyph for the end of a truncated screen line (the default for this is `$'). See section Glyphs.

257
: The glyph for the end of a continued line (the default is `\').

258
: The glyph for indicating a character displayed as an octal character code (the default is `\').

259
: The glyph for indicating a control character (the default is `^').

260
: A vector of glyphs for indicating the presence of invisible lines (the default is `...'). See section Selective Display.

For example, here is how to construct a display table that mimics the effect of setting `ctl-arrow` to a non-`nil` value:

```
(setq disptab (make-display-table))
(let ((i 0))
  (while (< i 32)
    (or (= i ?\t) (= i ?\n)
        (aset disptab i (vector ?^ (+ i 64))))
    (setq i (1+ i)))
  (aset disptab 127 (vector ?^ ??)))
```

Active Display Table

Each window can specify a display table, and so can each buffer. When a buffer b is displayed in window w, display uses the display table for window w if it has one; otherwise, the display table for buffer b if it has one; otherwise, the standard display table if any. The display table chosen is called the active display table.

Function: **window-display-table** *window*

This function returns window's display table, or `nil` if window does not have an assigned display table.

Function: **set-window-display-table** *window table*

This function sets the display table of window to table. The argument table should be either a display table or `nil`.

Variable: **buffer-display-table**

This variable is automatically local in all buffers; its value in a particular buffer is the display table f that buffer, or `nil` if the buffer does not have any assigned display table.

Variable: **standard-display-table**

This variable's value is the default display table, used when neither the current buffer nor the window displaying it has an assigned display table. This variable is `nil` by default.

If neither the selected window nor the current buffer has a display table, and if the variable `standard-display-table` is `nil`, then Emacs uses the usual display conventions. See section Usual Display Conventions.

Glyphs

A glyph is a generalization of a character; it stands for an image that takes up a single character position on the screen. Glyphs are represented in Lisp as integers, just as characters are.

The meaning of each integer, as a glyph, is defined by the glyph table, which is the value of the variable `glyph-table`.

Variable: **glyph-table**

The value of this variable is the current glyph table. It should be a vector; the gth element defines glyph code g. If the value is `nil` instead of a vector, then all glyphs are simple (see below).

Here are the possible types of elements in the glyph table:

integer

Define this glyph code as an alias for code integer. This is used with X Windows to specify a face code.

string

Send the characters in string to the terminal to output this glyph. This alternative is available on character terminals, but not under X.

`nil`

This glyph is simple. On an ordinary terminal, the glyph code mod 256 is the character to output. With X, the glyph code mod 256 is character to output, and the glyph code divided by 256 specifies the face id number to use while outputting it. See section Faces.

If a glyph code is greater than or equal to the length of the glyph table, that code is automatically simple.

ISO Latin 1

If you have a terminal that can handle the entire ISO Latin 1 character set, you can arrange to use that character set as follows:

```
(require 'disp-table)
;; Set char codes 160--255 to display as themselves.
```


```
;; (Codes 128--159 are the additional control characters.)
(standard-display-8bit 160 255)
```

If you are editing buffers written in the ISO Latin 1 character set and your terminal doesn't handle anything but ASCII, you can load the file `iso-ascii' to set up a display table which makes the other ISO characters display as sequences of ASCII characters. For example, the character "o with umlaut" displays as `{"o}'.

Some European countries have terminals that don't support ISO Latin 1 but do support the special characters for that country's language. You can define a display table to work one language using such terminals. For an example, see `lisp/iso-swed.el', which handles certain Swedish terminals.

You can load the appropriate display table for your terminal automatically by writing a terminal-specific Lisp file for the terminal type.

Beeping

You can make Emacs ring a bell (or blink the screen) to attract the user's attention. Be conservative about how often you do this; frequent bells can become irritating. Also be careful not to use beeping alone when signaling an error is appropriate. (See section Errors.)

Function: **ding** *&optional dont-terminate*

This function beeps, or flashes the screen (see `visible-bell` below). It also terminates any keyboard macro currently executing unless dont-terminate is non-`nil`.

Function: **beep** *&optional dont-terminate*

This is a synonym for `ding`.

Variable: **visible-bell**

This variable determines whether Emacs should flash the screen to represent a bell. Non-`nil` means yes, `nil` means no. This is effective only if the Termcap entry for the terminal in use has the visible bell flag (`vb') set.

Window Systems

Emacs works with several window systems, most notably the X Window Syste,. Note that both Emacs and X use the term "window", but use it differently. An Emacs frame is a single window as far as X is concerned; the individual Emacs windows are not known to X at all.

Variable: **window-system**

This variable tells Lisp programs what window system Emacs is running under. Its value should be a symbol such as `x` (if Emacs is running under X) or `nil` (if Emacs is running on an ordinary terminal).

Variable: **window-system-version**

This variable distinguishes between different versions of the X Window System. Its value is 10 or 11 when using X; `nil` otherwise.

Variable: **window-setup-hook**

This variable is a normal hook which Emacs runs after loading your `` `.emacs' `` file and the default initialization file (if any), after loading terminal-specific Lisp code, and after running the hook `term-setup-hook`.

This hook is used for internal purposes: setting up communication with the window system, and creating the initial window. Users should not interfere with it.

Go to the previous, next section.

Go to the previous, next section.

Customizing the Calendar and Diary

There are many customizations that you can use to make the calendar and diary suit your personal tastes.

Customizing the Calendar

If you set the variable `view-diary-entries-initially` to `t`, calling up the calendar automatically displays the diary entries for the current date as well. The diary dates appear only if the current date is visible. If you add both of the following lines to your `.emacs' file:

```
(setq view-diary-entries-initially t)
(calendar)
```

they display both the calendar and diary windows whenever you start Emacs.

Similarly, if you set the variable `view-calendar-holidays-initially` to `t`, entering the calendar automatically displays a list of holidays for the current three month period. The holiday list appears in a separate window.

You can set the variable `mark-diary-entries-in-calendar` to `t` in order to place a plus sign (`+') beside any dates with diary entries. Whenever the calendar window is displayed or redisplayed, the diary entries are automatically marked for holidays.

Similarly, setting the variable `mark-holidays-in-calendar` to `t` places an asterisk (`*') after all holiday dates visible in the calendar window.

There are many customizations that you can make with the hooks provided. For example, the variable `calendar-load-hook`, whose default value is `nil`, is a normal hook run when the calendar package is first loaded (before actually starting to display the calendar).

The variable `initial-calendar-window-hook`, whose default value is `nil`, is a normal hook run the first time the calendar window is displayed. The function is invoked only when you first enter Calendar mode, not when you redisplay an existing Calendar window. But if you leave the calendar with the q command and reenter it, the hook runs again.

The variable `today-visible-calendar-hook`, whose default value is `nil`, is a normal hook run after the calendar buffer has been prepared with the calendar when the current date is visible in the window. One use of this hook is to replace today's date with asterisks; a function `calendar-star-date` is included for this purpose. In your `.emacs' file, put:

```
(setq today-visible-calendar-hook 'calendar-star-date)
```

Another standard hook function adds asterisks around the current date. Here's how to use it:

```
(setq today-visible-calendar-hook 'calendar-mark-today)
```

A corresponding variable, `today-invisible-calendar-hook`, whose default value is `nil`, is a normal hook run after the calendar buffer text has been prepared, if the current date is *not* visible in the window.

Customizing the Holidays

Emacs knows about holidays defined by entries on one of several lists. You can customize theses lists of holidays to your own needs, adding holidays or deleting lists of holidays. The lists of holidays that Emacs uses are for general holidays (`general-holidays`), local holidays (`local-holidays`), Christian holidays (`christian-holidays`), Hebrew (Jewish) holidays (`hebrew-holidays`), Islamic (Moslem) holidays (`islamic-holidays`), and other holidays (`other-holidays`).

The general holidays are, by default, holidays common throughout the United States. To eliminate these holidays, set `general-holidays` to `nil`.

There are no default local holidays (but sites may supply some). You can set the variable `local-holidays` to any list of holidays, as described below.

By default, Emacs does not consider all the holidays of these religions, only those commonly found in secular calendars. For a more extensive collection of religious holidays, you can set any (or all) of the variables `all-christian-calendar-holidays`, `all-hebrew-calendar-holidays`, or `all-islamic-calendar-holidays` to `t`. If you want to eliminate the religious holidays, set any or all of the corresponding variables `christian-holidays`, `hebrew-holidays`, and `islamic-holidays` to `nil`.

You can set the variable `other-holidays` to any list of holidays. This list, normally empty, is intended for your use.

Each of the lists (`general-holidays`), (`local-holidays`), (`christian-holidays`), (`hebrew-holidays`), (`islamic-holidays`),and (`other-holidays`) is a list of holiday forms, each holiday form describing a holiday (or sometimes a list of holidays). Holiday forms may have the following formats:

`(fixed month day string)`

A fixed date on the Gregorian calendar. month and day are numbers, string is the name of the holiday.

`(float month dayname k string)`

The kth dayname in month on the Gregorian calendar (dayname=0 for Sunday, and so on); negative k means count back from the end of the month. string is the name of the holiday.

`(hebrew month day string)`

A fixed date on the Hebrew calendar. month and day are numbers, string is the name of the holiday.

`(islamic month day string)`

A fixed date on the Islamic calendar. month and day are numbers, string is the name of the holiday.

`(julian month day string)`

A fixed date on the Julian calendar. month and day are numbers, string is the name of the holiday.

`(sexp sexp string)`

sexp is a Lisp expression that should use the variable `year` to compute the date of a holiday, or `nil` if the holiday doesn't happen this year. The value represents the date as a list of the form `(month day year)`. string is the name of the holiday.

`(if boolean holiday-form &optional holiday-form)`

A choice between two holidays based on the value of boolean.

`(function &optional args)`

Dates requiring special computation; args, if any, are passed in a list to the function `calendar-holiday-function-function`.

For example, suppose you want to add Bastille Day, celebrated in France on July 14. You can do this by adding the following line to your `.emacs' file:

```
(setq other-holidays '((fixed 7 14 "Bastille Day")))
```

The holiday form `(fixed 7 14 "Bastille Day")` specifies the fourteenth day of the seventh month (July).

Many holidays occur on a specific day of the week, at a specific time of month. Here is a holiday form describing Hurricane Supplication Day, celebrated in the Virgin Islands on the fourth Monday in August:

```
(float 8 1 4 "Hurricane Supplication Day")
```

Here the 8 specifies August, the 1 specifies Monday (Sunday is 0, Tuesday is 2, and so on), and the 4 specifies the fourth occurrence in the month (1 specifies the first occurrence, 2 the second occurrence, -1 the last occurrence, -2 the second-to-last occurrence, and so on).

You can specify holidays that occur on fixed days of the Hebrew, Islamic, and Julian calendars too. For example,

```
(setq other-holidays
      '((hebrew 10 2 "Last day of Hanukkah")
        (islamic 3 12 "Mohammed's Birthday")
        (julian 4 2 "Jefferson's Birthday")))
```

adds the last day of Hanukkah (since the Hebrew months are numbered with 1 starting from Nisan), the Islamic feast celebrating Mohammed's birthday (since the Islamic months are numbered from 1 starting with Muharram), and Thomas Jefferson's birthday, which is 2 April 1743 on the Julian calendar.

To include a holiday conditionally, use either the `if' or the `sexp' form. For example, American presidential elections occur on the first Tuesday after the first Monday in November of years divisible by 4:

```
(sexp (if (= 0 (% year 4))
          (calendar-gregorian-from-absolute
            (1+ (calendar-dayname-on-or-before
                  1 (+ 6 (calendar-absolute-from-gregorian
                           (list 11 1 year))))))
      "US Presidential Election"))
```

or

```
(if (= 0 (% displayed-year 4))
    (fixed 11
           (extract-calendar-day
             (calendar-gregorian-from-absolute
               (1+ (calendar-dayname-on-or-before
                     1 (+ 6 (calendar-absolute-from-gregorian
                              (list 11 1 displayed-year)))))))
           "US Presidential Election"))
```

Some holidays just don't fit into any of these forms because special calculations are involved in their determination. In such cases you must write a Lisp function to do the calculation. The function should return a (possibly empty) list of the relevant Gregorian dates among the range visible in the calendar window, with descriptive strings, like this:

```
(((6 27 1991) "Lunar Eclipse") ((7 11 1991) "Solar Eclipse") ... )
```

Date Display Format

You can customize the manner of displaying dates in the diary, in mode lines, and in messages by setting `calendar-date-display-form`. This variable is a list of expressions that can involve the variables `month`, `day`, and `year`, all numbers in string form, and `monthname` and `dayname`, both alphabetic strings. In the American style, the default value of this list is as follows:

```
((if dayname (concat dayname ", ")) monthname " " day ", " year)
```

while in the European style this value is the default:

```
((if dayname (concat dayname ", ")) day " " monthname " " year)
```

The ISO standard date representation is this:

```
(year "-" month "-" day)
```

This specifies a typical American format:

```
(month "/" day "/" (substring year -2))
```

Time Display Format

In the calendar, diary, and related buffers, Emacs displays times of day in the conventional American style with the hours from 1 through 12, minutes, and either `am' or `pm'. If you prefer the "military" (European) style of writing times--in which the hours go from 00 to 23--you can alter the variable `calendar-time-display-form`. This variable is a list of expressions that can involve the variables `12-hours`, `24-hours`, and `minutes`, all numbers in string form, and `am-pm` and `time-zone`, both alphabetic strings. The default definition of `calendar-time-display-form` is as follows:

```
(12-hours ":" minutes am-pm (if time-zone " (") time-zone (if time-zone ")"))
```

Setting `calendar-time-display-form` to

```
(24-hours ":" minutes (if time-zone " (") time-zone (if time-zone ")"))
```

gives military-style times like `21:07 (UT)' if time zone names are defined, and times like `21:07' if they are not.

Daylight Savings Time

Emacs understands the difference between standard time and daylight savings time--the times given for sunrise, sunset, solstices, equinoxes, and the phases of the moon take that into account. The default starting and stopping dates for daylight savings time are the present-day American rules of the first Sunday in April until the last Sunday in October, but you can specify whatever rules you want by setting `calendar-daylight-savings-starts` and `calendar-daylight-savings-ends`. Their values should be Lisp expressions that refer to the variable `year`, and evaluate to the Gregorian date on which daylight savings time starts or (respectively) ends, in the form of a list `(month day year)`.

Emacs uses these expressions to determine the starting date of daylight savings time for the holiday list and for correcting times of day in the solar and lunar calculations.

The default value of `calendar-daylight-savings-starts` is this,

```
(calendar-nth-named-day 1 0 4 year)
```

which computes the first 0th day (Sunday) of the fourth month (April) in the year specified by `year`. If daylight savings time were changed to start on October 1, you would set `calendar-daylight-savings-starts` to

```
(list 10 1 year)
```

For a more complex example, suppose daylight savings time begins on the first of Nisan on the Hebrew calendar. You would set `calendar-daylight-savings-starts` to

```
(calendar-gregorian-from-absolute
  (calendar-absolute-from-hebrew
    (list 1 1 (+ year 3760))))
```

because Nisan is the first month in the Hebrew calendar and the Hebrew year differs from the Gregorian year by 3760 at Nisan.

If there is no daylight savings time at your location, or if you want all times in standard time, set `calendar-daylight-savings-starts` and `calendar-daylight-savings-ends` to `nil`.

Customizing the Diary

Ordinarily, the mode line of the diary buffer window indicates any holidays that fall on the date of the diary entries. The process of checking for holidays can take several seconds, so including holiday information delays the display of the diary buffer noticeably. If you'd prefer to have a faster display of the diary buffer but without the holiday information, set the variable `holidays-in-diary-buffer` to `nil`.

The variable `number-of-diary-entries` controls the number of days of diary entries to be displayed at one time. It affects the initial display when `view-diary-entries-initially` is `t`, as well as the command M-x diary. For example, the default value is 1, which says to display only the current day's diary entries. If the value is 2, both the current day's and the next day's entries are displayed. The value can also be a vector of seven elements: if the value is `[0 2 2 2 2 4 1]` then no diary entries appear on Sunday, the current date's and the next day's diary entries appear Monday through Thursday, Friday through Monday's entries appear on Friday, while on Saturday only that day's entries appear.

The variable `print-diary-entries-hook` is a normal hook run after preparation of a temporary buffer containing just the diary entries currently visible in the diary buffer. (The other, irrelevant diary entries are rea absent from the temporary buffer; in the diary buffer, they are merely hidden.) The default value of this hook does the printing with the command `lpr-buffer`. If you want to use a different command to do the printing, just change the value of this hook. Other uses might include, for example, rearranging the lines into order by day and time.

You can customize the form of dates in your diary file, if neither the standard American nor European styles suits your needs, by setting the variable `diary-date-forms`. This variable is a list of forms of dates recognized in the diary file. Each form is a list of regular expressions (see section Regular Expressions) and the variables `month`, `day`, `year`, `monthname`, and `dayname`. The variable `monthname` matches the name of the month, capitalized or not, or its three-letter abbreviation, followed by a period or not; it matches `*'. Similarly, `dayname` matches the name of the day, capitalized or not, or its three-letter abbreviation, followed by a period or not. The variables `month`, `day`, and `year` match those numerical values, preceded by arbitrarily many zeros; they also match `*'. The default value of `diary-date-forms` in the American style is

```
((month "/" day "[^/0-9]")
 (month "/" day "/" year "[^0-9]")
 (monthname " *" day "[^,0-9]")
 (monthname " *" day ", *" year "[^0-9]")
 (dayname "\\W"))
```

Emacs matches of the diary entries with the date forms is done with the standard syntax table from Fundamental mode (see section Syntax Tables), but with the `*' changed so that it is a word constituent.

The forms on the list must be *mutually exclusive* and must not match any portion of the diary entry itself, just the date. If, to be mutually exclusive, the pattern must match a portion of the diary entry itself, the first element of the form *must* be `backup`. This causes the date recognizer to back up to the beginning of the current word of the diary entry. Even if you use `backup`, the form must absolutely not match more than a portion of the first word of the diary entry. The default value of `diary-date-forms` in the European style is this list:

```
((day "/" month "[^/0-9]")
 (day "/" month "/" year "[^0-9]")
 (backup day " *" monthname "\\W+\\<[^*0-9]")
 (day " *" monthname " *" year "[^0-9]")
 (dayname "\\W"))
```

Notice the use of `backup` in the middle form because part of the diary entry must be matched to distinguish this form from the following one.

Hebrew- and Islamic-Date Diary Entries

Your diary file can have entries based on Hebrew or Islamic dates, as well as entries based on our usual Gregorian calendar. However, because the processing of such entries is time-consuming and most people don't need them, you must customize the processing of your diary file to specify that you want such entries recognized. If you want Hebrew-date diary entries, for example, you must include these lines in your `.emacs' file:

```
(setq nongregorian-diary-listing-hook 'list-hebrew-diary-entries)
(setq nongregorian-diary-marking-hook 'mark-hebrew-diary-entries)
```

If you want Islamic-date entries, include these lines in your `.emacs' file:

```
(setq nongregorian-diary-listing-hook 'list-islamic-diary-entries)
(setq nongregorian-diary-marking-hook 'mark-islamic-diary-entries)
```

If you want both Hebrew- and Islamic-date entries, include these lines:

```
(setq nongregorian-diary-listing-hook
      '(list-hebrew-diary-entries list-islamic-diary-entries))
(setq nongregorian-diary-marking-hook
      '(mark-hebrew-diary-entries mark-islamic-diary-entries))
```

Hebrew- and Islamic-date diary entries have the same formats as Gregorian-date diary entries, except that the date must be preceded with an `H' for Hebrew dates and an `I' for Islamic dates. Moreover, because the Hebrew and Islamic month names are not uniquely specified by the first three letters, you may not abbreviate them. For example, a diary entry for the Hebrew date Heshvan 25 could look like

```
HHeshvan 25 Happy Hebrew birthday!
```

and would appear in the diary for any date that corresponds to Heshvan 25 on the Hebrew calendar. Similarly, an Islamic-date diary entry might be

```
IDhu al-Qada 25 Happy Islamic birthday!
```

and would appear in the diary for any date that corresponds to Dhu al-Qada 25 on the Islamic calendar.

As with Gregorian-date diary entries, Hebrew- and Islamic-date entries are nonmarking if they are preceded with an ampersand (`&').

There are commands to help you in making Hebrew- and Islamic-date entries to your diary:

i h d

: Add a diary entry for the Hebrew date corresponding to the selected date (`insert-hebrew-diary-entry`).

i h m

: Add a diary entry for the day of the Hebrew month corresponding to the selected date (`insert-monthly-hebrew-diary-entry`).

i h y

: Add a diary entry for the day of the Hebrew year corresponding to the selected date (`insert-yearly-hebrew-diary-entry`).

i i d

: Add a diary entry for the Islamic date corresponding to the selected date (`insert-islamic-diary-entry`).

i i m

: Add a diary entry for the day of the Islamic month corresponding to the selected date (`insert-monthly-islamic-diary-entry`).

i i y

: Add a diary entry for the day of the Islamic year corresponding to the selected date (`insert-yearly-islamic-diary-entry`).

These commands work exactly like the corresponding commands for ordinary diary entries: Move point to a d in the calendar window and the above commands insert the Hebrew or Islamic date (corresponding to the date indicated by point) at the end of your diary file and you can then type the diary entry. If you want the diary entry to be nonmarking, give a numeric argument to the command.

Fancy Diary Display

Diary display works by preparing the diary buffer and then running the hook `diary-display-hook`. The default value of this hook hides the irrelevant diary entries and then displays the buffer (`simple-diary-display`). However, if you specify the hook as follows,

```
(add-hook 'diary-display-hook 'fancy-diary-display)
```

then fancy mode displays diary entries and holidays by copying them into a special buffer that exists only for display. Copying provides an opportunity to change the displayed text to make it prettier--for example, to sort the entries by the dates they apply to.

As with simple diary display, you can print a hard copy of the buffer with `print-diary-entries`. To print a hard copy of a day-by-day diary for a week by positioning point on Sunday of that week, type 7 d and then do M-x print-diary-entries. As usual, the inclusion of the holidays slows down the display slightly; you can speed things up by setting the variable `holidays-in-diary-buffer` to `nil`.

Ordinarily, the fancy diary buffer does not show days for which there are no diary entries, even if that day is a holiday. If you want such days to be shown in the fancy diary buffer, set the variable `diary-list-include-blanks` to `t`.

If you use the fancy diary display, you can use the normal hook `list-diary-entries-hook` to sort each day's diary entries by their time of day. Add this line to your `` `.emacs' `` file:

```
(add-hook 'list-diary-entries-hook 'sort-diary-entries)
```

For each day, this sorts diary entries that begin with a recognizable time of day according to their times. Diary entries without times come first within each day.

Included Diary Files

If you use the fancy diary display, you can have diary entries from other files included with your own by an "include" mechanism. This facility makes possible the sharing of common diary files among groups of users. Lines in the diary file of this form:

```
#include "filename"
```

includes the diary entries from the file filename in the fancy diary buffer (because the ordinary diary buffer is just the buffer associated with your diary file, you cannot use the include mechanism unless you use the fancy diary buffer). The include mechanism is recursive, by the way, so that included files can include other files, and so on; you must be careful not to have a cycle of inclusions, of course. To enable the include facility, add lines as follows to your `` `.emacs' `` file:

```
(add-hook 'list-diary-entries-hook 'include-other-diary-files)
```


```
(add-hook 'mark-diary-entries-hook 'mark-included-diary-files)
```

Sexp Entries and the Fancy Diary Display

Sexp diary entries allow you to do more than just have complicated conditions under which a diary entry applies. If you use the fancy diary display, sexp entries can generate the text of the entry depending on the date itself. For example, an anniversary diary entry can insert the number of years since the anniversary date into the text of the diary entry. Thus the `%d' in this dairy entry:

```
%%(diary-anniversary 10 31 1948) Arthur's birthday (%d years old)
```

gets replaced by the age, so on October 31, 1990 the entry appears in the fancy diary buffer like this:

```
Arthur's birthday (42 years old)
```

If the diary file instead contains this entry:

```
%%(diary-anniversary 10 31 1948) Arthur's %d%s birthday
```

the entry in the fancy diary buffer for October 31, 1990 appears like this:

```
Arthur's 42nd birthday
```

Similarly, cyclic diary entries can interpolate the number of repetitions that have occurred:

```
%%(diary-cyclic 50 1 1 1990) Renew medication (%d%s time)
```

looks like this:

```
Renew medication (5th time)
```

in the fancy diary display on September 8, 1990.

The generality of sexp diary entries lets you specify any diary entry that you can describe algorithmically. Suppose you get paid on the 21st of the month if it is a weekday, and to the Friday before if the 21st is on a weekend. The diary entry

```
&%%(let ((dayname (calendar-day-of-week date))
         (day (car (cdr date))))
      (or (and (= day 21) (memq dayname '(1 2 3 4 5)))
          (and (memq day '(19 20)) (= dayname 5)))
        ) Pay check deposited
```

applies to just those dates. This example illustrates how the sexp can depend on the variable `date`; this variable is a list (month day year) that gives the Gregorian date for which the diary entries are being found. If the value of the expression is `t`, the entry applies to that date. If the expression evaluates to `nil`, the entry does *not* apply to that date.

The following sexp diary entries take advantage of the ability (in the fancy diary display) to concoct diary entries based on the date:

`%%(diary-sunrise-sunset)`
: Make a diary entry for the local times of today's sunrise and sunset.

`%%(diary-phases-of-moon)`
: Make a diary entry for the phases (quarters) of the moon.

`%%(diary-day-of-year)`
: Make a diary entry with today's day number in the current year and the number of days remaining in the current year.

`%%(diary-iso-date)`
: Make a diary entry with today's equivalent ISO commercial date.

`%%(diary-julian-date)`
: Make a diary entry with today's equivalent date on the Julian calendar.

`%%(diary-astro-day-number)`
: Make a diary entry with today's equivalent astronomical (Julian) day number.

`%%(diary-hebrew-date)`
: Make a diary entry with today's equivalent date on the Hebrew calendar.

`%%(diary-islamic-date)`
: Make a diary entry with today's equivalent date on the Islamic calendar.

`%%(diary-french-date)`
: Make a diary entry with today's equivalent date on the French Revolutionary calendar.

`%%(diary-mayan-date)`
: Make a diary entry with today's equivalent date on the Mayan calendar.

Thus including the diary entry

```
&%%(diary-hebrew-date)
```

causes every day's diary display to contain the equivalent date on the Hebrew calendar, if you are using the fancy diary display. (With simple diary display, the line `&%%(diary-hebrew-date)' appears in the diary for any date, but does nothing particularly useful.)

There are a number of other available sexp diary entries that are important to those who follow the Hebrew calendar:

`%%(diary-rosh-hodesh)`
: Make a diary entry that tells the occurrence and ritual announcement of each new Hebrew month.

`%%(diary-parasha)`
: Make a Saturday diary entry that tells the weekly synagogue scripture reading.

`%%(diary-sabbath-candles)`
: Make a Friday diary entry that tells the *local time* of Sabbath candle lighting.

`%%(diary-omer)`
: Make a diary entry that gives the omer count, when appropriate.

`%%(diary-yahrzeit month day year) name`
: Make a diary entry marking the anniversary of a date of death. The date is the *Gregorian* (civil) date of death. The diary entry appears on the proper Hebrew calendar anniversary and on the day before. (In the

European style, the order of the parameters is changed to day, month, year.)

Customizing Appointment Reminders

You can specify exactly how Emacs reminds you of an appointment and how far in advance it begins doing so. Here are the variables that you can set:

`appt-message-warning-time`
: The time in minutes before an appointment that the reminder begins. The default is 10 minutes.

`appt-audible`
: If this is `t` (the default), Emacs rings the terminal bell for appointment reminders.

`appt-visible`
: If this is `t` (the default), Emacs displays the appointment message in echo area.

`appt-display-mode-line`
: If this is `t` (the default), Emacs displays the number of minutes to the appointment on the mode line.

`appt-msg-window`
: If this is `t` (the default), Emacs displays the appointment message in another window.

`appt-display-duration`
: The number of seconds an appointment message is displayed. The default is 5 seconds.

Go to the previous, next section.

Go to the previous, next section.

Tips and Standards

This chapter describes no additional features of Emacs Lisp. Instead it gives advice on making effective use of the features described in the previous chapters.

Writing Clean Lisp Programs

Here are some tips for avoiding common errors in writing Lisp code intended for widespread use:

- Since all global variables share the same name space, and all functions share another name space, you should choose a short word to distinguish your program from other Lisp programs. Then take care to begin the names of all global variables, constants, and functions with the chosen prefix. This helps avoid name conflicts.

  This recommendation applies even to names for traditional Lisp primitives that are not primitives in Emacs Lisp--even to `cadr`. Believe it or not, there is more than one plausible way to define `cadr`. Play it safe; append your name prefix to produce a name like `foo-cadr` or `mylib-cadr` instead.

  If one prefix is insufficient, your package may use two or three alternative common prefixes, so long as they make sense.

  Separate the prefix from the rest of the symbol name with a hyphen, `-'. This will be consistent with Emacs itself and with most Emacs Lisp programs.
- It is often useful to put a call to `provide` in each separate library program, at least if there is more than one entry point to the program.
- If one file foo uses a macro defined in another file bar, foo should contain `(require 'bar)` before the first use of the macro. (And bar should contain `(provide 'bar)`, to make the `require` work.) This will cause bar to be loaded when you byte-compile foo. Otherwise, you risk compiling foo without the necessary macro loaded, and that would produce compiled code that won't work right. See section Macros and Byte Compilation.
- If you define a major mode, make sure to run a hook variable using `run-hooks`, just as the existing major modes do. See section Hooks.
- Please do not define C-c letter as a key in your major modes. These sequences are reserved for users; they are the **only** sequences reserved for users, so we cannot do without them.

  Instead, define sequences consisting of C-c followed by a non-letter. These sequences are reserved for major modes.

  Changing all the major modes in Emacs 18 so they would follow this convention was a lot of work. Abandoning this convention would waste that work and inconvenience the users.
- It is a bad idea to define aliases for the Emacs primitives. Use the standard names instead.

- Redefining an Emacs primitive is an even worse idea. It may do the right thing for a particular program, but there is no telling what other programs might break as a result.
- If a file does replace any of the functions or library programs of standard Emacs, prominent comments at the beginning of the file should say which functions are replaced, and how the behavior of the replacements differs from that of the originals.
- If a file requires certain standard library programs to be loaded beforehand, then the comments at the beginning of the file should say so.
- Please keep the names of your Emacs Lisp source files to 13 characters or less. This way, if the files are compiled, the compiled files' names will be 14 characters or less, which is short enough to fit on all kinds of Unix systems.
- Don't use `next-line` or `previous-line` in programs; nearly always, `forward-line` is more convenient as well as more predictable and robust. See section Motion by Text Lines.
- Don't use functions that set the mark in your Lisp code (unless you are writing a command to set the mark). The mark is a user-level feature, so it is incorrect to change the mark except to supply a value for the user's benefit. See section The Mark.

  In particular, don't use these functions:
  - `beginning-of-buffer`, `end-of-buffer`
  - `replace-string`, `replace-regexp`

  If you just want to move point, or replace a certain string, without any of the other features intended for interactive users, you can replace these functions with one or two lines of simple Lisp code.
- The recommended way to print a message in the echo area is with the `message` function, not `princ`. See section The Echo Area.
- When you encounter an error condition, call the function `error` (or `signal`). The function `error` does not return. See section How to Signal an Error.

  Do not use `message`, `throw`, `sleep-for`, or `beep` to report errors.
- Avoid using recursive edits. Instead, do what the Rmail w command does: use a new local keymap that contains one command defined to switch back to the old local keymap. Or do what the `edit-options` command does: switch to another buffer and let the user switch back at will. See section Recursive Editing.
- In some other systems there is a convention of choosing variable names that begin and end with `*'. We don't use that convention in Emacs Lisp, so please don't use it in your library. (In fact, in Emacs names of this form are conventionally used for program-generated buffers.) The users will find Emacs more coherent if all libraries use the same conventions.
- Indent each function with C-M-q (`indent-sexp`) using the default indentation parameters.
- Don't make a habit of putting close-parentheses on lines by themselves; Lisp programmers find this disconcerting. Once in a while, when there is a sequence of many consecutive close-parentheses, it may make sense to split them in one or two significant places.
- Please put a copyright notice on the file if you give copies to anyone. Use the same lines that

appear at the top of the Lisp files in Emacs itself. If you have not signed papers to assign the copyright to the Foundation, then place your name in the copyright notice in place of the Foundation's name.

Tips for Making Compiled Code Fast

Here are ways of improving the execution speed of byte-compiled lisp programs.

- Use the `profile' library to profile your program. See the file `profile.el' for instructions.
- Use iteration rather than recursion whenever possible. Function calls are slow in Emacs Lisp even when a compiled function is calling another compiled function.
- Using the primitive list-searching functions `memq`, `assq` or `assoc` is even faster than explicit iteration. It may be worth rearranging a data structure so that one of these primitive search functions can be used.
- Certain built-in functions are handled specially by the byte compiler avoiding the need for an ordinary function call. It is a good idea to use these functions rather than alternatives. To see whether a function is handled specially by the compiler, examine its `byte-compile` property. If the property is non-`nil`, then the function is handled specially.

  For example, the following input will show you that `aref` is compiled specially (see section Functions that Operate on Arrays) while `elt` is not (see section Sequences):

```
(get 'aref 'byte-compile)
     => byte-compile-two-args

(get 'elt 'byte-compile)
     => nil
```

- Make small functions inline, so that calls to them in compiled code run faster. See section Inline Functions.

Tips for Documentation Strings

Here are some tips for the writing of documentation strings.

- Every command, function or variable intended for users to know about should have a documentation string.
- An internal subroutine of a Lisp program need not have a documentation string, and you can save space by using a comment instead.
- The first line of the documentation string should consist of one or two complete sentences which stand on their own as a summary. In particular, start the line with a capital letter and end with a period.

  The documentation string can have additional lines which expand on the details of how to use the

function or variable. The additional lines should be made up of complete sentences also, but th may be filled if that looks good.

- Do not start or end a documentation string with whitespace.
- Format the documentation string so that it fits in an Emacs window on an 80 column screen. It is a good idea for most lines to be no wider than 60 characters. The first line can be wider if necessary to fit the information that ought to be there.

  However, rather than simply filling the entire documentation string, you can make it much more readable by choosing line breaks with care. Use blank lines between topics if the documentation string is long.
- **Do not** indent subsequent lines of a documentation string so that the text is lined up in the source code with the text of the first line. This looks nice in the source code, but looks bizarre when users view the documentation. Remember that the indentation before the starting double-quote is not part of the string!
- A variable's documentation string should start with `*' if the variable is one that users would want to set interactively often. If the value is a long list, or a function, or if the variable would only be set in init files, then don't start the documentation string with `*'. See section Defining Global Variables.
- The documentation string for a variable that is a yes-or-no flag should start with words such as "Non-nil means...", to make it clear both that the variable only has two meaningfully distinct values and which value means "yes".
- When a function's documentation string mentions the value of an argument of the function, use the argument name in capital letters as if it were a name for that value. Thus, the documentation string of the function `/` refers to its second argument as `DIVISOR'.

  Also use all caps for meta-syntactic variables, such as when you show the decomposition of a list or vector into subunits, some of which may be variable.
- When a documentation string refers to a Lisp symbol, write it as it would be printed (which usually means in lower case), with single-quotes around it. For example: ``lambda''. There are two exceptions: write `t` and `nil` without single-quotes.
- Don't write key sequences directly in documentation strings. Instead, use the `\\[...]' construct to stand for them. For example, instead of writing `C-f', write `\\[forward-char]'. When the documentation string is printed, Emacs will substitute whatever key is currently bound to `forward-char`. This will usually be `C-f', but if the user has moved key bindings, it will be the correct key for that user. See section Substituting Key Bindings in Documentation.
- In documentation strings for a major mode, you will want to refer to the key bindings of that mode's local map, rather than global ones. Therefore, use the construct `\\<...>' once in the documentation string to specify which key map to use. Do this before the first use of `\\[...]'. The text inside the `\\<...>' should be the name of the variable containing the local keymap for the major mode.

  It is not practical to use `\\[...]' very many times, because display of the documentation string will become slow. So use this to describe the most important commands in your major mode, and then use `\\{...}' to display the rest of the mode's keymap.

- Don't use the term "Elisp", since that is or was a trademark. Use the term "Emacs Lisp".

Tips on Writing Comments

We recommend these conventions for where to put comments and how to indent them:

`;'

Comments that start with a single semicolon, `;', should all be aligned to the same column on the right of the source code. Such comments usually explain how the code on the same line does its job. In Lisp mode and related modes, the M-; (`indent-for-comment`) command automatically inserts such a `;' in the right place, or aligns such a comment if it is already inserted.

(The following examples are taken from the Emacs sources.)

```
(setq base-version-list                  ; there was a base
      (assoc (substring fn 0 start-vn)   ; version to which
             file-version-assoc-list))   ; this looks like
                                         ; a subversion
```

`;;'

Comments that start with two semicolons, `;;', should be aligned to the same level of indentation as the code. Such comments are used to describe the purpose of the following lines or the state of the program at that point. For example:

```
(prog1 (setq auto-fill-function
             ...
             ...
  ;; update mode-line
  (force-mode-line-update)))
```

These comments are also written before a function definition to explain what the function does and how to call it properly.

`;;;'

Comments that start with three semicolons, `;;;', should start at the left margin. Such comments are not used within function definitions, but are used to make more general comments. For example:

```
;;; This Lisp code is run in Emacs
;;; when it is to operate asa server
;;; for other processes.
```

`;;;;'

Comments that start with four semicolons, `;;;;', should be aligned to the left margin and are used for headings of major sections of a program. For example:

```
;;;; The kill ring
```

The indentation commands of the Lisp modes in Emacs, such as M-; (`indent-for-comment`) and TAB (`lisp-indent-line`) automatically indent comments according to these conventions, depending on the the number of semicolons. See section 'Manipulating Comments' in The GNU Emacs Manual.

If you wish to "comment out" a number of lines of code, use triple semicolons at the beginnings of the lines.

Any character may be included in a comment, but it is advisable to precede a character with syntactic significance in Lisp (such as `\' or unpaired `(' or `)') with a `\', to prevent it from confusing the Emacs commands for editing Lisp.

Conventional Headers for Emacs Libraries

Emacs 19 has conventions for using special comments in Lisp libraries to divide them into sections and give information such as who wrote them. This section explains these conventions. First, an example:

```
;;; lisp-mnt.el -- minor mode for Emacs Lisp maintainers

;; Copyright (C) 1992 Free Software Foundation, Inc.

;; Author: Eric S. Raymond <esr@snark.thyrsus.com>
;; Maintainer: Eric S. Raymond <esr@snark.thyrsus.com>
;; Created: 14 Jul 1992
;; Version: 1.2
;; Keywords: docs

;; This file is part of GNU Emacs.
copying conditions...
```

The very first line should have this format:

```
;;; filename -- description
```

The description should be complete in one line.

After the copyright notice come several header comment lines, each beginning with `;;; header-name:'. Here is a table of the conventional possibilities for header-name:

`Author'

This line states the name and net address of at least the principal author of the library.

If there are multiple authors, you can list them on continuation lines led by `;;<TAB>`, like this:

```
;; Author: Ashwin Ram <Ram-Ashwin@cs.yale.edu>
;;      Dave Sill <de5@ornl.gov>
;;      Dave Brennan <brennan@hal.com>
```


```
;;      Eric Raymond <esr@snark.thyrsus.com>
```

`Maintainer'

This line should contain a single name/address as in the Author line, or an address only, or the string "FSF". If there is no maintainer line, the person(s) in the Author field are presumed to be the maintainers. The example above is mildly bogus because the maintainer line is redundant.

The idea behind the `Author' and `Maintainer' lines is to make possible a Lisp function to "send mail to the maintainer" without having to mine the name out by hand.

Be sure to surround the network address with `<...>' if you include the person's full name as well as the network address.

`Created'

This optional line gives the original creation date of the file. For historical interest only.

`Version'

If you wish to record version numbers for the individual Lisp program, put them in this line.

`Adapted-By'

In this header line, place the name of the person who adapted the library for installation (to make it fit the style conventions, for example.

`Keywords'

This line lists keywords for the `finder-by-keyword` help command. This field is important; it's how people will find your package when they're looking for things by topic area.

Just about every Lisp library ought to have the `Author' and `Keywords' header comment lines. Use the others if they are appropriate. You can also put in header lines with other header names--they have no standard meanings, so they can't do any harm.

We use additional stylized comments to subdivide the contents of the library file. Here is a table of them:

`;;; Commentary:'

This begins introductory comments that explain how the library works. It should come right after the copying permissions.

`;;; Change log:'

This begins change log information stored in the library file (if you store the change history there). For most of the Lisp files distributed with Emacs, the change history is kept in the file `` `ChangeLog' `` and not in the source file at all; these files do not have a `;;; Change log:' line.

`;;; Code:'

This begins the actual code of the program.

`;;; filename ends here'

This is the footer line; it appears at the very end of the file. Its purpose is to enable people to detect truncated versions of the file from the lack of a footer line.

Go to the previous, next section.

Go to the previous, next section.

GNU Emacs Internals

This chapter describes how the runnable Emacs executable is dumped with the preloaded Lisp libraries in it, how storage is allocated, and some internal aspects of GNU Emacs that may be of interest to C programmers.

Building Emacs

The first step in building Emacs is to compile the C sources. This produces a program called `temacs', also called a bare impure Emacs. It contains the Emacs Lisp interpreter and I/O routines, but not the editing commands.

Then, to create a working Emacs editor, issue the `temacs -l loadup' command. This directs `temacs' to evaluate the Lisp files specified in the file `loadup.el'. These files set up the normal Emacs editing environment, resulting in an Emacs which is still impure but no longer bare.

It takes a long time to load the standard Lisp files. Luckily, you don't have to do this each time you run Emacs; `temacs' can dump out an executable program called `emacs' which has these files preloaded. `emacs' starts more quickly because it does not need to load the files. This is the program that is normally installed.

To create `emacs', use the command `temacs -batch -l loadup dump'. The purpose of `-batch' here is to prevent `temacs' from trying to initialize any of its data on the terminal; this ensures that the tables of terminal information are empty in the dumped Emacs.

When the `emacs' executable is started, it automatically loads the user's `.emacs' file, or the default initialization file `default.el' if the user has none. (See section Starting Up Emacs.) With the `.emacs' file, you can produce a version of Emacs that suits you and is not the same as the version other people use. With `default.el', you can customize Emacs for all the users at your site who don't choose to customize it for themselves. (For further reflection: why is this different from the case of the barber who shaves every man who doesn't shave himself?)

On some systems, dumping does not work. Then, you must start Emacs with the `temacs -l loadup' command each time you use it. This takes a long time, but since you need to start Emacs once a day at most--and once a week or less frequently if you never log out--the extra time is not too severe a problem.

Before `emacs' is dumped, the documentation strings for primitive and preloaded functions (and variables) need to be found in the file where they are stored. This is done by calling `Snarf-documentation` (see section Access to Documentation Strings). These strings were moved out of `emacs' to make it smaller. See section Documentation Basics.

Function: **dump-emacs** *to-file from-file*

This function dumps the current state of Emacs into an executable file to-file. It takes symbols from from-file (this is normally the executable file `temacs').

If you use this function in an Emacs that was already dumped, you must set `command-line-processed`

to `nil` first for good results. See section Command Line Arguments.

Command: **emacs-version**

This function returns a string describing the version of Emacs that is running. It is useful to include this string in bug reports.

```
(emacs-version)
  => "GNU Emacs 18.36.1 of Fri Feb 27 1987 on slug
     (berkeley-unix)"
```

Called interactively, the function prints the same information in the echo area.

Variable: **emacs-build-time**

The value of this variable is the time at which Emacs was built at the local site.

```
emacs-build-time
     => "Fri Feb 27 14:55:57 1987"
```

Variable: **emacs-version**

The value of this variable is the version of Emacs being run. It is a string, e.g. `"18.36.1"`.

Pure Storage

There are two types of storage in GNU Emacs Lisp for user-created Lisp objects: normal storage and pure storage. Normal storage is where all the new data which is created during an Emacs session is kept; see the following section for information on normal storage. Pure storage is used for certain data in the preloaded standard Lisp files: data that should never change during actual use of Emacs.

Pure storage is allocated only while `` `temacs' `` is loading the standard preloaded Lisp libraries. In the file `` `emacs' ``, it is marked as read-only (on operating systems which permit this), so that the memory space can be shared by all the Emacs jobs running on the machine at once. Pure storage is not expandable; a fixed amount is allocated when Emacs is compiled, and if that is not sufficient for the preloaded libraries, `` `temacs' `` crashes. If that happens, you will have to increase the compilation parameter `PURESIZE` in the file `` `config.h' ``. This normally won't happen unless you try to preload additional libraries or add features to the standard ones.

Function: **purecopy** *object*

This function makes a copy of object in pure storage and returns it. It copies strings by simply making a new string with the same characters in pure storage. It recursively copies the contents of vectors and cons cells. It does not make copies of symbols, or any other objects, but just returns them unchanged. It signals an error if asked to copy markers.

This function is used only while Emacs is being built and dumped; it is called only in the file `` `emacs/lisp/loaddefs.el' ``.

Variable: **pure-bytes-used**

The value of this variable is the number of bytes of pure storage allocated so far. Typically, in a dumped Emacs, this number is very close to the total amount of pure storage available--if it were not, we would preallocate less.

Variable: **purify-flag**

This variable determines whether `defun` should make a copy of the function definition in pure storage. If it is non-`nil`, then the function definition is copied into pure storage.

This flag is `t` while loading all of the basic functions for building Emacs initially (allowing those functions to be sharable and non-collectible). It is set to `nil` when Emacs is saved out as `` `emacs' ``. The flag is set and reset in the C sources.

You should not change this flag in a running Emacs.

Garbage Collection

When a program creates a list or the user defines a new function (such as by loading a library), then that data is placed in normal storage. If normal storage runs low, then Emacs asks the operating system to allocate more memory in blocks of 1k bytes. Each block is used for one type of Lisp object, so symbols, cons cells, markers, etc. are segregated in distinct blocks in memory. (Vectors, buffers and certain other editing types, which are fairly large, are allocated in individual blocks, one per object, while strings are packed into blocks of 8k bytes.)

It is quite common to use some storage for a while, then release it by, for example, killing a buffer or deleting the last pointer to an object. Emacs provides a garbage collector to reclaim this abandoned storage. (This name is traditional, but "garbage recycler" might be a more intuitive metaphor for this facility.)

The garbage collector operates by scanning all the objects that have been allocated and marking those that are still accessible to Lisp programs. To begin with, all the symbols, their values and associated function definitions, and any data presently on the stack, are accessible. Any objects which can be reached indirectly through other accessible objects are also accessible.

When this is finished, all inaccessible objects are garbage. No matter what the Lisp program or the user does, it is impossible to refer to them, since there is no longer a way to reach them. Their space might as well be reused, since no one will notice. That is what the garbage collector arranges to do.

Unused cons cells are chained together onto a free list for future allocation; likewise for symbols and markers. The accessible strings are compacted so they are contiguous in memory; then the rest of the space formerly occupied by strings is made available to the string creation functions. Vectors, buffers, windows and other large objects are individually allocated and freed using `malloc`.

> **Common Lisp note:** unlike other Lisps, GNU Emacs Lisp does not call the garbage collector when the free list is empty. Instead, it simply requests the operating system to allocate more storage, and processing continues until `gc-cons-threshold` bytes have been used.
>
> This means that you can make sure that the garbage collector will not run during a certain portion of a Lisp program by calling the garbage collector explicitly just before it (provided that portion of the program does not use so much space as to force a second garbage collection).

Command: **garbage-collect**

This command runs a garbage collection, and returns information on the amount of space in use. (Garbage collection can also occur spontaneously if you use more than `gc-cons-threshold` bytes of Lisp data since the previous garbage collection.)

`garbage-collect` returns a list containing the following information:

```
((used-conses . free-conses)
 (used-syms . free-syms)
 (used-markers . free-markers)
 used-string-chars
 used-vector-slots
 (used-floats . free-floats))

(garbage-collect)
     => ((3435 . 2332) (1688 . 0) (57 . 417) 24510 3839 (4 . 1))
```

Here is a table explaining each element:

used-conses
: The number of cons cells in use.

free-conses
: The number of cons cells for which space has been obtained from the operating system, but that are not currently being used.

used-syms
: The number of symbols in use.

free-syms
: The number of symbols for which space has been obtained from the operating system, but that are not currently being used.

used-markers
: The number of markers in use.

free-markers
: The number of markers for which space has been obtained from the operating system, but that are not currently being used.

used-string-chars
: The total size of all strings, in characters.

used-vector-slots
: The total number of elements of existing vectors.

used-floats
: The number of floats in use.

free-floats
: The number of floats for which space has been obtained from the operating system, but that are not currently being used.

User Option: **gc-cons-threshold**

The value of this variable is the number of bytes of storage that must be allocated for Lisp objects after c garbage collection in order to request another garbage collection. A cons cell counts as eight bytes, a string as one byte per character plus a few bytes of overhead, and so on. (Space allocated to the contents of buffers does not count.) Note that the new garbage collection does not happen immediately when the threshold is exhausted, but only the next time the Lisp evaluator is called.

The initial threshold value is 100,000. If you specify a larger value, garbage collection will happen less often. This reduces the amount of time spent garbage collecting, but increases total memory use. You may want to do this when running a program which creates lots of Lisp data.

You can make collections more frequent by specifying a smaller value, down to 10,000. A value less than 10,000 will remain in effect only until the subsequent garbage collection, at which time `garbage-collect` will set the threshold back to 10,000.

Function: **memory-limit**

This function returns the address of the last byte Emacs has allocated, divided by 1024. We divide the value by 1024 to make sure it fits in a Lisp integer.

You can use this to get a general idea of how your actions affect the memory usage.

Writing Emacs Primitives

Lisp primitives are Lisp functions implemented in C. The details of interfacing the C function so that Lisp can call it are handled by a few C macros. The only way to really understand how to write new C code is to read the source, but we can explain some things here.

An example of a special form is the definition of `or`, from `` `eval.c' ``. (An ordinary function would have the same general appearance.)

```
DEFUN ("or", For, Sor, 0, UNEVALLED, 0,
  "Eval args until one of them yields non-NIL, then return that value.\n\
The remaining args are not evalled at all.\n\
If all args return NIL, return NIL.")
  (args)
     Lisp_Object args;
{
  register Lisp_Object val;
  Lisp_Object args_left;
  struct gcpro gcpro1;

  if (NULL(args))
    return Qnil;

  args_left = args;
  GCPRO1 (args_left);

  do
```


```
    {
      val = Feval (Fcar (args_left));
      if (!NULL (val))
        break;
      args_left = Fcdr (args_left);
    }
  while (!NULL(args_left));

  UNGCPRO;
  return val;
}
```

Let's start with a precise explanation of the arguments to the `DEFUN` macro. Here are the general names for them:

```
DEFUN (lname, fname, sname, min, max, interactive, doc)
```

lname

This is the name of the Lisp symbol to define with this function; in the example above, it is `or`.

fname

This is the C function name for this function. This is the name that is used in C code for calling the function. The name is, by convention, `F' prepended to the Lisp name, with all dashes (`-') in the Lisp name changed to underscores. Thus, to call this function from C code, call `For`. Remember that the arguments must be of type `Lisp_Object`; various macros and functions for creating values of type `Lisp_Object` are declared in the file `` `lisp.h' ``.

sname

This is a C variable name to use for a structure that holds the data for the subr object that represents the function in Lisp. This structure conveys the Lisp symbol name to the initialization routine that will create the symbol and store the subr object as its definition. By convention, this name is always fname with `F' replaced with `S'.

min

This is the minimum number of arguments that the function requires. For `or`, no arguments are required.

max

This is the maximum number of arguments that the function accepts. Alternatively, it can be `UNEVALLED`, indicating a special form that receives unevaluated arguments. A function with the equivalent of an `&rest` argument would have `MANY` in this position. Both `UNEVALLED` and `MANY` are macros. This argument must be one of these macros or a number at least as large as min. It may not be greater than six.

interactive

This is an interactive specification, a string such as might be used as the argument of `interactive` in a Lisp function. In the case of `or`, it is 0 (a null pointer), indicating that `or` cannot be called interactively. A value of `""` indicates an interactive function taking no arguments.

doc

This is the documentation string. It is written just like a documentation string for a function defined in

Lisp, except you must write `\n\' at the end of each line. In particular, the first line should be a sin sentence.

After the call to the `DEFUN` macro, you must write the list of argument names that every C function must have, followed by ordinary C declarations for them. Normally, all the arguments must be declared as `Lisp_Object`. If the function has no upper limit on the number of arguments in Lisp, then in C it receives two arguments: the number of Lisp arguments, and the address of a block containing their values. These have types `int` and `Lisp_Object *`.

Within the function `For` itself, note the use of the macros `GCPRO1` and `UNGCPRO`. `GCPRO1` is used to "protect" a variable from garbage collection--to inform the garbage collector that it must look in that variable and regard its contents as an accessible object. This is necessary whenever you call `Feval` or anything that can directly or indirectly call `Feval`. At such a time, any Lisp object that you intend to refer to again must be protected somehow. `UNGCPRO` cancels the protection of the variables that are protected in the current function. It is necessary to do this explicitly.

For most data types, it suffices to know that one pointer to the object is protected; as long as the object is not recycled, all pointers to it remain valid. This is not so for strings, because the garbage collector can move them. When a string is moved, any pointers to it that the garbage collector does not know about will not be properly relocated. Therefore, all pointers to strings must be protected across any point where garbage collection may be possible.

The macro `GCPRO1` protects just one local variable. If you want to protect two, use `GCPRO2` instead; repeating `GCPRO1` will not work. There are also `GCPRO3` and `GCPRO4`.

In addition to using these macros, you must declare the local variables such as `gcpro1` which they implicitly use. If you protect two variables, with `GCPRO2`, you must declare `gcpro1` and `gcpro2`, as it uses them both. Alas, we can't explain all the tricky details here.

Defining the C function is not enough; you must also create the Lisp symbol for the primitive and store a suitable subr object in its function cell. This is done by adding code to an initialization routine. The code looks like this:

```
defsubr (&subr-structure-name);
```

subr-structure-name is the name you used as the third argument to `DEFUN`.

If you are adding a primitive to a file that already has Lisp primitives defined in it, find the function (near the end of the file) named `syms_of_something`, and add that function call to it. If the file doesn't have this function, or if you create a new file, add to it a `syms_of_filename` (e.g., `syms_of_myfile`). Then find the spot in `` `emacs.c' `` where all of these functions are called, and add a call to `syms_of_filename` there.

This function `syms_of_filename` is also the place to define any C variables which are to be visible as Lisp variables. `DEFVAR_LISP` is used to make a C variable of type `Lisp_Object` visible in Lisp. `DEFVAR_INT` is used to make a C variable of type `int` visible in Lisp with a value that is an integer.

Here is another function, with more complicated arguments. This comes from the code for the X Window System, and it demonstrates the use of macros and functions to manipulate Lisp objects.


```
DEFUN ("coordinates-in-window-p", Fcoordinates_in_window_p,
  Scoordinates_in_window_p, 2, 2,
  "xSpecify coordinate pair: \nXExpression which evals to window: ",
  "Return non-nil if POSITIONS is in WINDOW.\n\
  \(POSITIONS is a list, (SCREEN-X SCREEN-Y)\)\n\
  Returned value is list of positions expressed\n\
  relative to window upper left corner.")
  (coordinate, window)
     register Lisp_Object coordinate, window;
{
  register Lisp_Object xcoord, ycoord;

  if (!CONSP (coordinate)) wrong_type_argument (Qlistp, coordinate);
  CHECK_WINDOW (window, 2);
  xcoord = Fcar (coordinate);
  ycoord = Fcar (Fcdr (coordinate));
  CHECK_NUMBER (xcoord, 0);
  CHECK_NUMBER (ycoord, 1);
  if ((XINT (xcoord) < XINT (XWINDOW (window)->left))
      || (XINT (xcoord) >= (XINT (XWINDOW (window)->left)
                            + XINT (XWINDOW (window)->width))))
    {
      return Qnil;
    }
  XFASTINT (xcoord) -= XFASTINT (XWINDOW (window)->left);
  if (XINT (ycoord) == (screen_height - 1))
    return Qnil;
  if ((XINT (ycoord) < XINT (XWINDOW (window)->top))
      || (XINT (ycoord) >= (XINT (XWINDOW (window)->top)
                            + XINT (XWINDOW (window)->height)) - 1))
    {
      return Qnil;
    }
  XFASTINT (ycoord) -= XFASTINT (XWINDOW (window)->top);
  return (Fcons (xcoord, Fcons (ycoord, Qnil)));
}
```

Note that you cannot directly call functions defined in Lisp as, for example, the primitive function `Fcons` is called above. You must create the appropriate Lisp form, protect everything from garbage collection, and `Feval` the form, as was done in `For` above.

`` `eval.c' `` is a very good file to look through for examples; `` `lisp.h' `` contains the definitions for some important macros and functions.

Object Internals

GNU Emacs Lisp manipulates many different types of data. The actual data are stored in a heap and the only access that programs have to it is through pointers. Pointers are thirty-two bits wide in most implementations. Depending on the operating system and type of machine for which you compile Emacs, twenty-four to twenty-six bits are used to address the object, and the remaining six to eight bits are used for a tag that identifies the object's type.

Because all access to data is through tagged pointers, it is always possible to determine the type of any object. This allows variables to be untyped, and the values assigned to them to be changed without regard to type. Function arguments also can be of any type; if you want a function to accept only a certain type of argument, you must check the type explicitly using a suitable predicate (see section Type Predicates).

Buffer Internals

Buffers contain fields not directly accessible by the Lisp programmer. We describe them here, naming them by the names used in the C code. Many are accessible indirectly in Lisp programs via Lisp primitives.

`name`
: The buffer name is a string which names the buffer. It is guaranteed to be unique. See section Buffer Names.

`save_modified`
: This field contains the time when the buffer was last saved, as an integer. See section Buffer Modification.

`modtime`
: This field contains the modification time of the visited file. It is set when the file is written or read. Every time the buffer is written to the file, this field is compared to the modification time of the file. See section Buffer Modification.

`auto_save_modified`
: This field contains the time when the buffer was last auto-saved.

`last_window_start`
: This field contains the `window-start` position in the buffer as of the last time the buffer was displayed in a window.

`undodata`
: This field points to the buffer's undo stack. See section Undo.

`syntax_table_v`
: This field contains the syntax table for the buffer. See section Syntax Tables.

`downcase_table`
: This field contains the conversion table for converting text to lower case. See section The Case Table.

`upcase_table`
: This field contains the conversion table for converting text to upper case. See section The Case Table.

`case_canon_table`

This field contains the conversion table for canonicalizing text for case-folding search. See section The Case Table.

`case_eqv_table`

This field contains the equivalence table for case-folding search. See section The Case Table.

`display_table`

This field contains the buffer's display table, or `nil` if it doesn't have one. See section Display Tables.

`markers`

This field contains the chain of all markers that point into the buffer. At each deletion or motion of the buffer gap, all of these markers must be checked and perhaps updated. See section Markers.

`backed_up`

This field is a flag which tells whether a backup file has been made for the visited file of this buffer.

`mark`

This field contains the mark for the buffer. The mark is a marker, hence it is also included on the list `markers`. See section The Mark.

`local_var_alist`

This field contains the association list containing all of the variables local in this buffer, and their values. The function `buffer-local-variables` returns a copy of this list. See section Buffer-Local Variables.

`mode_line_format`

This field contains a Lisp object which controls how to display the mode line for this buffer. See section Mode Line Format.

Window Internals

Windows have the following accessible fields:

`frame`

The frame that this window is on.

`mini_p`

Non-`nil` if this window is a minibuffer window.

`height`

The height of the window, measured in lines.

`width`

The width of the window, measured in columns.

`buffer`

The buffer which the window is displaying. This may change often during the life of the window.

`dedicated`

Non-`nil` if this window is dedicated to its buffer.

`start`

The position in the buffer which is the first character to be displayed in the window.

`pointm`

This is the value of point in the current buffer when this window is selected; when it is not selected, it retains its previous value.

`left`

This is the left-hand edge of the window, measured in columns. (The leftmost column on the screen is column 0.)

`top`

This is the top edge of the window, measured in lines. (The top line on the screen is line 0.)

`next`

This is the window that is the next in the chain of siblings.

`prev`

This is the window that is the previous in the chain of siblings.

`force_start`

This is a flag which, if non-`nil`, says that the window has been scrolled explicitly by the Lisp program. At the next redisplay, if point is off the screen, instead of scrolling the window to show the text around point, point will be moved to a location that is on the screen.

`hscroll`

This is the number of columns that the display in the window is scrolled horizontally to the left. Normally, this is 0.

`use_time`

This is the last time that the window was selected. The function `get-lru-window` uses this field.

`display_table`

The window's display table, or `nil` if none is specified for it.

Process Internals

The fields of a process are:

`name`

A string, the name of the process.

`command`

A list containing the command arguments that were used to start this process.

`filter`

A function used to accept output from the process instead of a buffer, or `nil`.

`sentinel`

A function called whenever the process receives a signal, or `nil`.

`buffer`

The associated buffer of the process.

`pid`

An integer, the Unix process ID.

`childp`

A flag, non-`nil` if this is really a child process. It is `nil` for a network connection.

`flags`

A symbol indicating the state of the process. Possible values include `run`, `stop`, `closed`, etc.

`reason`

An integer, the Unix signal number that the process received that caused the process to terminate or stop. If the process has exited, then this is the exit code it specified.

`mark`

A marker indicating the position of end of last output from this process inserted into the buffer. This is usually the end of the buffer.

`kill_without_query`

A flag, non-`nil` meaning this process should not cause confirmation to be needed if Emacs is killed.

Go to the previous, next section.

Go to the previous, next section.

Standard Errors

Here is the complete list of the error symbols in standard Emacs, grouped by concept. The list includes each symbol's message (on the `error-message` property of the symbol), and a cross reference to a description of how the error can occur.

Each error symbol has an `error-conditions` property which is a list of symbols. Normally, this list includes the error symbol itself, and the symbol `error`. Occasionally it includes additional symbols, which are intermediate classifications, narrower than `error` but broader than a single error symbol. For example, all the errors in accessing files have the condition `file-error`.

As a special exception, the error symbol `quit` does not have the condition `error`, because quitting is not considered an error.

See section Errors, for an explanation of how errors are generated and handled.

`symbol`
: string; reference.

`error`
: `"error"`
See section Errors.

`quit`
: `"Quit"`
See section Quitting.

`args-out-of-range`
: `"Args out of range"`
See section Sequences, Arrays, and Vectors.

`arith-error`
: `"Arithmetic error"`
See `/` and `%` in section Numbers.

`beginning-of-buffer`
: `"Beginning of buffer"`
See section Motion.

`buffer-read-only`
: `"Buffer is read-only"`
See section Read-Only Buffers.

`end-of-buffer`
: `"End of buffer"`
See section Motion.

`end-of-file`

> `"End of file during parsing"`
> This is not a `file-error`.
> See section Input Functions.

`file-error`

> This error, and its subcategories, do not have error-strings, because the error message is constructed from the data items alone when the error condition `file-error` is present.
> See section Files.

`file-locked`

> This is a `file-error`.
> See section File Locks.

`file-already-exists`

> This is a `file-error`.
> See section Writing to Files.

`file-supersession`

> This is a `file-error`.
> See section Buffer Modification.

`invalid-function`

> `"Invalid function"`
> See section Classification of List Forms.

`invalid-read-syntax`

> `"Invalid read syntax"`
> See section Input Functions.

`invalid-regexp`

> `"Invalid regexp"`
> See section Regular Expressions.

`no-catch`

> `"No catch for tag"`
> See section Explicit Nonlocal Exits: `catch` and `throw`.

`search-failed`

> `"Search failed"`
> See section Searching and Matching.

`setting-constant`

> `"Attempt to set a constant symbol"`
> The values of the symbols `nil` and `t` may not be changed.
> See section Variables that Never Change.

`void-function`

> `"Symbol's function definition is void"`

See section Accessing Function Cell Contents.

`void-variable`

`"Symbol's value as variable is void"`
See section Accessing Variable Values.

`wrong-number-of-arguments`

`"Wrong number of arguments"`
See section Classification of List Forms.

`wrong-type-argument`

`"Wrong type argument"`
See section Type Predicates.

Go to the previous, next section.

Go to the previous, next section.

Buffer-Local Variables

The table below shows all of the variables that are automatically local (when set) in each buffer in Emacs Version 18 with the common packages loaded.

`abbrev-mode`

see section Abbrevs And Abbrev Expansion

`auto-fill-function`

see section Auto Filling

`buffer-auto-save-file-name`

see section Auto-Saving

`buffer-backed-up`

see section Backup Files

`buffer-display-table`

see section Display Tables

`buffer-file-name`

see section Buffer File Name

`buffer-file-number`

see section Buffer File Name

`buffer-file-truename`

see section Buffer File Name

`buffer-offer-save`

see section Saving Buffers

`buffer-read-only`

see section Read-Only Buffers

`buffer-saved-size`

see section Point

`buffer-undo-list`

see section Undo

`case-fold-search`

see section Searching and Case

`ctl-arrow`

see section Usual Display Conventions

`default-directory`

see section Operating System Environment

`fill-column`

see section Auto Filling

`left-margin`

see section Indentation

`local-abbrev-table`

see section Abbrevs And Abbrev Expansion

`local-write-file-hooks`

see section Saving Buffers

`major-mode`

see section Getting Help about a Major Mode

`mark-active`

see section The Mark

`mark-ring`

see section The Mark

`minor-modes`

see section Minor Modes

`mode-line-format`

see section The Data Structure of the Mode Line

`mode-name`

see section Variables Used in the Mode Line

`overwrite-mode`

see section Insertion

`paragraph-separate`

see section Standard Regular Expressions Used in Editing

`paragraph-start`

see section Standard Regular Expressions Used in Editing

`require-final-newline`

see section Insertion

`selective-display`

see section Selective Display

`selective-display-ellipses`

see section Selective Display

`tab-width`

see section Usual Display Conventions

TOC

`truncate-lines`

see section Truncation

Go to the previous, next section.

Go to the previous, next section.

Standard Keymaps

The following symbols are used as the names for various keymaps. Some of these exist when Emacs is first started, others are only loaded when their respective mode is used. This is not an exhaustive list.

Almost all of these maps are used as local maps. Indeed, of the modes that presently exist, only Vip mode and Terminal mode ever change the global keymap.

`Buffer-menu-mode-map`
: A full keymap used by Buffer Menu mode.

`c-mode-map`
: A sparse keymap used in C mode as a local map.

`command-history-map`
: A full keymap used by Command History mode.

`ctl-x-4-map`
: A sparse keymap for subcommands of the prefix C-x 4.

`ctl-x-map`
: A full keymap for C-x commands.

`debugger-mode-map`
: A full keymap used by Debugger mode.

`dired-mode-map`
: A full keymap for `dired-mode` buffers.

`doctor-mode-map`
: A sparse keymap used by Doctor mode.

`edit-abbrevs-map`
: A sparse keymap used in `edit-abbrevs`.

`edit-tab-stops-map`
: A sparse keymap used in `edit-tab-stops`.

`electric-buffer-menu-mode-map`
: A full keymap used by Electric Buffer Menu mode.

`electric-history-map`
: A full keymap used by Electric Command History mode.

`emacs-lisp-mode-map`
: A sparse keymap used in Emacs Lisp mode.

`function-keymap`
: The keymap for the definitions of keypad and function keys.

If there are none, then it contains an empty sparse keymap.

`fundamental-mode-map`

The local keymap for Fundamental mode.
It is empty and should not be changed.

`Helper-help-map`

A full keymap used by the help utility package.
It has the same keymap in its value cell and in its function cell.

`Info-edit-map`

A sparse keymap used by the e command of Info.

`Info-mode-map`

A sparse keymap containing Info commands.

`isearch-mode-map`

A keymap that defines the characters you can type within incremental search.

`lisp-interaction-mode-map`

A sparse keymap used in Lisp mode.

`lisp-mode-map`

A sparse keymap used in Lisp mode.

`mode-specific-map`

The keymap for characters following C-c. Note, this is in the global map. This map is not actually mode specific: its name was chosen to be informative for the user in C-h b (`display-bindings`), where it describes the main use of the C-c prefix key.

`occur-mode-map`

A local keymap used in Occur mode.

`query-replace-map`

A local keymap used for responses in `query-replace` and related commands; also for `y-or-n-p` and `map-y-or-n-p`. The functions that use this map do not support prefix keys; they look up one event at a time.

`text-mode-map`

A sparse keymap used by Text mode.

`view-mode-map`

A full keymap used by View mode.

Go to the previous, next section.

Go to the previous section.

Standard Hooks

The following is a list of hook variables which let you provide functions to be called from within Emacs on suitable occasions.

Most of these variables have names ending with `-hook' are normal hooks, that are run with `run-hooks`. The value of such a hook is a list of functions. The recommended way to put a new function on such a hook is to call `add-hook`. See section Hooks, for more information about using hooks.

The variables whose names end in `-function' have single functions as their values. Usually there is a specific reason why the variable is not a normal hook, such as, the need to pass an argument to the function. (In older Emacs versions, some of these variables had names ending in `-hook' even though they were not normal hooks.)

The variables whose names end in `-hooks' have lists of functions as their values, but these functions are called in a special way (they are passed arguments, or else their values are used).

`activate-mark-hook`
`after-change-function`
`after-init-hook`
`auto-fill-function`
`auto-save-hook`
`before-change-function`
`before-init-hook`
`blink-paren-function`
`c-mode-hook`
`command-history-hook`
`comment-indent-function`
`deactivate-mark-hook`
`dired-mode-hook`
`disabled-command-hook`
`edit-picture-hook`
`electric-buffer-menu-mode-hook`
`electric-command-history-hook`
`electric-help-mode-hook`
`emacs-lisp-mode-hook`
`find-file-hooks`
`find-file-not-found-hooks`

first-change-hook
fortran-comment-hook
fortran-mode-hook
ftp-setup-write-file-hooks
ftp-write-file-hook
indent-mim-hook
LaTeX-mode-hook
ledit-mode-hook
lisp-indent-function
lisp-interaction-mode-hook
lisp-mode-hook
m2-mode-hook
mail-mode-hook
mail-setup-hook
medit-mode-hook
mh-compose-letter-hook
mh-folder-mode-hook
mh-letter-mode-hook
mim-mode-hook
news-mode-hook
news-reply-mode-hook
news-setup-hook
nroff-mode-hook
outline-mode-hook
plain-TeX-mode-hook
pre-abbrev-expand-hook
pre-command-hook
post-command-hook
prolog-mode-hook
protect-innocence-hook
rmail-edit-mode-hook
rmail-mode-hook
rmail-summary-mode-hook
scheme-indent-hook
scheme-mode-hook
scribe-mode-hook

`shell-mode-hook`
`shell-set-directory-error-hook`
`suspend-hook`
`suspend-resume-hook`
`temp-buffer-show-function`
`term-setup-hook`
`terminal-mode-hook`
`terminal-mode-break-hook`
`TeX-mode-hook`
`text-mode-hook`
`vi-mode-hook`
`view-hook`
`window-setup-hook`
`write-contents-hooks`
`write-file-hooks`

Go to the previous section.

GNU Emacs Lisp Reference Manual

(1)

This definition of "environment" is specifically not intended to include all the data which can affect the result of a program.

(2)

Button-down is the conservative antithesis of drag.

GNU Emacs Manual

Go to the next section.

@shorttitlepage GNU Emacs Manual

GNU Emacs Manual

Twelfth Edition, Updated for Emacs Version 19.34

Twelfth Edition
Updated for Emacs Version 19.34,
August 1996

ISBN 1-882114-05-1

Published by the Free Software Foundation
59 Temple Place, Suite 330
Boston, MA 02111-1307 USA

Cover art by Etienne Suvasa.

Preface

This manual documents the use and simple customization of the Emacs editor. The reader is not expected to be a programmer; simple customizations do not require programming skill. But the user who is not interested in customizing can ignore the scattered customization hints.

This is primarily a reference manual, but can also be used as a primer. For complete beginners, it is a good idea to start with the on-line, learn-by-doing tutorial, before reading the manual. To run the tutorial, start Emacs and type C-h t. This way you can learn Emacs by using Emacs on a specially designed file which describes commands, tells you when to try them, and then explains the results you see.

On first reading, just skim chapters one and two, which describe the notational conventions of the manual and the general appearance of the Emacs display screen. Note which questions are answered in these

chapters, so you can refer back later. After reading chapter four you should practice the commands th
The next few chapters describe fundamental techniques and concepts that are used constantly. You n
to understand them thoroughly, experimenting with them if necessary.

Chapters 14 through 18 describe intermediate-level features that are useful for all kinds of editing. Chapter 19 and following chapters describe features that you may or may not want to use; read those chapters when you need them

Read the Trouble chapter if Emacs does not seem to be working properly. It explains how to cope with some common problems (see section Dealing with Emacs Trouble), as well as when and how to report Emacs bugs (see section Reporting Bugs). To find the documentation on a particular command, look in the index. Keys (character commands) and command names have separate indexes. There is also a glossary, with a cross reference for each term.

This manual is available as a printed book and also as an Info file. The Info file is for on-line perusal with the Info program, which will be the principle way of viewing documentation on-line in the GNU system. Both the Info file and the Info program itself are distributed along with GNU Emacs. The Info file and the printed book contain substantially the same text and are generated from the same source files, which are also distributed along with GNU Emacs.

GNU Emacs is a member of the Emacs editor family. There are many Emacs editors, all sharing common principles of organization. For information on the underlying philosophy of Emacs and the lessons learned from its development, write for a copy of AI memo 519a, "Emacs, the Extensible, Customizable Self-Documenting Display Editor", to Publications Department, Artificial Intelligence Lab, 545 Tech Square, Cambridge, MA 02139, USA. At last report they charge $2.25 per copy. Another useful publication is LCS TM-165, "A Cookbook for an Emacs", by Craig Finseth, available from Publications Department, Laboratory for Computer Science, 545 Tech Square, Cambridge, MA 02139, USA. The price today is $3.

This edition of the manual is intended for use with GNU Emacs installed on GNU and Unix systems. GNU Emacs can also be used on VMS, MS-DOS (aka. MS-DOG), Windows NT, and Windows 95 systems. Those systems use different file name syntax; in addition, VMS and MS-DOS do not support all GNU Emacs features. We don't try to describe VMS usage in this manual. See section MS-DOS Issues, for information about using Emacs on MS-DOS.

Go to the next section.

Go to the previous, next section.

Distribution

GNU Emacs is free software; this means that everyone is free to use it and free to redistribute it on certain conditions. GNU Emacs is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of GNU Emacs that they might get from you. The precise conditions are found in the GNU General Public License that comes with Emacs and also appears following this section.

One way to get a copy of GNU Emacs is from someone else who has it. You need not ask for our permission to do so, or tell any one else; just copy it. If you have access to the Internet, you can get the latest distribution version of GNU Emacs by anonymous FTP; see the file `etc/FTP' in the Emacs distribution for more information.

You may also receive GNU Emacs when you buy a computer. Computer manufacturers are free to distribute copies on the same terms that apply to everyone else. These terms require them to give you the full sources, including whatever changes they may have made, and to permit you to redistribute the GNU Emacs received from them under the usual terms of the General Public License. In other words, the program must be free for you when you get it, not just free for the manufacturer.

You can also order copies of GNU Emacs from the Free Software Foundation, on various magnetic media or on CD-ROM. This is a convenient and reliable way to get a copy; it is also a good way to help fund our work. (The Foundation has always received most of its funds in this way.) An order form is included at the end of manuals printed by the Foundation. It is also included in the file `etc/ORDERS' in the Emacs distribution. For further information, write to

```
Free Software Foundation
59 Temple Place, Suite 330
Boston, MA  02111-1307 USA
USA
```

The income from distribution fees goes to support the foundation's purpose: the development of new free software, and improvements to our existing programs including GNU Emacs.

If you find GNU Emacs useful, please **send a donation** to the Free Software Foundation to support our work. Donations to the Free Software Foundation are tax deductible. If you use GNU Emacs at your workplace, suggest that the company make a donation. If company policy is unsympathetic to the idea of donating to charity, you might instead suggest ordering a CD-ROM from the Foundation occasionally, or subscribing to periodic updates.

Contributors to GNU Emacs include Per Abrahamsen, Jay K. Adams, Joe Arceneaux, Boaz Ben-Zvi, Jim Blandy, Frank Bresz, Kevin Broadey, Vincent Broman, David M. Brown, Hans Chalupsky, Bob Chassell, James Clark, Mike Clarkson, Doug Cutting, Michael DeCorte, Gary Delp, Matthieu Devin, Scott Draves, Viktor Dukhovni, Rolf Ebert, Torbj@"orn Einarsson, Hans Henrik Eriksen, Michael Ernst,

Ata Etemadi, Fred Fish, Karl Fogel, Noah Friedman, Keith Gabryelski, Kevin Gallagher, Kevin Gall Howard Gayle, Stephen Gildea, David Gillespie, Boris Goldowsky, Michael Gschwind, Henry Guillaume, Doug Gwyn, Chris Hanson, K. Shane Hartman, Markus Heritsch, Karl Heuer, Manabu Higashida, Anders Holst, Lars Ingebrigtsen, Andrew Innes, Michael K. Johnson, Kyle Jones, Brewster Kahle, David Kaufman, Henry Kautz, Howard Kaye, Michael Kifer, Richard King, Larry K. Kolodney, Robert Krawitz, Sebastian Kremer, Geoff Kuenning, David K@aa gedal, Daniel LaLiberte, Aaron Larson, James R. Larus, Lars Lindberg, Neil M. Mager, Ken Manheimer, Bill Mann, Brian Marick, Simon Marshall, Bengt Martensson, Charlie Martin, Thomas May, Roland McGrath, David Megginson, Richard Mlynarik, Keith Moore, Erik Naggum, Thomas Neumann, Mike Newton, Jurgen Nickelsen, Jeff Norden, Jeff Peck, Damon Anton Permezel, Tom Perrine, Daniel Pfeiffer, Fred Pierresteguy, Christian Plaunt, Francesco A. Potorti, Michael D. Prange, Ashwin Ram, Eric S. Raymond, Paul Reilly, Edward M. Reingold, Rob Riepel, Roland B. Roberts, John Robinson, William Rosenblatt, Guillermo J. Rozas, Wolfgang Rupprecht, James B. Salem, Masahiko Sato, William Schelter, Gregor Schmid, Michael Schmidt, Ronald S. Schnell, Philippe Schnoebelen, Stephen Schoef, Randal Schwartz, Mark Shapiro, Olin Shivers, Espen Skoglund, Rick Sladkey, Lynn Slater, Chris Smith, David Smith, William Sommerfeld, Ake Stenhoff, Jonathan Stigelman, Steve Strassman, Spencer Thomas, Jim Thompson, Masanobu Umeda, Geoffrey Voelker, Johan Vromans, Barry Warsaw, Morten Welinder, Joseph Brian Wells, Ed Wilkinson, Mike Williams, Steven A. Wood, Dale R. Worley, Felix S. T. Wu, Tom Wurgler, Eli Zaretskii, Jamie Zawinski, and Neal Ziring.

Go to the previous, next section.

Go to the previous, next section.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free

use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that wor not derived from the Program, and can be reasonably considered independent and separate wo in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
    1. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
    2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
    3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

    The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

    If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so

long as such parties remain in full compliance.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number

of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the

"copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type `show c'
for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program `Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License

instead of this License.

Go to the previous, next section.

TOC

Go to the previous, next section.

Introduction

You are reading about GNU Emacs, the GNU incarnation of the advanced, self-documenting, customizable, extensible real-time display editor Emacs. (The `G' in `GNU' is not silent.)

We say that Emacs is a display editor because normally the text being edited is visible on the screen and is updated automatically as you type your commands. See section The Organization of the Screen.

We call it a real-time editor because the display is updated very frequently, usually after each character or pair of characters you type. This minimizes the amount of information you must keep in your head as you edit. See section Basic Editing Commands.

We call Emacs advanced because it provides facilities that go beyond simple insertion and deletion: controlling subprocesses; automatic indentation of programs; viewing two or more files at once; editing formatted text; and dealing in terms of characters, words, lines, sentences, paragraphs, and pages, as well as expressions and comments in several different programming languages.

Self-documenting means that at any time you can type a special character, Control-h, to find out what your options are. You can also use it to find out what any command does, or to find all the commands that pertain to a topic. See section Help.

Customizable means that you can change the definitions of Emacs commands in little ways. For example, if you use a programming language in which comments start with `<**' and end with `**>', you can tell the Emacs comment manipulation commands to use those strings (see section Manipulating Comments). Another sort of customization is rearrangement of the command set. For example, if you prefer the four basic cursor motion commands (up, down, left and right) on keys in a diamond pattern on the keyboard, you can rebind the keys that way. See section Customization.

Extensible means that you can go beyond simple customization and write entirely new commands, programs in the Lisp language to be run by Emacs's own Lisp interpreter. Emacs is an "on-line extensible" system, which means that it is divided into many functions that call each other, any of which can be redefined in the middle of an editing session. Almost any part of Emacs can be replaced without making a separate copy of all of Emacs. Most of the editing commands of Emacs are written in Lisp already; the few exceptions could have been written in Lisp but are written in C for efficiency. Although only a programmer can write an extension, anybody can use it afterward.

When run under the X Window System, Emacs provides its own menus and convenient bindings to mouse buttons. But Emacs can provide many of the benefits of a window system on a text-only terminal. For instance, you can look at or edit several files at once, move text between them, and edit files at the same time as you run shell commands.

Go to the previous, next section.

Go to the previous, next section.

The Organization of the Screen

On a text-only terminal, the Emacs display occupies the whole screen. On the X Window System, Emacs creates its own X windows to use. We use the term frame to mean an entire text-only screen or an entire X window used by Emacs. Emacs uses both kinds of frames in the same way to display your editing. Emacs normally starts out with just one frame, but you can create additional frames if you wish. See section Frames and X Windows.

When you start Emacs, the entire frame except for the last line is devoted to the text you are editing. This area is called window. The last line is a special echo area or minibuffer window where prompts appear and where you can enter responses. You can subdivide the large text window horizontally or vertically into multiple text windows, each of which can be used for a different file (see section Multiple Windows). In this manual, the word "window" always refers to the subdivisions of a frame within Emacs.

The window that the cursor is in is the selected window, in which editing takes place. Most Emacs commands implicitly apply to the text in the selected window (though mouse commands generally operate on whatever window you click them in, whether selected or not). The other windows display text for reference only, unless/until you select them. If you use multiple frames under the X Window System, then giving the input focus to a particular frame selects a window in that frame.

Each window's last line is a mode line which describes what is going on in that window. It appears in inverse video if the terminal supports that, and contains text that starts like `-----Emacs: something'. Its purpose is to indicate what buffer is being displayed above it in the window; what major and minor modes are in use; and whether the buffer contains unsaved changes.

Point

Within Emacs, the terminal's cursor shows the location at which editing commands will take effect. This location is called point. Many Emacs commands move point through the text, so that you can edit at different places in it. You can also place point by clicking mouse button 1.

While the cursor appears to point at a character, you should think of point as between two characters; it points before the character that appears under the cursor. For example, if your text looks like `frob' with the cursor over the `b', then point is between the `o' and the `b'. If you insert the character `!' at that position, the result is `fro!b', with point between the `!' and the `b'. Thus, the cursor remains over the `b', as before.

Sometimes people speak of "the cursor" when they mean "point", or speak of commands that move point as "cursor motion" commands.

Terminals have only one cursor, and when output is in progress it must appear where the typing is being done. This does not mean that point is moving. It is only that Emacs has no way to show you the location

of point except when the terminal is idle.

If you are editing several files in Emacs, each in its own buffer, each buffer has its own point location. A buffer that is not currently displayed remembers where point is in case you display it again later.

When there are multiple windows in a frame, each window has its own point location. The cursor shows the location of point in the selected window. This also is how you can tell which window is selected. If the same buffer appears in more than one window, each window has its own position for point in that buffer.

When there are multiple frames, each frame can display one cursor. The cursor in the selected frame is solid; the cursor in other frames is a hollow box, and appears in the window that would be selected if you give the input focus to that frame.

The term `point' comes from the character `.', which was the command in TECO (the language in which the original Emacs was written) for accessing the value now called `point'.

The Echo Area

The line at the bottom of the frame (below the mode line) is the echo area. It is used to display small amounts of text for several purposes.

Echoing means displaying the characters that you type. Outside Emacs, the operating system normally echoes all your input. Emacs handles echoing differently.

Single-character commands do not echo in Emacs, and multi-character commands echo only if you pause while typing them. As soon as you pause for more than a second in the middle of a command, Emacs echoes all the characters of the command so far. This is to prompt you for the rest of the command. Once echoing has started, the rest of the command echoes immediately as you type it. This behavior is designed to give confident users fast response, while giving hesitant users maximum feedback. You can change this behavior by setting a variable (see section Variables Controlling Display).

If a command cannot be executed, it may print an error message in the echo area. Error messages are accompanied by a beep or by flashing the screen. Also, any input you have typed ahead is thrown away when an error happens.

Some commands print informative messages in the echo area. These messages look much like error messages, but they are not announced with a beep and do not throw away input. Sometimes the message tells you what the command has done, when this is not obvious from looking at the text being edited. Sometimes the sole purpose of a command is to print a message giving you specific information--for example, C-x = prints a message describing the character position of point in the text and its current column in the window. Commands that take a long time often display messages ending in `...' while they are working, and add `done' at the end when they are finished.

Echo area informative messages are saved in an editor buffer named `*Messages*'. (We have not explained buffers yet; see section Using Multiple Buffers, for more information about them.) If you miss a message that appears briefly on the screen, you can switch to the `*Messages*' buffer to see it again. Successive progress messages are often collapsed into one.

The size of `*Messages*' is limited to a certain number of lines. The variable `message-log-max` specifies how many lines. Once the buffer has that many lines, each line added at the end deletes one line from the beginning. See section Variables, for how to set variables such as `message-log-max`.

The echo area is also used to display the minibuffer, a window that is used for reading arguments to commands, such as the name of a file to be edited. When the minibuffer is in use, the echo area begins with a prompt string that usually ends with a colon; also, the cursor appears in that line because it is the selected window. You can always get out of the minibuffer by typing C-g. See section The Minibuffer.

The Mode Line

Each text window's last line is a mode line which describes what is going on in that window. When there is only one text window, the mode line appears right above the echo area. The mode line is in inverse video if the terminal supports that, it starts and ends with dashes, and it contains text like `Emacs: something'.

A few special editing modes, such as Dired and Rmail, display something else in place of `Emacs: something'. The rest of the mode line still has the usual meaning.

Normally, the mode line looks like this:

```
--ch-Emacs: buf      (major minor)--line--pos------
```

This gives information about the buffer being displayed in the window: the buffer's name, what major and minor modes are in use, whether the buffer's text has been changed, and how far down the buffer you are currently looking.

ch contains two stars `**' if the text in the buffer has been edited (the buffer is "modified"), or `--' if the buffer has not been edited. For a read-only buffer, it is `%*' if the buffer is modified, and `%%' otherwise.

buf is the name of the window's buffer. In most cases this is the same as the name of a file you are editing. See section Using Multiple Buffers.

The buffer displayed in the selected window (the window that the cursor is in) is also Emacs's selected buffer, the one that editing takes place in. When we speak of what some command does to "the buffer", we are talking about the currently selected buffer.

line is `L' followed by the current line number of point. This is present when Line Number mode is enabled (which it normally is). You can optionally display the current column number too, by turning on Column Number mode (which is not enabled by default because it is somewhat slower). See section Optional Mode Line Features.

pos tells you whether there is additional text above the top of the window, or below the bottom. If your buffer is small and it is all visible in the window, pos is `All'. Otherwise, it is `Top' if you are looking at the beginning of the buffer, `Bot' if you are looking at the end of the buffer, or `nn%', where nn is the percentage of the buffer above the top of the window.

major is the name of the major mode in effect in the buffer. At any time, each buffer is in one and only one of the possible major modes. The major modes available include Fundamental mode (the least specialized), Text mode, Lisp mode, C mode, Texinfo mode, and many others. See section Major Modes, for details of how the modes differ and how to select one.

Some major modes display additional information after the major mode name. For example, Rmail buffers display the current message number and the total number of messages. Compilation buffers and Shell buffers display the status of the subprocess.

minor is a list of some of the minor modes that are turned on at the moment in the window's chosen buffer. For example, `Fill' means that Auto Fill mode is on. `Abbrev' means that Word Abbrev mode is on. `Ovwrt' means that Overwrite mode is on. See section Minor Modes, for more information. `Narrow' means that the buffer being displayed has editing restricted to only a portion of its text. This is not really a minor mode, but is like one. See section Narrowing. `Def' means that a keyboard macro is being defined. See section Keyboard Macros.

In addition, if Emacs is currently inside a recursive editing level, square brackets (`[...]') appear around the parentheses that surround the modes. If Emacs is in one recursive editing level within another, double square brackets appear, and so on. Since recursive editing levels affect Emacs globally, not just one buffer, the square brackets appear in every window's mode line or not in any of them. See section Recursive Editing Levels.

See section Optional Mode Line Features, for features that add other handy information to the mode line, such as the current line number of point, the current time, and whether new mail for you has arrived.

Go to the previous, next section.

Go to the previous, next section.

Characters, Keys and Commands

This chapter explains the character sets used by Emacs for input commands and for the contents of files, and also explains the concepts of keys and commands which are fundamental for understanding how Emacs interprets your keyboard and mouse input.

Kinds of User Input

GNU Emacs uses an extension of the ASCII character set for keyboard input; it also accepts non-character input events including function keys and mouse button actions.

ASCII consists of 128 character codes. Some of these codes are assigned graphic symbols such as `a' and `='; the rest are control characters, such as Control-a (usually written C-a for short). C-a gets its name from the fact that you type it by holding down the CTRL key while pressing a.

Some control characters have special names, and special keys you can type them with: for example, RET, TAB, LFD, DEL and ESC. The space character is usually referred to below as SPC, even though strictly speaking it is a graphic character whose graphic happens to be blank.

On ASCII terminals, there are only 32 possible control characters. These are the control variants of letters and `@[]\^_'. In addition, the shift key is meaningless with control characters: C-a and C-A are the same character, and Emacs cannot distinguish them.

But the Emacs character set has room for control variants of all characters, and for distinguishing between C-a and C-A. X Windows makes it possible to enter all these characters. For example, C-- (that's Control-Minus) and C-5 are meaningful Emacs commands under X.

Another Emacs character set extension is that characters have additional modifier bits. Only one modifier bit is commonly used; it is called Meta. Every character has a Meta variant; examples include Meta-a (normally written M-a, for short), M-A (not the same character as M-a, but those two characters normally have the same meaning in Emacs), M-RET, and M-C-a. For reasons of tradition, we usually write C-M-a rather than M-C-a; logically speaking, the order in which the modifier keys CTRL and META are mentioned does not matter.

Some terminals have a META key, and allow you to type Meta characters by holding this key down. Thus, Meta-a is typed by holding down META and pressing a. The META key works much like the SHIFT key. Such a key is not always labeled META, however, as this function is often a special option for a key with some other primary purpose.

If there is no META key, you can still type Meta characters using two-character sequences starting with ESC. Thus, to enter M-a, you could type ESC a. To enter C-M-a, you would type ESC C-a. ESC is allowed on terminals with META keys, too, in case you have formed a habit of using it. X Windows provides several other modifier keys that can be applied to any input character. These are called SUPER, HYPER and ALT. We write `s-', `H-' and `A-' to say that a character uses these modifiers. Thus, s-H-C-x

is short for Super-Hyper-Control-x. Not all X terminals actually provide keys for these modifier flags; fact, many terminals have a key labeled ALT which is really a META key. The standard key bindings Emacs do not include any characters with these modifiers. But you can assign them meanings of your own by customizing Emacs.

Keyboard input includes keyboard keys that are not characters at all: for example function keys and arrow keys. Mouse buttons are also outside the gamut of characters. You can modify these events with the modifier keys CONTROL, META, SUPER, HYPER and ALT like keyboard characters.

Input characters and non-character inputs are collectively called input events. See section 'Input Events' in The Emacs Lisp Manual, for more information. If you are not doing Lisp programming, but simply want to redefine the meaning of some characters or non-character events, see section Customization.

ASCII terminals cannot really send anything to the computer except ASCII characters. These terminals use a sequence of characters to represent each function key. But that is invisible to the Emacs user, because the keyboard input routines recognize these special sequences and convert them to function key events before any other part of Emacs gets to see them.

Keys

A key sequence (key, for short) is a sequence of input events that are meaningful as a unit--as "a single command." Some Emacs command sequences are just one character or one event; for example, just C-f is enough to move forward one character. But Emacs also has commands that take two or more events to invoke.

If a sequence of events is enough to invoke a command, it is a complete key. Examples of complete keys include C-a, X, RET, NEXT (a function key), DOWN (an arrow key), C-x C-f and C-x 4 C-f. If it isn't long enough to be complete, we call it a prefix key. The above examples show that C-x and C-x 4 are prefix keys. Every key sequence is either a complete key or a prefix key.

Most single characters constitute complete keys in the standard Emacs command bindings. A few of them are prefix keys. A prefix key combines with the following input event to make a longer key sequence, which may itself be complete or a prefix. For example, C-x is a prefix key, so C-x and the next input event combine to make a two-character key sequence. Most of these key sequences are complete keys, including C-x C-f and C-x b. A few, such as C-x 4 and C-x r, are themselves prefix keys that lead to three-character key sequences. There's no limit to the length of a key sequence, but in practice people rarely use sequences longer than four events.

By contrast, you can't add more events onto a complete key. For example, the two-character sequence C-f C-k is not a key, because the C-f is a complete key in itself. It's impossible to give C-f C-k an independent meaning as a command. C-f C-k is two key sequences, not one.

All told, the prefix keys in Emacs are C-c, C-h, C-x, C-x C-a, C-x n, C-x r, C-x v, C-x 4, C-x 5, C-x 6, and ESC. But this is not cast in concrete; it is just a matter of Emacs's standard key bindings. If you customize Emacs, you can make new prefix keys, or eliminate these. See section Customizing Key Bindings.

If you do make or eliminate prefix keys, that changes the set of possible key sequences. For example, if you redefine C-f as a prefix, C-f C-k automatically becomes a key (complete, unless you define it to be a prefix). Conversely, if you remove the prefix definition of C-x 4, then C-x 4 f (or C-x 4 anything) is no longer a key.

Typing the help character (C-h or F1) after a prefix character displays a list of the commands starting with that prefix. There are a few prefix characters for which C-h does not work--for historical reasons, they have other meanings for C-h which are not easy to change. But F1 should work for all prefix characters.

Keys and Commands

This manual is full of passages that tell you what particular keys do. But Emacs does not assign meanings to keys directly. Instead, Emacs assigns meanings to named commands, and then gives keys their meanings by binding them to commands.

Every command has a name chosen by a programmer. The name is usually made of a few English words separated by dashes; for example, `next-line` or `forward-word`. A command also has a function definition which is a Lisp program; this is what makes the command do what it does. In Emacs Lisp, a command is actually a special kind of Lisp function; one which specifies how to read arguments for it and call it interactively. For more information on commands and functions, see section 'What Is a Function' in The Emacs Lisp Reference Manual. (The definition we use in this manual is simplified slightly.)

The bindings between keys and commands are recorded in various tables called keymaps. See section Keymaps.

When we say that "C-n moves down vertically one line" we are glossing over a distinction that is irrelevant in ordinary use but is vital in understanding how to customize Emacs. It is the command `next-line` that is programmed to move down vertically. C-n has this effect *because* it is bound to that command. If you rebind C-n to the command `forward-word` then C-n will move forward by words instead. Rebinding keys is a common method of customization.

In the rest of this manual, we usually ignore this subtlety to keep things simple. To give the information needed for customization, we state the name of the command which really does the work in parentheses after mentioning the key that runs it. For example, we will say that "The command C-n (`next-line`) moves point vertically down," meaning that `next-line` is a command that moves vertically down and C-n is a key that is standardly bound to it.

While we are on the subject of information for customization only, it's a good time to tell you about variables. Often the description of a command will say, "To change this, set the variable `mumble-foo`." A variable is a name used to remember a value. Most of the variables documented in this manual exist just to facilitate customization: some command or other part of Emacs examines the variable and behaves differently according to the value that you set. Until you are interested in customizing, you can ignore the information about variables. When you are ready to be interested, read the basic information on variables, and then the information on individual variables will make sense. See section Variables.

Character Set for Text

Emacs buffers use an 8-bit character set, because bytes have 8 bits. ASCII graphic characters in Emacs buffers are displayed with their graphics. The newline character (which has the same character code as LFD) is displayed by starting a new line. The tab character is displayed by moving to the next tab stop column (normally every 8 columns). Other control characters are displayed as a caret (`^') followed by the non-control version of the character; thus, C-a is displayed as `^A'. Non-ASCII characters 128 and up are displayed with octal escape sequences; thus, character code 243 (octal) is displayed as `\243'.

You can customize the display of these character codes (or ASCII characters) by creating a display table. See section 'Display Tables' in The Emacs Lisp Reference Manual. This is useful for editing files that use 8-bit European character sets. See section European Character Set Display.

Go to the previous, next section.

Go to the previous, next section.

Entering and Exiting Emacs

The usual way to invoke Emacs is with the shell command `emacs'. Emacs clears the screen and then displays an initial help message and copyright notice. Some operating systems discard all type-ahead when Emacs starts up; they give Emacs no way to prevent this. Therefore, it is advisable to wait until Emacs clears the screen before typing your first editing command.

If you run Emacs from a shell window under the X Window System, run it in the background with `emacs&'. This way, Emacs does not tie up the shell window, so you can use that to run other shell commands while Emacs operates its own X windows. You can begin typing Emacs commands as soon as you direct your keyboard input to the Emacs frame.

When Emacs starts up, it makes a buffer named `*scratch*'. That's the buffer you start out in. The `*scratch*' buffer uses Lisp Interaction mode; you can use it to type Lisp expressions and evaluate them, or you can ignore that capability and simply doodle. (You can specify a different major mode for this buffer by setting the variable `initial-major-mode` in your init file. See section The Init File, `~/.emacs'.)

It is possible to specify files to be visited, Lisp files to be loaded, and functions to be called, by giving Emacs arguments in the shell command line. See section Command Line Arguments. But we don't recommend doing this. The feature exists mainly for compatibility with other editors.

Many other editors are designed to be started afresh each time you want to edit. You edit one file and then exit the editor. The next time you want to edit either another file or the same one, you must run the editor again. With these editors, it makes sense to use a command line argument to say which file to edit.

But starting a new Emacs each time you want to edit a different file does not make sense. For one thing, this would be annoyingly slow. For another, this would fail to take advantage of Emacs's ability to visit more than one file in a single editing session. And it would lose the other accumulated context, such as registers, undo history, and the mark ring.

The recommended way to use GNU Emacs is to start it only once, just after you log in, and do all your editing in the same Emacs session. Each time you want to edit a different file, you visit it with the existing Emacs, which eventually comes to have many files in it ready for editing. Usually you do not kill the Emacs until you are about to log out. See section File Handling, for more information on visiting more than one file.

Exiting Emacs

There are two commands for exiting Emacs because there are two kinds of exiting: suspending Emacs and killing Emacs.

Suspending means stopping Emacs temporarily and returning control to its parent process (usually a

shell), allowing you to resume editing later in the same Emacs job, with the same buffers, same kill r same undo history, and so on. This is the usual way to exit.

Killing Emacs means destroying the Emacs job. You can run Emacs again later, but you will get a fresh Emacs; there is no way to resume the same editing session after it has been killed.

C-z

Suspend Emacs (`suspend-emacs`) or iconify a frame (`iconify-or-deiconify-frame`).

C-x C-c

Kill Emacs (`save-buffers-kill-emacs`).

To suspend Emacs, type C-z (`suspend-emacs`). This takes you back to the shell from which you invoked Emacs. You can resume Emacs with the shell command `%emacs' in most common shells.

On systems that do not support suspending programs, C-z starts an inferior shell that communicates directly with the terminal. Emacs waits until you exit the subshell. (The way to do that is probably with C-d or `exit', but it depends on which shell you use.) The only way on these systems to get back to the shell from which Emacs was run (to log out, for example) is to kill Emacs.

Suspending also fails if you run Emacs under a shell that doesn't support suspending programs, even if the system itself does support it. In such a case, you can set the variable `cannot-suspend` to a non-`nil` value to force C-z to start an inferior shell. (One might also describe Emacs's parent shell as "inferior" for failing to support job control properly, but that is a matter of taste.)

When Emacs communicates directly with an X server and creates its own dedicated X windows, C-z has a different meaning. Suspending an applications that uses its own X windows is not meaningful or useful. Instead, C-z runs the command `iconify-or-deiconify-frame`, which temporarily closes up the selected Emacs frame (see section Frames and X Windows). The way to get back to a shell window is with the window manager.

To kill Emacs, type C-x C-c (`save-buffers-kill-emacs`). A two-character key is used for this to make it harder to type. This command first offers to save any modified file-visiting buffers. If you do not save them all, it asks for reconfirmation with yes before killing Emacs, since any changes not saved will be lost forever. Also, if any subprocesses are still running, C-x C-c asks for confirmation about them, since killing Emacs will kill the subprocesses immediately.

There is no way to restart an Emacs session once you have killed it. You can, however, arrange for Emacs to record certain session information, such as which files are visited, when you kill it, so that the next time you restart Emacs it will try to visit the same files and so on. See section Saving Emacs Sessions.

The operating system usually listens for certain special characters whose meaning is to kill or suspend the program you are running. **This operating system feature is turned off while you are in Emacs.** The meanings of C-z and C-x C-c as keys in Emacs were inspired by the use of C-z and C-c on several operating systems as the characters for stopping or killing a program, but that is their only relationship with the operating system. You can customize these keys to run any commands of your choice (see section Keymaps).

TOC

Go to the previous, next section.

Go to the previous, next section.

Basic Editing Commands

We now give the basics of how to enter text, make corrections, and save the text in a file. If this material is new to you, you might learn it more easily by running the Emacs learn-by-doing tutorial. To use the tutorial, run Emacs and type Control-h t (`help-with-tutorial`).

To clear the screen and redisplay, type C-l (`recenter`).

Inserting Text

To insert printing characters into the text you are editing, just type them. This inserts the characters you type into the buffer at the cursor (that is, at point; see section Point). The cursor moves forward, and any text after the cursor moves forward too. If the text in the buffer is `FOOBAR', with the cursor before the `B', then if you type XX, you get `FOOXXBAR', with the cursor still before the `B'.

To delete text you have just inserted, use DEL. DEL deletes the character *before* the cursor (not the one that the cursor is on top of or under; that is the character after the cursor). The cursor and all characters after it move backwards. Therefore, if you type a printing character and then type DEL, they cancel out.

To end a line and start typing a new one, type RET. This inserts a newline character in the buffer. If point is in the middle of a line, RET splits the line. Typing DEL when the cursor is at the beginning of a line deletes the preceding newline, thus joining the line with the preceding line.

Emacs can split lines automatically when they become too long, if you turn on a special minor mode called Auto Fill mode. See section Filling Text, for how to use Auto Fill mode.

If you prefer to have text characters replace (overwrite) existing text rather than shove it to the right, you can enable Overwrite mode, a minor mode. See section Minor Modes.

Direct insertion works for printing characters and SPC, but other characters act as editing commands and do not insert themselves. If you need to insert a control character or a character whose code is above 200 octal, you must quote it by typing the character Control-q (`quoted-insert`) first. (This character's name is normally written C-q for short.) There are two ways to use C-q:

- C-q followed by any non-graphic character (even C-g) inserts that character.
- C-q followed by three octal digits inserts the character with the specified character code.

A numeric argument to C-q specifies how many copies of the quoted character should be inserted (see section Numeric Arguments).

Customization information: DEL in most modes runs the command `delete-backward-char`; RET runs the command `newline`, and self-inserting printing characters run the command `self-insert`, which inserts whatever character was typed to invoke it. Some major modes rebind DEL to other commands.

Changing the Location of Point

To do more than insert characters, you have to know how to move point (see section Point). The simplest way to do this is with arrow keys, or by clicking the left mouse button where you want to move to.

There are also control and meta characters for cursor motion. Some are equivalent to the arrow keys (these date to the days before terminals had arrow keys, and are usable on terminals which don't have them). Others do more sophisticated things.

C-a

Move to the beginning of the line (`beginning-of-line`).

C-e

Move to the end of the line (`end-of-line`).

C-f

Move forward one character (`forward-char`).

C-b

Move backward one character (`backward-char`).

M-f

Move forward one word (`forward-word`).

M-b

Move backward one word (`backward-word`).

C-n

Move down one line, vertically (`next-line`). This command attempts to keep the horizontal position unchanged, so if you start in the middle of one line, you end in the middle of the next. When on the last line of text, C-n creates a new line and moves onto it.

C-p

Move up one line, vertically (`previous-line`).

M-r

Move point to left margin, vertically centered in the window (`move-to-window-line`). Text does not move on the screen.

A numeric argument says which screen line to place point on. It counts screen lines down from the top of the window (zero for the top line). A negative argument counts lines from the bottom (-1 for the bottom line).

M-<

Move to the top of the buffer (`beginning-of-buffer`). With numeric argument n, move to n/10 of the way from the top. See section Numeric Arguments, for more information on numeric arguments.

M->

Move to the end of the buffer (`end-of-buffer`).

M-x goto-char

Read a number n and move point to character number n. Position 1 is the beginning of the buffer.

M-x goto-line

Read a number n and move point to line number n. Line 1 is the beginning of the buffer.

C-x C-n

Use the current column of point as the semipermanent goal column for C-n and C-p (`set-goal-column`). Henceforth, those commands always move to this column in each line moved into, or as close as possible given the contents of the line. This goal column remains in effect until canceled.

C-u C-x C-n

Cancel the goal column. Henceforth, C-n and C-p once again try to stick to a fixed horizontal position, as usual.

If you set the variable `track-eol` to a non-`nil` value, then C-n and C-p when at the end of the starting line move to the end of another line. Normally, `track-eol` is `nil`. See section Variables, for how to set variables such as `track-eol`.

Normally, C-n on the last line of a buffer appends a newline to it. If the variable `next-line-add-newlines` is `nil`, then C-n gets an error instead (like C-p on the first line).

Erasing Text

DEL
: Delete the character before point (`delete-backward-char`).

C-d
: Delete the character after point (`delete-char`).

C-k
: Kill to the end of the line (`kill-line`).

M-d
: Kill forward to the end of the next word (`kill-word`).

M-DEL
: Kill back to the beginning of the previous word (`backward-kill-word`).

You already know about the DEL key which deletes the character before point (that is, before the cursor). Another key, Control-d (C-d for short), deletes the character after point (that is, the character that the cursor is on). This shifts the rest of the text on the line to the left. If you type C-d at the end of a line, it joins together that line and the next line.

To erase a larger amount of text, use the C-k key, which kills a line at a time. If you type C-k at the beginning or middle of a line, it kills all the text up to the end of the line. If you type C-k at the end of a line, it joins that line and the next line.

See section Deletion and Killing, for more flexible ways of killing text.

Undoing Changes

You can undo all the recent changes in the buffer text, up to a certain point. Each buffer records changes individually, and the undo command always applies to the current buffer. Usually each editing command makes a separate entry in the undo records, but some commands such as `query-replace` make many entries, and very simple commands such as self-inserting characters are often grouped to make undoing less tedious.

C-x u
: Undo one batch of changes--usually, one command worth (`undo`).

C-_
: The same.

The command C-x u or C-_ is how you undo. The first time you give this command, it undoes the last change. Point moves back to where it was before the command that made the change.

Consecutive repetitions of C-_ or C-x u undo earlier and earlier changes, back to the limit of the undo information available. If all recorded changes have already been undone, the undo command prints an error message and does

nothing.

Any command other than an undo command breaks the sequence of undo commands. Starting from that moment, the previous undo commands become ordinary changes that you can undo. Thus, to redo changes you have undone, type C-f or any other command that will harmlessly break the sequence of undoing, then type more undo commands.

If you notice that a buffer has been modified accidentally, the easiest way to recover is to type C-_ repeatedly until the stars disappear from the front of the mode line. At this time, all the modifications you made have been canceled. Whenever an undo command makes the stars disappear from the mode line, it means that the buffer contents are the same as they were when the file was last read in or saved.

If you do not remember whether you changed the buffer deliberately, type C-_ once. When you see the last change you made undone, you will see whether it was an intentional change. If it was an accident, leave it undone. If it was deliberate, redo the change as described above.

Not all buffers record undo information. Buffers whose names start with spaces don't; these buffers are used internally by Emacs and its extensions to hold text that users don't normally look at or edit.

You cannot undo mere cursor motion; only changes in the buffer contents save undo information. However, some cursor motion commands set the mark, so if you use these commands from time to time, you can move back to the neighborhoods you have moved through by popping the mark ring (see section The Mark Ring).

When the undo information for a buffer becomes too large, Emacs discards the oldest undo information from time to time (during garbage collection). You can specify how much undo information to keep by setting two variables: `undo-limit` and `undo-strong-limit`. Their values are expressed in units of bytes of space.

The variable `undo-limit` sets a soft limit: Emacs keeps undo data for enough commands to reach this size, and perhaps exceed it, but does not keep data for any earlier commands beyond that. Its default value is 20000. The variable `undo-strong-limit` sets a stricter limit: the command which pushes the size past this amount is itself forgotten. Its default value is 30000.

Regardless of the values of those variables, the most recent change is never discarded, so there is no danger that garbage collection occurring right after an unintentional large change might prevent you from undoing it.

The reason the `undo` command has two keys, C-x u and C-_, set up to run it is that it is worthy of a single-character key, but on some keyboards it is not obvious how to type C-_. C-x u is an alternative you can type straightforwardly on any terminal.

Files

The commands described above are sufficient for creating and altering text in an Emacs buffer; the more advanced Emacs commands just make things easier. But to keep any text permanently you must put it in a file. Files are named units of text which are stored by the operating system for you to retrieve later by name. To look at or use the contents of a file in any way, including editing the file with Emacs, you must specify the file name.

Consider a file named `` `/usr/rms/foo.c' ``. In Emacs, to begin editing this file, type

```
C-x C-f /usr/rms/foo.c RET
```

Here the file name is given as an argument to the command C-x C-f (`find-file`). That command uses the minibuffer to read the argument, and you type RET to terminate the argument (see section The Minibuffer).

Emacs obeys the command by visiting the file: creating a buffer, copying the contents of the file into the buffer, and then displaying the buffer for you to edit. If you alter the text, you can save the new text in the file by typing C-x C-s

(`save-buffer`). This makes the changes permanent by copying the altered buffer contents back into the file `/usr/rms/foo.c'. Until you save, the changes exist only inside Emacs, and the file `foo.c' is unaltered.

To create a file, just visit the file with C-x C-f as if it already existed. This creates an empty buffer in which you can insert the text you want to put in the file. The file is actually created when you save this buffer with C-x C-s.

Of course, there is a lot more to learn about using files. See section File Handling.

Help

If you forget what a key does, you can find out with the Help character, which is C-h (or F1, which is an alias for C-h). Type C-h k followed by the key you want to know about; for example, C-h k C-n tells you all about what C-n does. C-h is a prefix key; C-h k is just one of its subcommands (the command `describe-key`). The other subcommands of C-h provide different kinds of help. Type C-h twice to get a description of all the help facilities. See section Help.

Blank Lines

Here are special commands and techniques for putting in and taking out blank lines.

C-o
: Insert one or more blank lines after the cursor (`open-line`).

C-x C-o
: Delete all but one of many consecutive blank lines (`delete-blank-lines`).

When you want to insert a new line of text before an existing line, you can do it by typing the new line of text, followed by RET. However, it may be easier to see what you are doing if you first make a blank line and then insert the desired text into it. This is easy to do using the key C-o (`open-line`), which inserts a newline after point but leaves point in front of the newline. After C-o, type the text for the new line. C-o F O O has the same effect as F O O RET, except for the final location of point.

You can make several blank lines by typing C-o several times, or by giving it a numeric argument to tell it how many blank lines to make. See section Numeric Arguments, for how. If you have a fill prefix, then C-o command inserts the fill prefix on the new line, when you use it at the beginning of a line. See section The Fill Prefix.

The easy way to get rid of extra blank lines is with the command C-x C-o (`delete-blank-lines`). C-x C-o in a run of several blank lines deletes all but one of them. C-x C-o on a solitary blank line deletes that blank line. When point is on a nonblank line, C-x C-o deletes any blank lines following that nonblank line.

Continuation Lines

If you add too many characters to one line without breaking it with RET, the line will grow to occupy two (or more) lines on the screen, with a `\' at the extreme right margin of all but the last of them. The `\' says that the following screen line is not really a distinct line in the text, but just the continuation of a line too long to fit the screen. Continuation is also called line wrapping.

Sometimes it is nice to have Emacs insert newlines automatically when a line gets too long. Continuation on the screen does not do that. Use Auto Fill mode (see section Filling Text) if that's what you want.

As an alternative to continuation, Emacs can display long lines by truncation. This means that all the characters that

do not fit in the width of the screen or window do not appear at all. They remain in the buffer, temporarily invis
`$' is used in the last column instead of `\' to inform you that truncation is in effect.

Truncation instead of continuation happens whenever horizontal scrolling is in use, and optionally in all side-by-side windows (see section Multiple Windows). You can enable truncation for a particular buffer by setting the variable `truncate-lines` to non-`nil` in that buffer. (See section Variables.) Altering the value of `truncate-lines` makes it local to the current buffer; until that time, the default value is in effect. The default is initially `nil`. See section Local Variables.

See section Variables Controlling Display, for additional variables that affect how text is displayed.

Cursor Position Information

Here are commands to get information about the size and position of parts of the buffer, and to count lines.

M-x what-page
: Print page number of point, and line number within page.

M-x what-line
: Print line number of point in the buffer.

M-x line-number-mode
: Toggle automatic display of current line number.

M-=
: Print number of lines in the current region (`count-lines-region`). See section The Mark and the Region, for information about the region.

C-x =
: Print character code of character after point, character position of point, and column of point (`what-cursor-position`).

There are two commands for working with line numbers. M-x what-line computes the current line number and displays it in the echo area. To go to a given line by number, use M-x goto-line; it prompts you for the number. These line numbers count from one at the beginning of the buffer.

You can also see the current line number in the mode line; See section The Mode Line. If you narrow the buffer, then the line number in the mode line is relative to the accessible portion (see section Narrowing). By contrast, `what-line` shows both the line number relative to the narrowed region and the line number relative to the whole buffer.

By contrast, M-x what-page counts pages from the beginning of the file, and counts lines within the page, printing both numbers. See section Pages.

While on this subject, we might as well mention M-= (`count-lines-region`), which prints the number of lines in the region (see section The Mark and the Region). See section Pages, for the command C-x l which counts the lines in the current page.

The command C-x = (`what-cursor-position`) can be used to find out the column that the cursor is in, and other miscellaneous information about point. It prints a line in the echo area that looks like this:

```
Char: c (0143, 99, 0x63)  point=21044 of 26883(78%)  column 53
```

(In fact, this is the output produced when point is before the `column' in the example.)

The two values after `Char:' describe the character that follows point, first by showing it and second by giving its octal character code.

`point=' is followed by the position of point expressed as a character count. The front of the buffer counts as position 1, one character later as 2, and so on. The next, larger number is the total number of characters in the buffer. Afterward in parentheses comes the position expressed as a percentage of the total size.

`column' is followed by the horizontal position of point, in columns from the left edge of the window.

If the buffer has been narrowed, making some of the text at the beginning and the end temporarily inaccessible, C-x = prints additional text describing the currently accessible range. For example, it might display this:

```
Char: C (0103, 67, 0x43)  point=22015 of 26889(82%) <21660 - 22099>  column 0
```

where the two extra numbers give the smallest and largest character position that point is allowed to assume. The characters between those two positions are the accessible ones. See section Narrowing.

If point is at the end of the buffer (or the end of the accessible part), C-x = omits any description of the character after point. The output might look like this:

```
point=26957 of 26956(100%)  column 0
```

Numeric Arguments

In mathematics and computer usage, the word argument means "data provided to a function or operation." You can give any Emacs command a numeric argument (also called a prefix argument). Some commands interpret the argument as a repetition count. For example, C-f with an argument of ten moves forward ten characters instead of one. With these commands, no argument is equivalent to an argument of one. Negative arguments tell most such commands to move or act in the opposite direction.

If your terminal keyboard has a META key, the easiest way to specify a numeric argument is to type digits and/or a minus sign while holding down the the META key. For example,

```
M-5 C-n
```

would move down five lines. The characters Meta-1, Meta-2, and so on, as well as Meta--, do this because they are keys bound to commands (`digit-argument` and `negative-argument`) that are defined to contribute to an argument for the next command. Digits and - modified with Control, or Control and Meta, also specify numeric arguments.

Another way of specifying an argument is to use the C-u (`universal-argument`) command followed by the digits of the argument. With C-u, you can type the argument digits without holding down modifier keys; C-u works on all terminals. To type a negative argument, type a minus sign after C-u. Just a minus sign without digits normally means -1.

C-u followed by a character which is neither a digit nor a minus sign has the special meaning of "multiply by four". It multiplies the argument for the next command by four. C-u twice multiplies it by sixteen. Thus, C-u C-u C-f moves forward sixteen characters. This is a good way to move forward "fast", since it moves about 1/5 of a line in the usual size screen. Other useful combinations are C-u C-n, C-u C-u C-n (move down a good fraction of a screen), C-u C-u C-o (make "a lot" of blank lines), and C-u C-k (kill four lines).

Some commands care only about whether there is an argument, and not about its value. For example, the command

TOC

M-q (`fill-paragraph`) with no argument fills text; with an argument, it justifies the text as well. (See secti
Filling Text, for more information on M-q.) Plain C-u is a handy way of providing an argument for such comm

Some commands use the value of the argument as a repeat count, but do something peculiar when there is no argument. For example, the command C-k (`kill-line`) with argument n kills n lines, including their terminating newlines. But C-k with no argument is special: it kills the text up to the next newline, or, if point is right at the end of the line, it kills the newline itself. Thus, two C-k commands with no arguments can kill a nonblank line, just like C-k with an argument of one. (See section Deletion and Killing, for more information on C-k.)

A few commands treat a plain C-u differently from an ordinary argument. A few others may treat an argument of just a minus sign differently from an argument of -1. These unusual cases are described when they come up; they are always for reasons of convenience of use of the individual command.

You can use a numeric argument to insert multiple copies of a character. This is straightforward unless the character is a digit; for example, C-u 6 4 a inserts 64 copies of the character `a'. But this does not work for inserting digits; C-u 6 4 1 specifies an argument of 641, rather than inserting anything. To separate the digit to insert from the argument, type another C-u; for example, C-u 6 4 C-u 1 does insert 64 copies of the character `1'.

We use the term "prefix argument" as well as "numeric argument" to emphasize that you type the argument before the command, and to distinguish these arguments from minibuffer arguments that come after the command.

Go to the previous, next section.

Go to the previous, next section.

The Minibuffer

The minibuffer is the facility used by Emacs commands to read arguments more complicated than a single number. Minibuffer arguments can be file names, buffer names, Lisp function names, Emacs command names, Lisp expressions, and many other things, depending on the command reading the argument. You can use the usual Emacs editing commands in the minibuffer to edit the argument text.

When the minibuffer is in use, it appears in the echo area, and the terminal's cursor moves there. The beginning of the minibuffer line displays a prompt which says what kind of input you should supply and how it will be used. Often this prompt is derived from the name of the command that the argument is for. The prompt normally ends with a colon.

Sometimes a default argument appears in parentheses after the colon; it too is part of the prompt. The default will be used as the argument value if you enter an empty argument (e.g., just type RET). For example, commands that read buffer names always show a default, which is the name of the buffer that will be used if you type just RET.

The simplest way to enter a minibuffer argument is to type the text you want, terminated by RET which exits the minibuffer. You can cancel the command that wants the argument, and get out of the minibuffer, by typing C-g.

Since the minibuffer uses the screen space of the echo area, it can conflict with other ways Emacs customarily uses the echo area. Here is how Emacs handles such conflicts:

- If a command gets an error while you are in the minibuffer, this does not cancel the minibuffer. However, the echo area is needed for the error message and therefore the minibuffer itself is hidden for a while. It comes back after a few seconds, or as soon as you type anything.
- If in the minibuffer you use a command whose purpose is to print a message in the echo area, such as C-x =, the message is printed normally, and the minibuffer is hidden for a while. It comes back after a few seconds, or as soon as you type anything.
- Echoing of keystrokes does not take place while the minibuffer is in use.

Minibuffers for File Names

Sometimes the minibuffer starts out with text in it. For example, when you are supposed to give a file name, the minibuffer starts out containing the default directory, which ends with a slash. This is to inform you which directory the file will be found in if you do not specify a directory.

For example, the minibuffer might start out with these contents:

```
Find File: /u2/emacs/src/
```

where `Find File: ' is the prompt. Typing buffer.c specifies the file `/u2/emacs/src/buffer.c'.

To find files in nearby directories, use ..; thus, if you type ../lisp/simple.el, you will get the file name `/u2/emacs/lisp/simple.el'. Alternatively, you can kill with M-DEL the directory names you don't want (see section Words).

If you don't want any of the default, you can kill it with C-a C-k. But you don't need to kill the default; you can simply ignore it. Insert an absolute file name, one starting with a slash or a tilde, after the default directory. For example, to specify the file `/etc/termcap', just insert that name, giving these minibuffer contents:

```
Find File: /u2/emacs/src//etc/termcap
```

Two slashes in a row are not normally meaningful in a file name, but they are allowed in GNU Emacs. They mean, "ignore everything before the second slash in the pair." Thus, `/u2/emacs/src/' is ignored in the example above, and you get the file `/etc/termcap'.

If you set `insert-default-directory` to `nil`, the default directory is not inserted in the minibuffer. This way, the minibuffer starts out empty. But the name you type, if relative, is still interpreted with respect to the same default directory.

Editing in the Minibuffer

The minibuffer is an Emacs buffer (albeit a peculiar one), and the usual Emacs commands are available for editing the text of an argument you are entering.

Since RET in the minibuffer is defined to exit the minibuffer, you can't use it to insert a newline in the minibuffer. To do that, type C-o or C-q LFD. (Recall that a newline is really the LFD character.)

The minibuffer has its own window which always has space on the screen but acts as if it were not there when the minibuffer is not in use. When the minibuffer is in use, its window is just like the others; you can switch to another window with C-x o, edit text in other windows and perhaps even visit more files, before returning to the minibuffer to submit the argument. You can kill text in another window, return to the minibuffer window, and then yank the text to use it in the argument. See section Multiple Windows.

There are some restrictions on the use of the minibuffer window, however. You cannot switch buffers in it--the minibuffer and its window are permanently attached. Also, you cannot split or kill the minibuffer window. But you can make it taller in the normal fashion with C-x ^. If you enable Resize-Minibuffer mode, then the minibuffer window expands vertically as necessary to hold the text that you put in the minibuffer. Use M-x resize-minibuffer-mode to enable or disable this minor mode (see section Minor Modes).

If while in the minibuffer you issue a command that displays help text of any sort in another window, you can use the C-M-v command while in the minibuffer to scroll the help text. This lasts until you exit the minibuffer. This feature is especially useful if a completing minibuffer gives you a list of possible completions. See section Using Other Windows.

Emacs normally disallows most commands that use the minibuffer while the minibuffer is selected. This rule is to prevent recursive minibuffers from confusing novice users. If you want to be able to use such

commands in the minibuffer, set the variable `enable-recursive-minibuffers` to a non-`nil` value.

Completion

For certain kinds of arguments, you can use completion to enter the argument value. Completion means that you type part of the argument, then Emacs visibly fills in the rest, or as much as can be determined from the part you have typed.

When completion is available, certain keys---TAB, RET, and SPC---are rebound to complete the text present in the minibuffer into a longer string that it stands for, by matching it against a set of completion alternatives provided by the command reading the argument. ? is defined to display a list of possible completions of what you have inserted.

For example, when M-x uses the minibuffer to read the name of a command, it provides a list of all available Emacs command names to complete against. The completion keys match the text in the minibuffer against all the command names, find any additional name characters implied by the ones already present in the minibuffer, and add those characters to the ones you have given. This is what makes it possible to type M-x ins SPC b RET instead of M-x insert-buffer RET (for example).

Case is normally significant in completion, because it is significant in most of the names that you can complete (buffer names, file names and command names). Thus, `fo' does not complete to `Foo'. Completion does ignore case distinctions for certain arguments in which case does not matter.

Completion Example

A concrete example may help here. If you type M-x au TAB, the TAB looks for alternatives (in this case, command names) that start with `au'. There are only two: `auto-fill-mode` and `auto-save-mode`. These are the same as far as `auto-`, so the `au' in the minibuffer changes to `auto-'.

If you type TAB again immediately, there are multiple possibilities for the very next character--it could be `s' or `f'---so no more characters are added; instead, TAB displays a list of all possible completions in another window.

If you go on to type f TAB, this TAB sees `auto-f'. The only command name starting this way is `auto-fill-mode`, so completion fills in the rest of that. You now have `auto-fill-mode' in the minibuffer after typing just au TAB f TAB. Note that TAB has this effect because in the minibuffer it is bound to the command `minibuffer-complete` when completion is available.

Completion Commands

Here is a list of the completion commands defined in the minibuffer when completion is available.

TAB

Complete the text in the minibuffer as much as possible (`minibuffer-complete`).

SPC

Complete the minibuffer text, but don't go beyond one word (`minibuffer-complete-wo`

RET

Submit the text in the minibuffer as the argument, possibly completing first as described below (`minibuffer-complete-and-exit`).

?

Print a list of all possible completions of the text in the minibuffer (`minibuffer-list-completions`).

SPC completes much like TAB, but never goes beyond the next hyphen or space. If you have `auto-f' in the minibuffer and type SPC, it finds that the completion is `auto-fill-mode', but it stops completing after `fill-'. This gives `auto-fill-'. Another SPC at this point completes all the way to `auto-fill-mode'. SPC in the minibuffer when completion is available runs the command `minibuffer-complete-word`.

Here are some commands you can use to choose a completion from a window that displays a list of completions:

Mouse-2

Clicking mouse button 2 on a completion in the list of possible completions chooses that completion (`mouse-choose-completion`). You normally use this command while point is in the minibuffer; but you must click in the list of completions, not in the minibuffer itself.

PRIOR

M-v

Typing PRIOR or PAGE-UP, or M-v, while in the minibuffer, selects the window showing the completion list buffer (`switch-to-completions`). This paves the way for using the commands below. (Selecting that window in the usual ways has the same effect, but this way is more convenient.)

RET

Typing RET *in the completion list buffer* chooses the completion that point is in or next to (`choose-completion`). To use this command, you must first switch windows to the window that shows the list of completions.

RIGHT

Typing the right-arrow key RIGHT *in the completion list buffer* moves point to the following completion (`next-completion`).

LEFT

Typing the left-arrow key LEFT *in the completion list buffer* moves point toward the beginning of the buffer, to the previous completion (`previous-completion`).

Strict Completion

There are three different ways that RET can work in completing minibuffers, depending on how the argument will be used.

- Strict completion is used when it is meaningless to give any argument except one of the known alternatives. For example, when C-x k reads the name of a buffer to kill, it is meaningless to give

anything but the name of an existing buffer. In strict completion, RET refuses to exit if the text in the minibuffer does not complete to an exact match.

- Cautious completion is similar to strict completion, except that RET exits only if the text was an exact match already, not needing completion. If the text is not an exact match, RET does not exit, but it does complete the text. If it completes to an exact match, a second RET will exit.

  Cautious completion is used for reading file names for files that must already exist.

- Permissive completion is used when any string whatever is meaningful, and the list of completion alternatives is just a guide. For example, when C-x C-f reads the name of a file to visit, any file name is allowed, in case you want to create a file. In permissive completion, RET takes the text in the minibuffer exactly as given, without completing it.

The completion commands display a list of all possible completions in a window whenever there is more than one possibility for the very next character. Also, typing ? explicitly requests such a list. If the list of completions is long, you can scroll it with C-M-v (see section Using Other Windows).

Completion Options

When completion is done on file names, certain file names are usually ignored. The variable `completion-ignored-extensions` contains a list of strings; a file whose name ends in any of those strings is ignored as a possible completion. The standard value of this variable has several elements including `".o"`, `".elc"`, `".dvi"` and `"~"`. The effect is that, for example, `foo' can complete to `foo.c' even though `foo.o' exists as well. However, if *all* the possible completions end in "ignored" strings, then they are not ignored. Ignored extensions do not apply to lists of completions--those always mention all possible completions.

Normally, a completion command that finds the next character is undetermined automatically displays a list of all possible completions. If the variable `completion-auto-help` is set to `nil`, this does not happen, and you must type ? to display the possible completions.

The `complete` library implements a more powerful kind of completion that can complete multiple words at a time. For example, it can complete the command name abbreviation `p-b` into `print-buffer`, because no other command starts with two words whose initials are `p' and `b'. To use this library, put `(load "complete")` in your `` `~/.emacs' `` file (see section The Init File, `` `~/.emacs' ``).

Icomplete mode presents a constantly-updated display that tells you what completions are available for the text you've entered so far. The command to enable or disable this minor mode is M-x icomplete-mode.

Minibuffer History

Every argument that you enter with the minibuffer is saved on a minibuffer history list so that you can use it again later in another argument. Special commands load the text of an earlier argument in the minibuffer. They discard the old minibuffer contents, so you can think of them as moving through the history of previous arguments.

M-p

Move to the next earlier argument string saved in the minibuffer history (`previous-history-element`).

M-n

Move to the next later argument string saved in the minibuffer history (`next-history-element`).

M-r regexp RET

Move to an earlier saved argument in the minibuffer history that has a match for regexp (`previous-matching-history-element`).

M-s regexp RET

Move to a later saved argument in the minibuffer history that has a match for regexp (`next-matching-history-element`).

The simplest way to reuse the saved arguments in the history list is to move through the history list one element at a time. While in the minibuffer, type M-p (`previous-history-element`) to "move to" the next earlier minibuffer input, and use M-n (`next-history-element`) to "move to" the next later input.

The previous input that you fetch from the history entirely replaces the contents of the minibuffer. To use it as the argument, exit the minibuffer as usual with RET. You can also edit the text before you reuse it; this does not change the history element that you "moved" to, but your new argument does go at the end of the history list in its own right.

There are also commands to search forward or backward through the history. As of this writing, they search for history elements that match a regular expression that you specify with the minibuffer. M-r (`previous-matching-history-element`) searches older elements in the history, while M-s (`next-matching-history-element`) searches newer elements. By special dispensation, these commands can use the minibuffer to read their arguments even though you are already in the minibuffer when you issue them.

All uses of the minibuffer record your input on a history list, but there are separate history lists for different kinds of arguments. For example, there is a list for file names, used by all the commands that read file names. There is a list for arguments of commands like `query-replace`. There are several very specific history lists, including one for command names read by M-x and one for compilation commands read by `compile`. Finally, there is one "miscellaneous" history list that most minibuffer arguments use.

Repeating Minibuffer Commands

Every command that uses the minibuffer at least once is recorded on a special history list, together with the values of its arguments, so that you can repeat the entire command. In particular, every use of M-x is recorded there, since M-x uses the minibuffer to read the command name.

C-x ESC ESC

Re-execute a recent minibuffer command (`repeat-complex-command`).

M-x list-command-history

> Display the entire command history, showing all the commands C-x ESC ESC can repeat, most recent first.

C-x ESC ESC is used to re-execute a recent minibuffer-using command. With no argument, it repeats the last such command. A numeric argument specifies which command to repeat; one means the last one, and larger numbers specify earlier ones.

C-x ESC ESC works by turning the previous command into a Lisp expression and then entering a minibuffer initialized with the text for that expression. If you type just RET, the command is repeated as before. You can also change the command by editing the Lisp expression. Whatever expression you finally submit is what will be executed. The repeated command is added to the front of the command history unless it is identical to the most recently executed command already there.

Even if you don't understand Lisp syntax, it will probably be obvious which command is displayed for repetition. If you do not change the text, it will repeat exactly as before.

Once inside the minibuffer for C-x ESC ESC, you can use the minibuffer history commands (M-p, M-n, M-r, M-s; see section Minibuffer History) to move through the history list of saved entire commands. After finding the desired previous command, you can edit its expression as usual and then resubmit it by typing RET as usual.

The list of previous minibuffer-using commands is stored as a Lisp list in the variable `command-history`. Each element is a Lisp expression which describes one command and its arguments. Lisp programs can reexecute a command by calling `eval` with the `command-history` element.

Go to the previous, next section.

Go to the previous, next section.

Running Commands by Name

The Emacs commands that are used often or that must be quick to type are bound to keys--short sequences of characters--for convenient use. Other Emacs commands that do not need to be brief are not bound to keys; to run them, you must refer to them by name.

A command name is, by convention, made up of one or more words, separated by hyphens; for example, `auto-fill-mode` or `manual-entry`. The use of English words makes the command name easier to remember than a key made up of obscure characters, even though it is more characters to type.

The way to run a command by name is to start with M-x, type the command name, and finish it with RET. M-x uses the minibuffer to read the command name. RET exits the minibuffer and runs the command. The string `M-x' appears at the beginning of the minibuffer as a prompt to remind you to enter the name of a command to be run. See section The Minibuffer, for full information on the features of the minibuffer.

You can use completion to enter the command name. For example, the command `forward-char` can be invoked by name by typing

```
M-x forward-char RET
```

or

```
M-x forw TAB c RET
```

Note that `forward-char` is the same command that you invoke with the key C-f. You can run any Emacs command by name using M-x, whether or not any keys are bound to it. If you use M-x to run a command which also has a key binding, it displays a message to tell you about the key binding, before running the command. (You can turn off this notification feature by setting the variable `suggest-key-bindings` to `nil`.)

If you type C-g while the command name is being read, you cancel the M-x command and get out of the minibuffer, ending up at top level.

To pass a numeric argument to the command you are invoking with M-x, specify the numeric argument before the M-x. M-x passes the argument along to the command it runs. The argument value appears in the prompt while the command name is being read.

If the command you type has a key binding of its own, Emacs mentions this in the echo area before it runs the command. For example, if you type M-x forward-word, the message says that you can run the same command more easily by typing M-f. You can turn off these messages by setting `suggest-key-bindings` to `nil`. If `suggest-key-bindings` is a number, it says how long to show the message before proceeding with the command.

Normally, when describing a command that is run by name, we omit the RET that is needed to terminate

TOC

the name. Thus we might speak of M-x auto-fill-mode rather than M-x auto-fill-mode RET. We men the RET only when there is a need to emphasize its presence, such as when we show the command together with following arguments.

M-x works by running the command `execute-extended-command`, which is responsible for reading the name of another command and invoking it.

Go to the previous, next section.

Go to the previous, next section.

Help

Emacs provides extensive help features accessible through a single character, C-h. C-h is a prefix key that is used only for documentation-printing commands. The characters that you can type after C-h are called help options. One help option is C-h; that is how you ask for help about using C-h. To cancel, type C-g. The function key F1 is equivalent to C-h.

C-h C-h (`help-for-help`) displays a list of the possible help options, each with a brief description. Before you type a help option, you can use SPC or DEL to scroll through the list.

C-h or F1 means "help" in various other contexts as well. For example, in `query-replace`, it describes the options available. After a prefix key, it displays a list of the alternatives that can follow the prefix key. (A few prefix keys don't support this because they define other meanings for C-h.)

Most help buffers use a special major mode, Help mode, which lets you scroll conveniently with SPC and DEL.

Here is a summary of the defined help commands.

C-h a regexp RET

: Display list of commands whose names match regexp (`apropos-command`).

C-h b

: Display a table of all key bindings in effect now, in this order: minor mode bindings, major mode bindings, and global bindings (`describe-bindings`).

C-h c key

: Print the name of the command that key runs (`describe-key-briefly`). Here c stands for `character'. For more extensive information on key, use C-h k.

C-h f function RET

: Display documentation on the Lisp function named function (`describe-function`). Since commands are Lisp functions, a command name may be used.

C-h i

: Run Info, the program for browsing documentation files (`info`). The complete Emacs manual is available on-line in Info.

C-h k key

: Display name and documentation of the command that key runs (`describe-key`).

C-h l

: Display a description of the last 100 characters you typed (`view-lossage`).

C-h m

: Display documentation of the current major mode (`describe-mode`).

C-h n

Display documentation of Emacs changes, most recent first (`view-emacs-news`).

C-h p

Find packages by topic keyword (`finder-by-keyword`).

C-h s

Display current contents of the syntax table, plus an explanation of what they mean (`describe-syntax`). See section The Syntax Table.

C-h t

Enter the Emacs interactive tutorial (`help-with-tutorial`).

C-h v var RET

Display the documentation of the Lisp variable var (`describe-variable`).

C-h w command RET

Print which keys run the command named command (`where-is`).

C-h C-f function RET

Enter Info and go to the node documenting the Emacs function function (`Info-goto-emacs-command-node`).

C-h C-k key

Enter Info and go to the node where the key sequence key is documented (`Info-goto-emacs-key-command-node`).

C-h C-c

Display the copying conditions for GNU Emacs.

C-h C-d

Display information about getting new versions of GNU Emacs.

C-h C-p

Display information about the GNU Project.

Documentation for a Key

The most basic C-h options are C-h c (`describe-key-briefly`) and C-h k (`describe-key`). C-h c key prints in the echo area the name of the command that key is bound to. For example, C-h c C-f prints `forward-char'. Since command names are chosen to describe what the commands do, this is a good way to get a very brief description of what key does.

C-h k key is similar but gives more information: it displays the documentation string of the command as well as its name. This is too big for the echo area, so a window is used for the display.

C-h c and C-h k work for any sort of key sequences, including function keys and mouse events.

Help by Command or Variable Name

C-h f (`describe-function`) reads the name of a Lisp function using the minibuffer, then displays that function's documentation string in a window. Since commands are Lisp functions, you can use this to get the documentation of a command that you know by name. For example,

```
C-h f auto-fill-mode RET
```

displays the documentation of `auto-fill-mode`. This is the only way to get the documentation of a command that is not bound to any key (one which you would normally run using M-x).

C-h f is also useful for Lisp functions that you are planning to use in a Lisp program. For example, if you have just written the expression `(make-vector len)` and want to check that you are using `make-vector` properly, type C-h f make-vector RET. Because C-h f allows all function names, not just command names, you may find that some of your favorite abbreviations that work in M-x don't work in C-h f. An abbreviation may be unique among command names yet fail to be unique when other function names are allowed.

The function name for C-h f to describe has a default which is used if you type RET leaving the minibuffer empty. The default is the function called by the innermost Lisp expression in the buffer around point, *provided* that is a valid, defined Lisp function name. For example, if point is located following the text `(make-vector (car x)', the innermost list containing point is the one that starts with `(make-vector', so the default is to describe the function `make-vector`.

C-h f is often useful just to verify that you have the right spelling for the function name. If C-h f mentions a name from the buffer as the default, that name must be defined as a Lisp function. If that is all you want to know, just type C-g to cancel the C-h f command, then go on editing.

C-h w command RET tells you what keys are bound to command. It prints a list of the keys in the echo area. If it says the command is not on any key, you must use M-x to run it. C-h w runs the command `where-is`.

C-h v (`describe-variable`) is like C-h f but describes Lisp variables instead of Lisp functions. Its default is the Lisp symbol around or before point, but only if that is the name of a known Lisp variable. See section Variables.

Apropos

A more sophisticated sort of question to ask is, "What are the commands for working with files?" To ask this question, type C-h a file RET, which displays a list of all command names that contain `file', including `copy-file`, `find-file`, and so on. With each command name appears a brief description of how to use the command, and what keys you can currently invoke it with. For example, it would say that you can invoke `find-file` by typing C-x C-f. The a in C-h a stands for `Apropos'; C-h a runs the command `apropos-command`. This command does not check user variables by default; specify a numeric argument if you want it to check them.

Because C-h a looks only for functions whose names contain the string which you specify, you must ingenuity in choosing the string. If you are looking for commands for killing backwards and C-h a kill-backwards RET doesn't reveal any, don't give up. Try just kill, or just backwards, or just back. Be persistent. Also note that you can use a regular expression as the argument, for more flexibility (see section Syntax of Regular Expressions).

Here is a set of arguments to give to C-h a that covers many classes of Emacs commands, since there are strong conventions for naming the standard Emacs commands. By giving you a feel for the naming conventions, this set should also serve to aid you in developing a technique for picking `apropos` strings.

> char, line, word, sentence, paragraph, region, page, sexp, list, defun, rect, buffer, frame, window, face, file, dir, register, mode, beginning, end, forward, backward, next, previous, up, down, search, goto, kill, delete, mark, insert, yank, fill, indent, case, change, set, what, list, find, view, describe, default.

To list all Lisp symbols that contain a match for a regexp, not just the ones that are defined as commands, use the command M-x apropos instead of C-h a. This command does not check key bindings by default; specify a numeric argument if you want it to check them.

The `apropos-documentation` command is like `apropos` except that it searches documentation strings as well as symbol names for matches for the specified regular expression.

The `apropos-value` command is like `apropos` except that it searches symbols' values for matches for the specified regular expression. This command does not check function definitions or property lists by default; specify a numeric argument if you want it to check them.

If you want more information about a function definition, variable or symbol property listed in the Apropos buffer, you can click on it with Mouse-2 or move there and type RET.

Keyword Search for Lisp Libraries

The C-h p command lets you search the standard Emacs Lisp libraries by topic keywords. Here is a partial list of keywords you can use:

`abbrev'

> Abbreviation handling, typing shortcuts, macros.

`bib'

> Support for the bibliography processor `bib`.

`c'

> C and C++ language support.

`calendar'

> Calendar and time management support.

`comm'

> Communications, networking, remote access to files.

`data'
: Support for editing files of data.

`docs'
: Support for Emacs documentation.

`emulations'
: Emulations of other editors.

`extensions'
: Emacs Lisp language extensions.

`faces'
: Support for using faces (fonts and colors; see section Using Multiple Typefaces).

`frames'
: Support for Emacs frames and window systems.

`games'
: Games, jokes and amusements.

`hardware'
: Support for interfacing with exotic hardware.

`help'
: Support for on-line help systems.

`hypermedia'
: Support for links within text, or other media types.

`i18n'
: Internationalization and alternate character-set support.

`internal'
: Code for Emacs internals, build process, defaults.

`languages'
: Specialized modes for editing programming languages.

`lisp'
: Support for using Lisp (including Emacs Lisp).

`local'
: Libraries local to your site.

`maint'
: Maintenance aids for the Emacs development group.

`mail'
: Modes for electronic-mail handling.

`matching'
: Searching and matching.

`news'
: Support for netnews reading and posting.

`non-text'
: Support for editing files that are not ordinary text.

`oop'
: Support for object-oriented programming.

`outlines'
: Hierarchical outlining.

`processes'
: Process, subshell, compilation, and job control support.

`terminals'
: Support for terminal types.

`tex'
: Support for the TeX formatter.

`tools'
: Programming tools.

`unix'
: Front-ends/assistants for, or emulators of, Unix features.

`vms'
: Support code for VMS.

`wp'
: Word processing.

Other Help Commands

C-h i (`info`) runs the Info program, which is used for browsing through structured documentation files. The entire Emacs manual is available within Info. Eventually all the documentation of the GNU system will be available. Type h after entering Info to run a tutorial on using Info.

There are two special help commands for accessing Emacs documentation through Info. C-h C-f function RET enters Info and goes straight to the documentation of the Emacs function function. C-h C-k key enters Info and goes straight to the documentation of the key key. These two keys run the commands `Info-goto-emacs-command-node` and `Info-goto-emacs-key-command-node`.

If something surprising happens, and you are not sure what commands you typed, use C-h l (`view-lossage`). C-h l prints the last 100 command characters you typed in. If you see commands that you don't know, you can use C-h c to find out what they do.

Emacs has numerous major modes, each of which redefines a few keys and makes a few other changes in how editing works. C-h m (`describe-mode`) prints documentation on the current major mode, which normally describes all the commands that are changed in this mode.

C-h b (`describe-bindings`) and C-h s (`describe-syntax`) present other information about the current Emacs mode. C-h b displays a list of all the key bindings now in effect; the local bindings defined by the current minor modes first, then the local bindings defined by the current major mode, and finally the global bindings (see section Customizing Key Bindings). C-h s displays the contents of the syntax table, with explanations of each character's syntax (see section The Syntax Table).

You can get a similar list for a particular prefix key by typing C-h after the prefix key. (There are a few prefix keys for which this does not work--those that provide their own bindings for C-h. One of these is ESC, because ESC C-h is actually C-M-h, which marks a defun.)

The other C-h options display various files of useful information. C-h C-w displays the full details on the complete absence of warranty for GNU Emacs. C-h n (`view-emacs-news`) displays the file `` `emacs/etc/NEWS' ``, which contains documentation on Emacs changes arranged chronologically. C-h t (`help-with-tutorial`) displays the learn-by-doing Emacs tutorial. C-h C-c (`describe-copying`) displays the file `` `emacs/etc/COPYING' ``, which tells you the conditions you must obey in distributing copies of Emacs. C-h C-d (`describe-distribution`) displays the file `` `emacs/etc/DISTRIB' ``, which tells you how you can order a copy of the latest version of Emacs. C-h C-p (`describe-project`) displays general information about the GNU Project.

Go to the previous, next section.

Go to the previous, next section.

The Mark and the Region

Many Emacs commands operate on an arbitrary contiguous part of the current buffer. To specify the text for such a command to operate on, you set the mark at one end of it, and move point to the other end. The text between point and the mark is called the region. Emacs highlights the region whenever there is one, if you enable Transient Mark mode (see section Transient Mark Mode).

You can move point or the mark to adjust the boundaries of the region. It doesn't matter which one is set first chronologically, or which one comes earlier in the text. Once the mark has been set, it remains where you put it until you set it again at another place. Each Emacs buffer has its own mark, so that when you return to a buffer that had been selected previously, it has the same mark it had before.

Many commands that insert text, such as C-y (`yank`) and M-x insert-buffer, position point and the mark at opposite ends of the inserted text, so that the region contains the text just inserted.

Aside from delimiting the region, the mark is also useful for remembering a spot that you may want to go back to. To make this feature more useful, each buffer remembers 16 previous locations of the mark in the mark ring.

Setting the Mark

Here are some commands for setting the mark:

C-SPC
: Set the mark where point is (`set-mark-command`).

C-@
: The same.

C-x C-x
: Interchange mark and point (`exchange-point-and-mark`).

Drag-Mouse-1
: Set point and the mark around the text you drag across.

Mouse-3
: Set the mark where point is, then move point to where you click (`mouse-save-then-kill`).

For example, suppose you wish to convert part of the buffer to all upper-case, using the C-x C-u (`upcase-region`) command which operates on the text in the region. You can first go to the beginning of the text to be capitalized, type C-SPC to put the mark there, move to the end, and then type C-x C-u. Or, you can set the mark at the end of the text, move to the beginning, and then type C-x C-u.

The most common way to set the mark is with the C-SPC command (`set-mark-command`). This sets the mark where point is. Then you can move point away, leaving the mark behind.

There are two ways to set the mark with the mouse. You can drag mouse button one across a range of text; that puts point where you release the mouse button, and sets the mark at the other end of that range. Or you can click mouse button three, which sets the mark at point (like C-SPC) and them moves point (like Mouse-1). Both of these methods copy the region into the kill ring in addition to setting the mark; that gives behavior consistent with other window-driven applications, but if you don't want to modify the kill ring, you must use keyboard commands to set the mark. See section Mouse Commands for Editing.

Ordinary terminals have only one cursor, so there is no way for Emacs to show you where the mark is located. You have to remember. The usual solution to this problem is to set the mark and then use it soon, before you forget where it is. Alternatively, you can see where the mark is with the command C-x C-x (`exchange-point-and-mark`) which puts the mark where point was and point where the mark was. The extent of the region is unchanged, but the cursor and point are now at the previous position of the mark. In Transient Mark mode, this command reactivates the mark.

C-x C-x is also useful when you are satisfied with the position of point but want to move the mark; do C-x C-x to put point at that end of the region, and then move it. A second use of C-x C-x, if necessary, puts the mark at the new position with point back at its original position.

There is no such character as C-SPC in ASCII; when you type SPC while holding down CTRL, what you get on most ordinary terminals is the character C-@. This key is actually bound to `set-mark-command`. But unless you are unlucky enough to have a terminal where typing C-SPC does not produce C-@, you might as well think of this character as C-SPC. Under X, C-SPC is actually a distinct character, but its binding is still `set-mark-command`.

Transient Mark Mode

Emacs can highlight the current region, using X Windows. But normally it does not. Why not?

Highlighting the region doesn't work well ordinarily in Emacs, because once you have set a mark, there is *always* a region (in that buffer). And highlighting the region all the time would be a nuisance.

You can turn on region highlighting by enabling Transient Mark mode. This is a more rigid mode of operation in which the region "lasts" only temporarily, so you must set up a region for each command that uses one. In Transient Mark mode, most of the time there is no region; therefore, highlighting the region when it exists is convenient.

To enable Transient Mark mode, type M-x transient-mark-mode. This command toggles the mode, so you can repeat the command to turn off the mode.

Here are the details of Transient Mark mode:

- To set the mark, type C-SPC (`set-mark-command`). This makes the mark active; as you move point, you will see the region highlighting change in extent.
- The mouse commands for specifying the mark also make it active. So do keyboard commands whose purpose is to specify a region, including M-@, C-M-@, M-h, C-M-h, C-x C-p, and C-x h.
- When the mark is active, you can execute commands that operate on the region, such as killing, indentation, or writing to a file.

- Any change to the buffer, such as inserting or deleting a character, deactivates the mark. This means any subsequent command that operates on a region will get an error and refuse to operate. You can make the region active again by typing C-x C-x.
- Commands like M-> and C-s that "leave the mark behind" in addition to some other primary purpose do not activate the new mark. You can activate the new region by executing C-x C-x (`exchange-point-and-mark`).
- C-s when the mark is active does not alter the mark.
- Quitting with C-g deactivates the mark.

Transient Mark mode is also sometimes known as "Zmacs mode" because the Zmacs editor on the MIT Lisp Machine handled the mark in a similar way.

When multiple windows show the same buffer, they can have different regions, because they can have different values of point (though they all share common one mark position). In Transient Mark mode, each window highlights its own region. The part that is highlighted in the selected window is the region that editing commands use. See section Multiple Windows.

When Transient Mark mode is not enabled, every command that sets the mark also activates it, and nothing ever deactivates it.

If the variable `mark-even-if-inactive` is non-`nil` in Transient Mark mode, then commands can use the mark and the region even when it is inactive. Region highlighting appears and disappears just as it normally does in Transient Mark mode, but the mark doesn't really go away when the highlighting disappears.

Operating on the Region

Once you have a region and the mark is active, here are some of the ways you can operate on the region:

- Kill it with C-w (see section Deletion and Killing).
- Save it in a register with C-x r s (see section Registers).
- Save it in a buffer or a file (see section Accumulating Text).
- Convert case with C-x C-l or C-x C-u (see section Case Conversion Commands).
- Indent it with C-x TAB or C-M-\ (see section Indentation).
- Fill it as text with M-x fill-region (see section Filling Text).
- Print hardcopy with M-x print-region (see section Hardcopy Output).
- Evaluate it as Lisp code with M-x eval-region (see section Evaluating Emacs-Lisp Expressions).

Most commands that operate on the text in the region have the word `region` in their names.

Commands to Mark Textual Objects

Here are the commands for placing point and the mark around a textual object such as a word, list, paragraph or page.

M-@
: Set mark after end of next word (`mark-word`). This command and the following one do not move point.

C-M-@
: Set mark after end of next Lisp expression (`mark-sexp`).

M-h
: Put region around current paragraph (`mark-paragraph`).

C-M-h
: Put region around current Lisp defun (`mark-defun`).

C-x h
: Put region around entire buffer (`mark-whole-buffer`).

C-x C-p
: Put region around current page (`mark-page`).

M-@ (`mark-word`) puts the mark at the end of the next word, while C-M-@ (`mark-sexp`) puts it at the end of the next Lisp expression. These commands handle arguments just like M-f and C-M-f.

Other commands set both point and mark, to delimit an object in the buffer. For example, M-h (`mark-paragraph`) moves point to the beginning of the paragraph that surrounds or follows point, and puts the mark at the end of that paragraph (see section Paragraphs). It prepares the region so you can indent, case-convert, or kill a whole paragraph.

C-M-h (`mark-defun`) similarly puts point before and the mark after the current or following defun (see section Defuns). C-x C-p (`mark-page`) puts point before the current page, and mark at the end (see section Pages). The mark goes after the terminating page delimiter (to include it), while point goes after the preceding page delimiter (to exclude it). A numeric argument specifies a later page (if positive) or an earlier page (if negative) instead of the current page.

Finally, C-x h (`mark-whole-buffer`) sets up the entire buffer as the region, by putting point at the beginning and the mark at the end.

In Transient Mark mode, all of these commands activate the mark.

The Mark Ring

Aside from delimiting the region, the mark is also useful for remembering a spot that you may want to go back to. To make this feature more useful, each buffer remembers 16 previous locations of the mark, in the mark ring. Commands that set the mark also push the old mark onto this ring. To return to a marked

location, use C-u C-SPC (or C-u C-@); this is the command `set-mark-command` given a numeric argument. It moves point to where the mark was, and restores the mark from the ring of former marks. Thus, repeated use of this command moves point to all of the old marks on the ring, one by one. The mark positions you move through in this way are not lost; they go to the end of the ring.

Each buffer has its own mark ring. All editing commands use the current buffer's mark ring. In particular, C-u C-SPC always stays in the same buffer.

Many commands that can move long distances, such as M-< (`beginning-of-buffer`), start by setting the mark and saving the old mark on the mark ring. This is to make it easier for you to move back later. Searches set the mark if they move point. You can tell when a command sets the mark because it displays `Mark Set' in the echo area.

If you want to move back to the same place over and over, the mark ring may not be convenient enough. If so, you can record the position in a register for later retrieval (see section Saving Positions in Registers).

The variable `mark-ring-max` specifies the maximum number of entries to keep in the mark ring. If that many entries exist and another one is pushed, the last one in the list is discarded. Repeating C-u C-SPC circulates through the positions currently in the ring.

The variable `mark-ring` holds the mark ring itself, as a list of marker objects in the order most recent first. This variable is local in every buffer.

The Global Mark Ring

In addition to the ordinary mark ring that belongs to each buffer, Emacs has a single global mark ring. It records a sequence of buffers in which you have recently set the mark, so you can go back to those buffers.

Setting the mark always makes an entry on the current buffer's mark ring. If you have switched buffers since the previous mark setting, the new mark position makes an entry on the global mark ring also. The result is that the global mark ring records a sequence of buffers that you have been in, and, for each buffer, a place where you set the mark.

The command C-x C-SPC (`pop-global-mark`) jumps to the buffer and position of the latest entry in the global ring. It also rotates the ring, so that successive uses of C-x C-SPC take you to earlier and earlier buffers.

Go to the previous, next section.

Go to the previous, next section.

Killing and Moving Text

Killing means erasing text and copying it into the kill ring, from which it can be retrieved by yanking it. Some systems use the terms "cutting" and "pasting" for these operations.

The commonest way of moving or copying text within Emacs is to kill it and later yank elsewhere it in one or more places. This is very safe because Emacs remembers several recent kills, not just the last one. It is versatile, because the many commands for killing syntactic units can also be used for moving those units. But there are other ways of copying text for special purposes.

Emacs has only one kill ring for all buffers, so you can kill text in one buffer and yank it in another buffer.

Deletion and Killing

Most commands which erase text from the buffer save it in the kill ring so that you can move or copy it to other parts of the buffer. These commands are known as kill commands. The rest of the commands that erase text do not save it in the kill ring; they are known as delete commands. (This distinction is made only for erasure of text in the buffer.) If you do a kill or delete command by mistake, you can use the C-x u (`undo`) command to undo it (see section Undoing Changes).

The delete commands include C-d (`delete-char`) and DEL (`delete-backward-char`), which delete only one character at a time, and those commands that delete only spaces or newlines. Commands that can destroy significant amounts of nontrivial data generally kill. The commands' names and individual descriptions use the words `kill' and `delete' to say which they do.

Deletion

C-d
: Delete next character (`delete-char`).

DEL
: Delete previous character (`delete-backward-char`).

M-\
: Delete spaces and tabs around point (`delete-horizontal-space`).

M-SPC
: Delete spaces and tabs around point, leaving one space (`just-one-space`).

C-x C-o
: Delete blank lines around the current line (`delete-blank-lines`).

M-^

Join two lines by deleting the intervening newline, along with any indentation following it (`delete-indentation`).

The most basic delete commands are C-d (`delete-char`) and DEL (`delete-backward-char`). C-d deletes the character after point, the one the cursor is "on top of". This doesn't move point. DEL deletes the character before the cursor, and moves point back. You can delete newlines like any other characters in the buffer; deleting a newline joins two lines. Actually, C-d and DEL aren't always delete commands; when given arguments, they kill instead, since they can erase more than one character this way.

The other delete commands are those which delete only whitespace characters: spaces, tabs and newlines. M-\ (`delete-horizontal-space`) deletes all the spaces and tab characters before and after point. M-SPC (`just-one-space`) does likewise but leaves a single space after point, regardless of the number of spaces that existed previously (even zero).

C-x C-o (`delete-blank-lines`) deletes all blank lines after the current line. If the current line is blank, it deletes all blank lines preceding the current line as well (leaving one blank line, the current line).

M-^ (`delete-indentation`) joins the current line and the previous line, by deleting a newline and all surrounding spaces, usually leaving a single space. See section Indentation.

Killing by Lines

C-k

Kill rest of line or one or more lines (`kill-line`).

The simplest kill command is C-k. If given at the beginning of a line, it kills all the text on the line, leaving it blank. When used on a blank line, it kills the whole line including its newline. To kill an entire non-blank line, go to the beginning and type C-k twice.

More generally, C-k kills from point up to the end of the line, unless it is at the end of a line. In that case it kills the newline following point, thus merging the next line into the current one. Spaces and tabs that you can't see at the end of the line are ignored when deciding which case applies, so if point appears to be at the end of the line, you can be sure C-k will kill the newline.

When C-k is given a positive argument, it kills that many lines and the newlines that follow them (however, text on the current line before point is spared). With a negative argument -n, it kills n lines preceding the current line (together with the text on the current line before point). Thus, C-u - 2 C-k at the front of a line kills the two previous lines.

C-k with an argument of zero kills the text before point on the current line.

If the variable `kill-whole-line` is non-`nil`, C-k at the very beginning of a line kills the entire line including the following newline. This variable is normally `nil`.

Other Kill Commands

C-w
: Kill region (from point to the mark) (`kill-region`).

M-d
: Kill word (`kill-word`). See section Words.

M-DEL
: Kill word backwards (`backward-kill-word`).

C-x DEL
: Kill back to beginning of sentence (`backward-kill-sentence`). See section Sentences.

M-k
: Kill to end of sentence (`kill-sentence`).

C-M-k
: Kill sexp (`kill-sexp`). See section Lists and Sexps.

M-z char
: Kill through the next occurrence of char (`zap-to-char`).

A kill command which is very general is C-w (`kill-region`), which kills everything between point and the mark. With this command, you can kill any contiguous sequence of characters, if you first set the region around them.

A convenient way of killing is combined with searching: M-z (`zap-to-char`) reads a character and kills from point up to (and including) the next occurrence of that character in the buffer. A numeric argument acts as a repeat count. A negative argument means to search backward and kill text before point.

Other syntactic units can be killed: words, with M-DEL and M-d (see section Words); sexps, with C-M-k (see section Lists and Sexps); and sentences, with C-x DEL and M-k (see section Sentences).

You can use kill commands in read-only buffers. They don't actually change the buffer, and they beep to warn you of that, but they do copy the text you tried to kill into the kill ring, so you can yank it into other buffers. Most of the kill commands move point across the text they copy in this way, so that successive kill commands build up a single kill ring entry as usual.

Yanking

Yanking means reinserting text previously killed. This is what some systems call "pasting". The usual way to move or copy text is to kill it and then yank it elsewhere one or more times.

C-y
: Yank last killed text (`yank`).

M-y

Replace text just yanked with an earlier batch of killed text (`yank-pop`).

M-w

Save region as last killed text without actually killing it (`kill-ring-save`).

C-M-w

Append next kill to last batch of killed text (`append-next-kill`).

The Kill Ring

All killed text is recorded in the kill ring, a list of blocks of text that have been killed. There is only one kill ring, shared by all buffers, so you can kill text in one buffer and yank it in another buffer. This is the usual way to move text from one file to another. (See section Accumulating Text, for some other ways.)

The command C-y (`yank`) reinserts the text of the most recent kill. It leaves the cursor at the end of the text. It sets the mark at the beginning of the text. See section The Mark and the Region.

C-u C-y leaves the cursor in front of the text, and sets the mark after it. This happens only if the argument is specified with just a C-u, precisely. Any other sort of argument, including C-u and digits, specifies an earlier kill to yank (see section Yanking Earlier Kills).

To copy a block of text, you can use M-w (`kill-ring-save`), which copies the region into the kill ring without removing it from the buffer. This is approximately equivalent to C-w followed by C-x u, except that M-w does not alter the undo history and does not temporarily change the screen.

Appending Kills

Normally, each kill command pushes a new entry onto the kill ring. However, two or more kill commands in a row combine their text into a single entry, so that a single C-y yanks all the text as a unit, just as it was before it was killed.

Thus, if you want to yank text as a unit, you need not kill all of it with one command; you can keep killing line after line, or word after word, until you have killed it all, and you can still get it all back at once.

Commands that kill forward from point add onto the end of the previous killed text. Commands that kill backward from point add text onto the beginning. This way, any sequence of mixed forward and backward kill commands puts all the killed text into one entry without rearrangement. Numeric arguments do not break the sequence of appending kills. For example, suppose the buffer contains this text:

```
This is a line -!-of sample text.
```

with point shown by -!-. If you type M-d M-DEL M-d M-DEL, killing alternately forward and backward, you end up with `a line of sample' as one entry in the kill ring, and `This is text.' in the buffer. (Note the double space, which you can clean up with M-SPC or M-q.)

Another way to kill the same text is to move back two words with M-b M-b, then kill all four words

forward with C-u M-d. This produces exactly the same results in the buffer and in the kill ring. M-f N C-u M-DEL kills the same text, all going backward; once again, the result is the same. The text in th ring entry always has the same order that it had in the buffer before you killed it.

If a kill command is separated from the last kill command by other commands (not just numeric arguments), it starts a new entry on the kill ring. But you can force it to append by first typing the command C-M-w (`append-next-kill`) right before it. The C-M-w tells the following command, if it is a kill command, to append the text it kills to the last killed text, instead of starting a new entry. With C-M-w, you can kill several separated pieces of text and accumulate them to be yanked back in one place.

A kill command following M-w does not append to the text that M-w copied into the kill ring.

Yanking Earlier Kills

To recover killed text that is no longer the most recent kill, use the M-y command (`yank-pop`). It takes the text previously yanked and replaces it with the text from an earlier kill. So, to recover the text of the next-to-the-last kill, first use C-y to yank the last kill, and then use M-y to replace it with the previous kill. M-y is allowed only after a C-y or another M-y.

You can understand M-y in terms of a "last yank" pointer which points at an entry in the kill ring. Each time you kill, the "last yank" pointer moves to the newly made entry at the front of the ring. C-y yanks the entry which the "last yank" pointer points to. M-y moves the "last yank" pointer to a different entry, and the text in the buffer changes to match. Enough M-y commands can move the pointer to any entry in the ring, so you can get any entry into the buffer. Eventually the pointer reaches the end of the ring; the next M-y moves it to the first entry again.

M-y moves the "last yank" pointer around the ring, but it does not change the order of the entries in the ring, which always runs from the most recent kill at the front to the oldest one still remembered.

M-y can take a numeric argument, which tells it how many entries to advance the "last yank" pointer by. A negative argument moves the pointer toward the front of the ring; from the front of the ring, it moves "around" to the last entry and continues forward from there.

Once the text you are looking for is brought into the buffer, you can stop doing M-y commands and it will stay there. It's just a copy of the kill ring entry, so editing it in the buffer does not change what's in the ring. As long as no new killing is done, the "last yank" pointer remains at the same place in the kill ring, so repeating C-y will yank another copy of the same previous kill.

If you know how many M-y commands it would take to find the text you want, you can yank that text in one step using C-y with a numeric argument. C-y with an argument restores the text the specified number of entries back in the kill ring. Thus, C-u 2 C-y gets the next to the last block of killed text. It is equivalent to C-y M-y. C-y with a numeric argument starts counting from the "last yank" pointer, and sets the "last yank" pointer to the entry that it yanks.

The length of the kill ring is controlled by the variable `kill-ring-max`; no more than that many blocks of killed text are saved.

The actual contents of the kill ring are stored in a variable named `kill-ring`; you can view the entire

contents of the kill ring with the command C-h v kill-ring.

Accumulating Text

Usually we copy or move text by killing it and yanking it, but there are other methods convenient for copying one block of text in many places, or for copying many scattered blocks of text into one place. To copy one block to many places, store it in a register (see section Registers). Here we describe the commands to accumulate scattered pieces of text into a buffer or into a file.

M-x append-to-buffer

Append region to contents of specified buffer.

M-x prepend-to-buffer

Prepend region to contents of specified buffer.

M-x copy-to-buffer

Copy region into specified buffer, deleting that buffer's old contents.

M-x insert-buffer

Insert contents of specified buffer into current buffer at point.

M-x append-to-file

Append region to contents of specified file, at the end.

To accumulate text into a buffer, use M-x append-to-buffer. This reads a buffer name, them inserts a copy of the region into the buffer specified. If you specify a nonexistent buffer, `append-to-buffer` creates the buffer. The text is inserted wherever point is in that buffer. If you have been using the buffer for editing, the copied text goes into the middle of the text of the buffer, wherever point happens to be in it.

Point in that buffer is left at the end of the copied text, so successive uses of `append-to-buffer` accumulate the text in the specified buffer in the same order as they were copied. Strictly speaking, `append-to-buffer` does not always append to the text already in the buffer--only if point in that buffer is at the end. However, if `append-to-buffer` is the only command you use to alter a buffer, then point is always at the end.

M-x prepend-to-buffer is just like `append-to-buffer` except that point in the other buffer is left before the copied text, so successive prependings add text in reverse order. M-x copy-to-buffer is similar except that any existing text in the other buffer is deleted, so the buffer is left containing just the text newly copied into it.

To retrieve the accumulated text from another buffer, use M-x insert-buffer; this too takes buffername as an argument. It inserts a copy of the text in buffer buffername into the selected buffer. You can alternatively select the other buffer for editing, then optionally move text from it by killing. See section Using Multiple Buffers, for background information on buffers.

Instead of accumulating text within Emacs, in a buffer, you can append text directly into a file with M-x append-to-file, which takes filename as an argument. It adds the text of the region to the end of the specified file. The file is changed immediately on disk.

You should use `append-to-file` only with files that are *not* being visited in Emacs. Using it on a file that you are editing in Emacs would change the file behind Emacs's back, which can lead to losing some of your editing.

Rectangles

The rectangle commands operate on rectangular areas of the text: all the characters between a certain pair of columns, in a certain range of lines. Commands are provided to kill rectangles, yank killed rectangles, clear them out, fill them with blanks or text, or delete them. Rectangle commands are useful with text in multicolumn formats, and for changing text into or out of such formats.

When you must specify a rectangle for a command to work on, you do it by putting the mark at one corner and point at the opposite corner. The rectangle thus specified is called the region-rectangle because you control it in about the same way the region is controlled. But remember that a given combination of point and mark values can be interpreted either as a region or as a rectangle, depending on the command that uses them.

If point and the mark are in the same column, the rectangle they delimit is empty. If they are in the same line, the rectangle is one line high. This asymmetry between lines and columns comes about because point (and likewise the mark) is between two columns, but within a line.

C-x r k

> Kill the text of the region-rectangle, saving its contents as the "last killed rectangle" (`kill-rectangle`).

C-x r d

> Delete the text of the region-rectangle (`delete-rectangle`).

C-x r y

> Yank the last killed rectangle with its upper left corner at point (`yank-rectangle`).

C-x r o

> Insert blank space to fill the space of the region-rectangle (`open-rectangle`). This pushes the previous contents of the region-rectangle rightward.

M-x clear-rectangle

> Clear the region-rectangle by replacing its contents with spaces.

M-x string-rectangle RET string RET

> Insert string on each line of the region-rectangle.

The rectangle operations fall into two classes: commands deleting and inserting rectangles, and commands for blank rectangles.

There are two ways to get rid of the text in a rectangle: you can discard the text (delete it) or save it as the "last killed" rectangle. The commands for these two ways are C-x r d (`delete-rectangle`) and C-x r k (`kill-rectangle`). In either case, the portion of each line that falls inside the rectangle's boundaries is deleted, causing following text (if any) on the line to move left into the gap.

Note that "killing" a rectangle is not killing in the usual sense; the rectangle is not stored in the kill ri but in a special place that can only record the most recent rectangle killed. This is because yanking a rectangle is so different from yanking linear text that different yank commands have to be used and yank-popping is hard to make sense of.

To yank the last killed rectangle, type C-x r y (`yank-rectangle`). Yanking a rectangle is the opposite of killing one. Point specifies where to put the rectangle's upper left corner. The rectangle's first line is inserted there, the rectangle's second line is inserted at a position one line vertically down, and so on. The number of lines affected is determined by the height of the saved rectangle.

You can convert single-column lists into double-column lists using rectangle killing and yanking; kill the second half of the list as a rectangle and then yank it beside the first line of the list. See section Two-Column Editing, for another way to edit multi-column text.

You can also copy rectangles into and out of registers with C-x r r r and C-x r i r. See section Saving Rectangles in Registers.

There are two commands for making with blank rectangles: M-x clear-rectangle to blank out existing text, and C-x r o (`open-rectangle`) to insert a blank rectangle. Clearing a rectangle is equivalent to deleting it and then inserting a blank rectangle of the same size.

The command M-x string-rectangle is similar to C-x r o, but it inserts a specified string instead of blanks. You specify the string with the minibuffer. Since the length of the string specifies how many columns to insert, the width of the region-rectangle does not matter for this command. What does matter is the position of the left edge (which specifies the column position for the insertion in each line) and the range of lines that the rectangle occupies. The previous contents of the text beyond the insertion column are pushed rightward.

Go to the previous, next section.

Go to the previous, next section.

Registers

Emacs registers are places you can save text or positions for later use. Once you save text or a rectangle in a register, you can copy it into the buffer once or many times; you can move point to a position saved in a register once or many times.

Each register has a name which is a single character. A register can store a piece of text, a rectangle, a position, a window configuration, or a file name, but only one thing at any given time. Whatever you store in a register remains there until you store something else in that register. To see what a register r contains, use M-x view-register.

M-x view-register RET r

Display a description of what register r contains.

Saving Positions in Registers

Saving a position records a place in a buffer so that you can move back there later. Moving to a saved position switches to that buffer and moves point to that place in it.

C-x r SPC r

Save position of point in register r (`point-to-register`).

C-x r j r

Jump to the position saved in register r (`jump-to-register`).

To save the current position of point in a register, choose a name r and type C-x r SPC r. The register r retains the position thus saved until you store something else in that register.

The command C-x r j r moves point to the position recorded in register r. The register is not affected; it continues to record the same position. You can jump to the saved position any number of times.

Saving Text in Registers

When you want to insert a copy of the same piece of text several times, it may be inconvenient to yank it from the kill ring, since each subsequent kill moves that entry further down the ring. An alternative is to store the text in a register and later retrieve it.

C-x r s r

Copy region into register r (`copy-to-register`).

C-x r i r

Insert text from register r (`insert-register`).

C-x r s r stores a copy of the text of the region into the register named r. Given a numeric argument, C-x r

s r deletes the text from the buffer as well.

C-x r i r inserts in the buffer the text from register r. Normally it leaves point before the text and places the mark after, but with a numeric argument (C-u) it puts point after the text and the mark before.

Saving Rectangles in Registers

A register can contain a rectangle instead of linear text. The rectangle is represented as a list of strings. See section Rectangles, for basic information on how to specify a rectangle in the buffer.

C-x r r r

> Copy the region-rectangle into register r (`copy-rectangle-to-register`). With numeric argument, delete it as well.

C-x r i r

> Insert the rectangle stored in register r (if it contains a rectangle) (`insert-register`).

The C-x r i r command inserts a text string if the register contains one, and inserts a rectangle if the register contains one.

See also the command `sort-columns`, which you can think of as sorting a rectangle. See section Sorting Text.

Saving Window Configurations in Registers

You can save the window configuration of the selected frame in a register, or even the configuration of all windows in all frames, and restore the configuration later.

C-x r w r

> Save the state of the selected frame's windows in register r (`window-configuration-to-register`).

C-x r f r

> Save the state of all frames, including all their windows, in register r (`frame-configuration-to-register`).

Use C-x r j r to restore a window or frame configuration. This is the same command used to restore a cursor position. When you restore a frame configuration, any existing frames not included in the configuration become invisible. If you wish to delete these frames instead, use C-u C-x r j r.

Keeping File Names in Registers

If you visit certain file names frequently, you can visit them more conveniently if you put their names in registers. Here's the Lisp code used to put a file name in a register:

```
(set-register ?r '(file . name))
```

For example,

```
(set-register ?z '(file . "/gd/gnu/emacs/19.0/src/ChangeLog"))
```

puts the file name shown in register `z'.

To visit the file whose name is in register r, type C-x r j r. (This is the same command used to jump to a position or restore a frame configuration.)

Bookmarks

Bookmarks are somewhat like registers in that they record positions you can jump to. Unlike registers, they have long names, and they persist automatically from one Emacs session to the next. The prototypical use of bookmarks is to record "where you were reading" in various files.

C-x r m RET
: Set the bookmark for the visited file, at point.

C-x r m bookmark RET
: Set the bookmark named bookmark at point (`bookmark-set`).

C-x r b bookmark RET
: Jump to the bookmark named bookmark (`bookmark-jump`).

C-x r l
: List all bookmarks (`list-bookmarks`).

M-x bookmark-save
: Save all the current bookmark values in the default bookmark file.

The prototypical use for bookmarks is to record one current position in each of several files. So the command C-x r m, which sets a bookmark, uses the visited file name as the default for the bookmark name. If you name each bookmark after the file it points to, then you can conveniently revisit any of those files with C-x r b, and move to the position of the bookmark at the same time.

To display a list of all your bookmarks in a separate buffer, type C-x r l (`list-bookmarks`). If you switch to that buffer, you can use it to edit your bookmark definitions or annotate the bookmarks. Type C-h m in that buffer for more information about its special editing commands.

When you kill Emacs, Emacs offers to save your bookmark values in your default bookmark file, `` `~/.emacs.bmk' ``, if you have changed any bookmark values. You can also save the bookmarks at any time with the M-x bookmark-save command. The bookmark commands load your default bookmark file automatically. This saving and loading is how bookmarks persist from one Emacs session to the next.

If you set the variable `bookmark-save-flag` to 1, then each command that sets a bookmark will also save your bookmarks; this way, you don't lose any bookmark values even if Emacs crashes. (The value, if a number, says how many bookmark modifications should go by between saving.)

Bookmark position values are saved with surrounding context, so that `bookmark-jump` can find the proper position even if the file is modified slightly. The variable `bookmark-search-size` says how

many characters of context to record, on each side of the bookmark's position.

Here are some additional commands for working with bookmarks:

M-x bookmark-load RET filename RET

Load a file named filename that contains a list of bookmark values. You can use this command, as well as `bookmark-write`, to work with other files of bookmark values in addition to your default bookmark file.

M-x bookmark-write RET filename RET

Save all the current bookmark values in the file filename.

M-x bookmark-delete RET bookmark RET

Delete the bookmark named bookmark.

M-x bookmark-insert-location RET bookmark RET

Insert in the buffer the name of the file that bookmark bookmark points to.

M-x bookmark-insert RET bookmark RET

Insert in the buffer the *contents* of the file that bookmark bookmark points to.

Go to the previous, next section.

Go to the previous, next section.

Controlling the Display

Since only part of a large buffer fits in the window, Emacs tries to show the part that is likely to be interesting. The display control commands allow you to specify which part of the text you want to see.

C-l
: Clear screen and redisplay, scrolling the selected window to center point vertically within it (`recenter`).

C-v
: Scroll forward (a windowful or a specified number of lines) (`scroll-up`).

NEXT
: Likewise, scroll forward.

M-v
: Scroll backward (`scroll-down`).

PRIOR
: Likewise, scroll backward.

arg C-l
: Scroll so point is on screen line arg (`recenter`).

C-x <
: Scroll text in current window to the left (`scroll-left`).

C-x >
: Scroll to the right (`scroll-right`).

C-x $
: Make deeply indented lines invisible (`set-selective-display`).

The names of all scroll commands are based on the direction that the text moves in the window. Thus, the command to scrolling forward is called `scroll-up`, since the text moves up.

Scrolling

If a buffer contains text that is too large to fit entirely within a window that is displaying the buffer, Emacs shows a contiguous portion of the text. The portion shown always contains point.

Scrolling means moving text up or down in the window so that different parts of the text are visible. Scrolling forward means that text moves up, and new text appears at the bottom. Scrolling backward moves text down and new text appears at the top.

Scrolling happens automatically if you move point past the bottom or top of the window. You can also explicitly request scrolling with the commands in this section.

The most basic scrolling command is C-l (`recenter`) with no argument. It clears the entire screen and redisplays all windows. In addition, it scrolls the selected window so that point is halfway down from the top of the window.

The scrolling commands C-v and M-v let you move all the text in the window up or down a few lines. C-v (`scroll-up`) with an argument shows you that many more lines at the bottom of the window, moving the text and point up together as C-l might. C-v with a negative argument shows you more lines at the top of the window. M-v (`scroll-down`) is like C-v, but moves in the opposite direction. The function keys NEXT and PRIOR are equivalent to C-v and M-v.

To read the buffer a windowful at a time, use C-v with no argument. It takes the last two lines at the bottom of the window and puts them at the top, followed by nearly a whole windowful of lines not previously visible. If point was in the text scrolled off the top, it moves to the new top of the window. M-v with no argument moves backward with overlap similarly. The number of lines of overlap across a C-v or M-v is controlled by the variable `next-screen-context-lines`; by default, it is two.

Another way to do scrolling is with C-l with a numeric argument. C-l does not clear the screen when given an argument; it only scrolls the selected window. With a positive argument n, it repositions text to put point n lines down from the top. An argument of zero puts point on the very top line. Point does not move with respect to the text; rather, the text and point move rigidly on the screen. C-l with a negative argument puts point that many lines from the bottom of the window. For example, C-u - 1 C-l puts point on the bottom line, and C-u - 5 C-l puts it five lines from the bottom. Just C-u as argument, as in C-u C-l, scrolls point to the center of the screen.

The C-M-l command (`reposition-window`) scrolls the current window heuristically in a way designed to get useful information onto the screen. For example, in a Lisp file, this command tries to get the entire current defun onto the screen if possible.

Scrolling happens automatically if point has moved out of the visible portion of the text when it is time to display. Usually the scrolling is done so as to put point vertically centered within the window. However, if the variable `scroll-step` has a nonzero value, an attempt is made to scroll the buffer by that many lines; if that is enough to bring point back into visibility, that is what is done.

Horizontal Scrolling

The text in a window can also be scrolled horizontally. This means that each line of text is shifted sideways in the window, and one or more characters at the beginning of each line are not displayed at all. When a window has been scrolled horizontally in this way, text lines are truncated rather than continued (see section Continuation Lines), with a `$' appearing in the first column when there is text truncated to the left, and in the last column when there is text truncated to the right.

The command C-x < (`scroll-left`) scrolls the selected window to the left by n columns with argument n. This moves part of the beginning of each line off the left edge of the window. With no argument, it scrolls by almost the full width of the window (two columns less, to be precise).

C-x > (`scroll-right`) scrolls similarly to the right. The window cannot be scrolled any farther to the right once it is displayed normally (with each line starting at the window's left margin); attempting to do

so has no effect. This means that you don't have to calculate the argument precisely for C-x >; any sufficiently large argument will restore normally display.

Selective Display

Emacs has the ability to hide lines indented more than a certain number of columns (you specify how many columns). You can use this to get an overview of a part of a program.

To hide lines, type C-x $ (`set-selective-display`) with a numeric argument n. Then lines with at least n columns of indentation disappear from the screen. The only indication of their presence is that three dots (`...') appear at the end of each visible line that is followed by one or more hidden ones.

The commands C-n and C-p move across the hidden lines as if they were not there.

The hidden lines are still present in the buffer, and most editing commands see them as usual, so you may find point in the middle of the hidden text. When this happens, the cursor appears at the end of the previous line, after the three dots. If point is at the end of the visible line, before the newline that ends it, the cursor appears before the three dots.

To make all lines visible again, type C-x $ with no argument.

If you set the variable `selective-display-ellipses` to `nil`, the three dots do not appear at the end of a line that precedes hidden lines. Then there is no visible indication of the hidden lines. This variable becomes local automatically when set.

European Character Set Display

Some European languages use accented letters and other special symbols. The ISO 8859 Latin-1 character set defines character codes for many European languages in the range 160 to 255.

Emacs can display those characters according to Latin-1, provided the terminal or font in use supports them. The M-x standard-display-european command toggles European character display mode. With a numeric argument, M-x standard-display-european enables European character display if and only if the argument is positive. Load the library `iso-syntax` to specify the correct syntactic properties and case conversion table for the Latin-1 character set.

If your terminal does not support display of the Latin-1 character set, Emacs can display these characters as ASCII sequences which at least give you a clear idea of what the characters are. To do this, load the library `iso-ascii`.

Some operating systems let you specify the language you are using by setting a locale. Emacs handles one common special case of this: if your locale name for character types contains the string `8859-1' or `88591', Emacs automatically enables European character display mode and its syntax.

There are three different ways you can enter Latin-1 characters:

- If your keyboard can generate character codes 128 and up, representing ISO Latin-1 characters, execute the following expression to enable Emacs to understand them:


```
(set-input-mode (car (current-input-mode))
                (nth 1 (current-input-mode))
                0)
```

- You can load the library `iso-transl` to turn the key C-x 8 into a "compose character" prefix for entry of the extra ISO Latin-1 printing characters. C-x 8 is good for insertion (in the minibuffer as well as other buffers), for searching, and in any other context where a key sequence is allowed. The ALT modifier key, if you have one, serves the same purpose as C-x 8; use ALT together with an accent character to modify the following letter.
- You can use ISO Accents mode. This minor mode is convenient if you enter non-ASCII ISO Latin-1 characters often. When this minor mode is enabled, the characters ``', `'', `'', `^', `/' and `~' modify the following letter by adding the corresponding diacritical mark to it, if possible. To enable or disable ISO Accents mode, use the command M-x iso-accents-mode. This command affects only the current buffer.

  To enter one of those six special characters while in ISO Accents mode, type the character, followed by a space. Some of those characters have a corresponding "dead key" accent character in the ISO Latin-1 character set; to enter that character, type the corresponding ASCII character twice. For example, " enters the Latin-1 character acute-accent (character code 0264).

  ISO Accents mode input is available whenever a key sequence is expected: for ordinary insertion, for searching, for the minibuffer, and for certain command arguments.

  In addition to the accented letters, you can use these special sequences in ISO Accents mode to enter certain other ISO Latin-1 characters:

  /A
  : `A' with ring.

  ~C
  : `C' with cedilla.

  ~D
  : `D' with stroke.

  /E
  : `AE' ligature.

  /a
  : `a' with ring.

  ~c
  : `c' with cedilla.

  ~d
  : `d' with stroke.

  /e
  : `ae' ligature.

  "s

German sharp `s'.

~<

Left guillemot.

~>

Right guillemot.

~!

Inverted exclamation mark.

~?

Inverted question mark.

Follow Mode

Follow mode is a minor mode which makes two windows showing the same buffer scroll as one tall "virtual window." To use Follow mode, go to a frame with just one window, split it into two side-by-side windows using C-x 3, and then type M-x follow-mode. From then on, you can edit the buffer in either of the two windows, or scroll either one; the other window follows it.

To turn off Follow mode, type M-x follow-mode a second time.

Optional Mode Line Features

The current line number of point appears in the mode line when Line Number mode is enabled. Use the command M-x line-number-mode to turn this mode on and off; normally it is on. The line number appears before the buffer percentage pos, with the letter `L' to indicate what it is. See section Minor Modes, for more information about minor modes and about how to use this command.

If the buffer is very large (larger than the value of `line-number-display-limit`), then the line number doesn't appear. Emacs doesn't compute the line number when the buffer is large, because that would be too slow. If you have narrowed the buffer (see section Narrowing), the displayed line number is relative to the accessible portion of the buffer.

You can also display the current column number by turning on Column Number mode. It displays the current column number preceded by the letter `C'. Type M-x column-number-mode to toggle this mode.

Emacs can optionally display the time and system load in all mode lines. To enable this feature, type M-x display-time. The information added to the mode line usually appears after the buffer name, before the mode names and their parentheses. It looks like this:

```
hh:mmpm l.ll
```

Here hh and mm are the hour and minute, followed always by `am' or `pm'. l.ll is the average number of running processes in the whole system recently. (Some fields may be missing if your operating system cannot support them.)

The word `Mail' appears after the load level if there is mail for you that you have not read yet.

Variables Controlling Display

This section contains information for customization only. Beginning users should skip it.

The variable `mode-line-inverse-video` controls whether the mode line is displayed in inverse video (assuming the terminal supports it); `nil` means don't do so. See section The Mode Line. If you specify the foreground color for the `modeline` face, and `mode-line-inverse-video` is non-`nil`, then the default background color for that face is the usual foreground color. See section Using Multiple Typefaces.

If the variable `inverse-video` is non-`nil`, Emacs attempts to invert all the lines of the display from what they normally are.

If the variable `visible-bell` is non-`nil`, Emacs attempts to make the whole screen blink when it would normally make an audible bell sound. This variable has no effect if your terminal does not have a way to make the screen blink.

When you reenter Emacs after suspending, Emacs normally clears the screen and redraws the entire display. On some terminals with more than one page of memory, it is possible to arrange the termcap entry so that the `ti' and `te' strings (output to the terminal when Emacs is entered and exited, respectively) switch between pages of memory so as to use one page for Emacs and another page for other output. Then you might want to set the variable `no-redraw-on-reenter` non-`nil`; this tells Emacs to assume, when resumed, that the screen page it is using still contains what Emacs last wrote there.

The variable `echo-keystrokes` controls the echoing of multi-character keys; its value is the number of seconds of pause required to cause echoing to start, or zero meaning don't echo at all. See section The Echo Area.

If the variable `ctl-arrow` is `nil`, control characters in the buffer are displayed with octal escape sequences, all except newline and tab. Altering the value of `ctl-arrow` makes it local to the current buffer; until that time, the default value is in effect. The default is initially `t`. See section 'Display Tables' in The Emacs Lisp Reference Manual.

Normally, a tab character in the buffer is displayed as whitespace which extends to the next display tab stop position, and display tab stops come at intervals equal to eight spaces. The number of spaces per tab is controlled by the variable `tab-width`, which is made local by changing it, just like `ctl-arrow`. Note that how the tab character in the buffer is displayed has nothing to do with the definition of TAB as a command. The variable `tab-width` must have an integer value between 1 and 1000, inclusive.

If the variable `truncate-lines` is non-`nil`, then each line of text gets just one screen line for display; if the text line is too long, display shows only the part that fits. If `truncate-lines` is `nil`, then long text lines display as more than one screen line, enough to show the whole text of the line. See section Continuation Lines. Altering the value of `truncate-lines` makes it local to the current buffer; until that time, the default value is in effect. The default is initially `nil`.

If the variable `truncate-partial-width-windows` is non-`nil`, it forces truncation rather than continuation in any window less than the full width of the screen or frame, regardless of the value of `truncate-lines`. For information about side-by-side windows, see section Splitting Windows. See also section 'Display' in The Emacs Lisp Reference Manual.

The variable `baud-rate` holds the the output speed of the terminal, as far as Emacs knows. Setting this variable does not change the speed of actual data transmission, but the value is used for calculations such as padding. It also affects decisions about whether to scroll part of the screen or redraw it instead--even when using a window system. (We designed it this way, despite the fact that a window system has no true "output speed", to give you a way to tune these decisions.)

Go to the previous, next section.

Go to the previous, next section.

Searching and Replacement

Like other editors, Emacs has commands for searching for occurrences of a string. The principal search command is unusual in that it is incremental; it begins to search before you have finished typing the search string. There are also nonincremental search commands more like those of other editors.

Besides the usual `replace-string` command that finds all occurrences of one string and replaces them with another, Emacs has a fancy replacement command called `query-replace` which asks interactively which occurrences to replace.

Incremental Search

An incremental search begins searching as soon as you type the first character of the search string. As you type in the search string, Emacs shows you where the string (as you have typed it so far) would be found. When you have typed enough characters to identify the place you want, you can stop. Depending on what you plan to do next, you may or may not need to terminate the search explicitly with RET.

C-s

Incremental search forward (`isearch-forward`).

C-r

Incremental search backward (`isearch-backward`).

C-s starts an incremental search. C-s reads characters from the keyboard and positions the cursor at the first occurrence of the characters that you have typed. If you type C-s and then F, the cursor moves right after the first `F'. Type an O, and see the cursor move to after the first `FO'. After another O, the cursor is after the first `FOO' after the place where you started the search. Meanwhile, the search string `FOO' has been echoed in the echo area.

If you make a mistake in typing the search string, you can cancel characters with DEL. Each DEL cancels the last character of search string. This does not happen until Emacs is ready to read another input character; first it must either find, or fail to find, the character you want to erase. If you do not want to wait for this to happen, use C-g as described below.

When you are satisfied with the place you have reached, you can type RET, which stops searching, leaving the cursor where the search brought it. Also, any command not specially meaningful in searches stops the searching and is then executed. Thus, typing C-a would exit the search and then move to the beginning of the line. RET is necessary only if the next command you want to type is a printing character, DEL, RET, or another control character that is special within searches (C-q, C-w, C-r, C-s, C-y, M-y, M-r, or M-s).

Sometimes you search for `FOO' and find it, but not the one you expected to find. There was a second `FOO' that you forgot about, before the one you were looking for. In this event, type another C-s to move to the next occurrence of the search string. This can be done any number of times. If you overshoot, you

can cancel some C-s characters with DEL.

After you exit a search, you can search for the same string again by typing just C-s C-s: the first C-s is the key that invokes incremental search, and the second C-s means "search again".

To reuse earlier search strings, use the search ring. The commands M-p and M-n move through the ring to pick a search string to reuse. These commands leave the selected search ring element in the minibuffer, where you can edit it. Type C-s or C-r to terminate editing the string and search for it.

If your string is not found at all, the echo area says `Failing I-Search'. The cursor is after the place where Emacs found as much of your string as it could. Thus, if you search for `FOOT', and there is no `FOOT', you might see the cursor after the `FOO' in `FOOL'. At this point there are several things you can do. If your string was mistyped, you can rub some of it out and correct it. If you like the place you have found, you can type RET or some other Emacs command to "accept what the search offered". Or you can type C-g, which removes from the search string the characters that could not be found (the `T' in `FOOT'), leaving those that were found (the `FOO' in `FOOT'). A second C-g at that point cancels the search entirely, returning point to where it was when the search started.

An upper-case letter in the search string makes the search case-sensitive. If you delete the upper-case character from the search string, it ceases to have this effect. See section Searching and Case.

If a search is failing and you ask to repeat it by typing another C-s, it starts again from the beginning of the buffer. Repeating a failing reverse search with C-r starts again from the end. This is called wrapping around. `Wrapped' appears in the search prompt once this has happened. If you keep on going past the original starting point of the search, it changes to `Overwrapped', which means that you are revisiting matches that you have already seen.

The C-g "quit" character does special things during searches; just what it does depends on the status of the search. If the search has found what you specified and is waiting for input, C-g cancels the entire search. The cursor moves back to where you started the search. If C-g is typed when there are characters in the search string that have not been found--because Emacs is still searching for them, or because it has failed to find them--then the search string characters which have not been found are discarded from the search string. With them gone, the search is now successful and waiting for more input, so a second C-g will cancel the entire search.

To search for a newline, type LFD (also known as C-j). To search for another control character such as control-S or carriage return, you must quote it by typing C-q first. This function of C-q is analogous to its meaning as an Emacs command: it causes the following character to be treated the way a graphic character would normally be treated in the same context. You can also specify a character by its octal code: enter C-q followed by three octal digits.

You can change to searching backwards with C-r. If a search fails because the place you started was too late in the file, you should do this. Repeated C-r keeps looking for more occurrences backwards. A C-s starts going forwards again. C-r in a search can be canceled with DEL.

If you know initially that you want to search backwards, you can use C-r instead of C-s to start the search, because C-r as a key runs a command (`isearch-backward`) to search backward.

The characters C-y and C-w can be used in incremental search to grab text from the buffer into the search

string. This makes it convenient to search for another occurrence of text at point. C-w copies the wor after point as part of the search string, advancing point over that word. Another C-s to repeat the sear will then search for a string including that word. C-y is similar to C-w but copies all the rest of the current line into the search string. Both C-y and C-w convert the text they copy to lower case if the search is current not case-sensitive; this is so the search remains case-insensitive.

The character M-y copies text from the kill ring into the search string. It uses the same text that C-y as a command would yank. See section Yanking.

When you exit the incremental search, it sets the mark to where point *was*, before the search. That is convenient for moving back there. In Transient Mark mode, incremental search sets the mark without activating it, and does so only if the mark is not already active.

To customize the special characters that incremental search understands, alter their bindings in the keymap `isearch-mode-map`. For a list of bindings, look at the documentation of `isearch-mode` with C-h f isearch-mode RET.

Slow Terminal Incremental Search

Incremental search on a slow terminal uses a modified style of display that is designed to take less time. Instead of redisplaying the buffer at each place the search gets to, it creates a new single-line window and uses that to display the line that the search has found. The single-line window comes into play as soon as point gets outside of the text that is already on the screen.

When you terminate the search, the single-line window is removed. Then Emacs redisplays the window in which the search was done, to show its new position of point.

The slow terminal style of display is used when the terminal baud rate is less than or equal to the value of the variable `search-slow-speed`, initially 1200.

The number of lines to use in slow terminal search display is controlled by the variable `search-slow-window-lines`. 1 is its normal value.

Nonincremental Search

Emacs also has conventional nonincremental search commands, which require you to type the entire search string before searching begins.

C-s RET string RET

Search for string.

C-r RET string RET

Search backward for string.

To do a nonincremental search, first type C-s RET. This enters the minibuffer to read the search string; terminate the string with RET, and then the search takes place. If the string is not found, the search command gets an error.

The way C-s RET works is that the C-s invokes incremental search, which is specially programmed t invoke nonincremental search if the argument you give it is empty. (Such an empty argument would otherwise be useless.) C-r RET also works this way.

However, nonincremental searches performed using C-s RET do not call `search-forward` right away. The first thing done is to see if the next character is C-w, which requests a word search.

Forward and backward nonincremental searches are implemented by the commands `search-forward` and `search-backward`. These commands may be bound to keys in the usual manner. The feature that you can get to them via the incremental search commands exists for historical reasons, and to avoid the need to find suitable key sequences for them.

Word Search

Word search searches for a sequence of words without regard to how the words are separated. More precisely, you type a string of many words, using single spaces to separate them, and the string can be found even if there are multiple spaces, newlines or other punctuation between the words.

Word search is useful for editing a printed document made with a text formatter. If you edit while looking at the printed, formatted version, you can't tell where the line breaks are in the source file. With word search, you can search without having to know them.

C-s RET C-w words RET

Search for words, ignoring details of punctuation.

C-r RET C-w words RET

Search backward for words, ignoring details of punctuation.

Word search is a special case of nonincremental search and is invoked with C-s RET C-w. This is followed by the search string, which must always be terminated with RET. Being nonincremental, this search does not start until the argument is terminated. It works by constructing a regular expression and searching for that; see section Regular Expression Search.

Use C-r RET C-w to do backward word search.

Forward and backward word searches are implemented by the commands `word-search-forward` and `word-search-backward`. These commands may be bound to keys in the usual manner. The feature that you can get to them via the incremental search commands exists for historical reasons, and to avoid the need to find suitable key sequences for them.

Regular Expression Search

A regular expression (regexp, for short) is a pattern that denotes a class of alternative strings to match, possibly infinitely many. In GNU Emacs, you can search for the next match for a regexp either incrementally or not.

Incremental search for a regexp is done by typing C-M-s (`isearch-forward-regexp`). This

command reads a search string incrementally just like C-s, but it treats the search string as a regexp rather than looking for an exact match against the text in the buffer. Each time you add text to the search string, you make the regexp longer, and the new regexp is searched for. To search backward in the buffer, use C-M-r (`isearch-backward-regexp`).

All of the control characters that do special things within an ordinary incremental search have the same function in incremental regexp search. Typing C-s or C-r immediately after starting the search retrieves the last incremental search regexp used; that is to say, incremental regexp and non-regexp searches have independent defaults. They also have separate search rings that you can access with M-p and M-n.

If you type SPC in incremental regexp search, it matches any sequence of whitespace characters, including newlines. If you want to match just a space, type C-q SPC.

Note that adding characters to the regexp in an incremental regexp search can make the cursor move back and start again. For example, if you have searched for `foo' and you add `\|bar', the cursor backs up in case the first `bar' precedes the first `foo'.

Nonincremental search for a regexp is done by the functions `re-search-forward` and `re-search-backward`. You can invoke these with M-x, or bind them to keys, or invoke them by way of incremental regexp search with C-M-s RET and C-M-r RET.

Syntax of Regular Expressions

Regular expressions have a syntax in which a few characters are special constructs and the rest are ordinary. An ordinary character is a simple regular expression which matches that same character and nothing else. The special characters are `$', `^', `.', `*', `+', `?', `[', `]' and `\'. Any other character appearing in a regular expression is ordinary, unless a `\' precedes it.

For example, `f' is not a special character, so it is ordinary, and therefore `f' is a regular expression that matches the string `f' and no other string. (It does *not* match the string `ff'.) Likewise, `o' is a regular expression that matches only `o'. (When case distinctions are being ignored, these regexps also match `F' and `O', but we consider this a generalization of "the same string", rather than an exception.)

Any two regular expressions a and b can be concatenated. The result is a regular expression which matches a string if a matches some amount of the beginning of that string and b matches the rest of the string.

As a simple example, we can concatenate the regular expressions `f' and `o' to get the regular expression `fo', which matches only the string `fo'. Still trivial. To do something nontrivial, you need to use one of the special characters. Here is a list of them.

. (Period)

> is a special character that matches any single character except a newline. Using concatenation, we can make regular expressions like `a.b' which matches any three-character string which begins with `a' and ends with `b'.

*

> is not a construct by itself; it is a postfix operator, which means to match the preceding regular

expression repetitively as many times as possible. Thus, `o*' matches any number of `o's (including no `o's).

`*' always applies to the *smallest* possible preceding expression. Thus, `fo*' has a repeating `o', not a repeating `fo'. It matches `f', `fo', `foo', and so on.

The matcher processes a `*' construct by matching, immediately, as many repetitions as can be found. Then it continues with the rest of the pattern. If that fails, backtracking occurs, discarding some of the matches of the `*'-modified construct in case that makes it possible to match the rest of the pattern. For example, matching `ca*ar' against the string `caaar', the `a*' first tries to match all three `a's; but the rest of the pattern is `ar' and there is only `r' left to match, so this try fails. The next alternative is for `a*' to match only two `a's. With this choice, the rest of the regexp matches successfully.

+

is a postfix character, similar to `*' except that it must match the preceding expression at least once. So, for example, `ca+r' matches the strings `car' and `caaaar' but not the string `cr', whereas `ca*r' matches all three strings.

?

is a postfix character, similar to `*' except that it can match the preceding expression either once or not at all. For example, `ca?r' matches `car' or `cr'; nothing else.

[...]

is a character set, which begins with `[' and is terminated by `]'. In the simplest case, the characters between the two brackets are what this set can match.

Thus, `[ad]' matches either one `a' or one `d', and `[ad]*' matches any string composed of just `a's and `d's (including the empty string), from which it follows that `c[ad]*r' matches `cr', `car', `cdr', `caddaar', etc.

You can also include character ranges a character set, by writing two characters with a `-' between them. Thus, `[a-z]' matches any lower-case letter. Ranges may be intermixed freely with individual characters, as in `[a-z$%.]', which matches any lower case letter or `$', `%' or period.

Note that the usual regexp special characters are not special inside a character set. A completely different set of special characters exists inside character sets: `]', `-' and `^'.

To include a `]' in a character set, you must make it the first character. For example, `[]a]' matches `]' or `a'. To include a `-', write `-' as the first or last character of the set, or put it after a range. Thus, `[]-]' matches both `]' and `-'.

To include `^', make it other than the first character in the set.

[^ ...]

`[^' begins a complemented character set, which matches any character except the ones specified. Thus, `[^a-z0-9A-Z]' matches all characters *except* letters and digits.

`^' is not special in a character set unless it is the first character. The character following the `^' is treated as if it were first (`-' and `]' are not special there).

A complemented character set can match a newline, unless newline is mentioned as one of the characters not to match. This is in contrast to the handling of regexps in programs such as `gre`.

^

is a special character that matches the empty string, but only at the beginning of a line in the text being matched. Otherwise it fails to match anything. Thus, `^foo' matches a `foo' which occurs at the beginning of a line.

$

is similar to `^' but matches only at the end of a line. Thus, `xx*$' matches a string of one `x' or more at the end of a line.

\

has two functions: it quotes the special characters (including `\'), and it introduces additional special constructs.

Because `\' quotes special characters, `\$' is a regular expression which matches only `$', and `\[' is a regular expression which matches only `[', etc.

Note: for historical compatibility, special characters are treated as ordinary ones if they are in contexts where their special meanings make no sense. For example, `*foo' treats `*' as ordinary since there is no preceding expression on which the `*' can act. It is poor practice to depend on this behavior; better to quote the special character anyway, regardless of where it appears.

For the most part, `\' followed by any character matches only that character. However, there are several exceptions: two-character sequences starting with `\' which have special meanings. The second character in the sequence is always an ordinary character on their own. Here is a table of `\' constructs.

\|

specifies an alternative. Two regular expressions a and b with `\|' in between form an expression that matches anything that either a or b matches.

Thus, `foo\|bar' matches either `foo' or `bar' but no other string.

`\|' applies to the largest possible surrounding expressions. Only a surrounding `\(... \)' grouping can limit the scope of `\|'.

Full backtracking capability exists to handle multiple uses of `\|'.

\(... \)

is a grouping construct that serves three purposes:

To enclose a set of `\|' alternatives for other operations. Thus, `\(foo\|bar\)x' matches either `foox' or `barx'.

To enclose a complicated expression for the postfix operators `*', `+' and `?' to operate on. Thus, `ba\(na\)*' matches `bananana', etc., with any (zero or more) number of `na' strings.

To mark a matched substring for future reference.

This last application is not a consequence of the idea of a parenthetical grouping; it is a separate feature which is assigned as a second meaning to the same `\(... \)' construct. In practice there is no

conflict between the two meanings. Here is an explanation of this feature:

\d

after the end of a `\( ... \)' construct, the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use `\' followed by the digit d to mean "match the same text matched the dth time by the `\(... \)' construct."

The strings matching the first nine `\( ... \)' constructs appearing in a regular expression are assigned numbers 1 through 9 in order that the open-parentheses appear in the regular expression. `\1' through `\9' refer to the text previously matched by the corresponding `\(... \)' construct.

For example, `\(.*\)\1' matches any newline-free string that is composed of two identical halves. The `\(.*\)' matches the first half, which may be anything, but the `\1' that follows must match the same exact text.

If a particular `\( ... \)' construct matches more than once (which can easily happen if it is followed by `*'), only the last match is recorded.

\`

matches the empty string, provided it is at the beginning of the buffer or string being matched against.

\'

matches the empty string, provided it is at the end of the buffer or string being matched against.

\=

matches the empty string, provided it is at point.

\b

matches the empty string, provided it is at the beginning or end of a word. Thus, `\bfoo\b' matches any occurrence of `foo' as a separate word. `\bballs?\b' matches `ball' or `balls' as a separate word.

`\b' matches at the beginning or end of the buffer regardless of what text appears next to it.

\B

matches the empty string, provided it is *not* at the beginning or end of a word.

\<

matches the empty string, provided it is at the beginning of a word. `\<' matches at the beginning of the buffer only if a word-constituent character follows.

\>

matches the empty string, provided it is at the end of a word. `\>' matches at the end of the buffer only if the contents end with a word-constituent character.

\w

matches any word-constituent character. The syntax table determines which characters these are. See section The Syntax Table.

\W

matches any character that is not a word-constituent.

\sc

matches any character whose syntax is c. Here c is a character which represents a syntax code: thus, `w' for word constituent, `(' for open-parenthesis, etc. Represent a character of whitespace (which can be a newline) by either `-' or a space character.

\Sc

matches any character whose syntax is not c.

The constructs that pertain to words and syntax are controlled by the setting of the syntax table (see section The Syntax Table).

Here is a complicated regexp, used by Emacs to recognize the end of a sentence together with any whitespace that follows. It is given in Lisp syntax to enable you to distinguish the spaces from the tab characters. In Lisp syntax, the string constant begins and ends with a double-quote. `\"' stands for a double-quote as part of the regexp, `\\' for a backslash as part of the regexp, `\t' for a tab and `\n' for a newline.

```
"[.?!][]\"')]*\\($\\|\t\\|  \\)[ \t\n]*"
```

This contains four parts in succession: a character set matching period, `?', or `!'; a character set matching close-brackets, quotes, or parentheses, repeated any number of times; an alternative in backslash-parentheses that matches end-of-line, a tab, or two spaces; and a character set matching whitespace characters, repeated any number of times.

To enter the same regexp interactively, you would type TAB to enter a tab, and C-q C-j to enter a newline. You would also type single backslashes as themselves, instead of doubling them for Lisp syntax.

Searching and Case

Incremental searches in Emacs normally ignore the case of the text they are searching through, if you specify the text in lower case. Thus, if you specify searching for `foo', then `Foo' and `foo' are also considered a match. Regexps, and in particular character sets, are included: `[ab]' would match `a' or `A' or `b' or `B'.

An upper-case letter in the incremental search string makes the search case-sensitive. Thus, searching for `Foo' does not find `foo' or `FOO'. This applies to regular expression search as well as to string search. The effect ceases if you delete the upper-case letter from the search string.

If you set the variable `case-fold-search` to `nil`, then all letters must match exactly, including case. This is a per-buffer variable; altering the variable affects only the current buffer, but there is a default value which you can change as well. See section Local Variables. This variable applies to nonincremental searches also, including those performed by the replace commands (see section Replacement Commands).

Replacement Commands

Global search-and-replace operations are not needed as often in Emacs as they are in other editors(1), but they are available. In addition to the simple M-x replace-string command which is like that found in most editors, there is a M-x query-replace command which asks you, for each occurrence of the pattern, whether to replace it.

The replace commands all replace one string (or regexp) with one replacement string. It is possible to perform several replacements in parallel using the command `expand-region-abbrevs`. See section Controlling Abbrev Expansion.

Unconditional Replacement

M-x replace-string RET string RET newstring RET

Replace every occurrence of string with newstring.

M-x replace-regexp RET regexp RET newstring RET

Replace every match for regexp with newstring.

To replace every instance of `foo' after point with `bar', use the command M-x replace-string with the two arguments `foo' and `bar'. Replacement happens only in the text after point, so if you want to cover the whole buffer you must go to the beginning first. All occurrences up to the end of the buffer are replaced; to limit replacement to part of the buffer, narrow to that part of the buffer before doing the replacement (see section Narrowing).

When `replace-string` exits, it leaves point at the last occurrence replaced. It sets the mark to the prior position of point (where the `replace-string` command was issued); use C-u C-SPC to move back there.

A numeric argument restricts replacement to matches that are surrounded by word boundaries. The argument's value doesn't matter.

Regexp Replacement

The M-x replace-string command replaces exact matches for a single string. The similar command M-x replace-regexp replaces any match for a specified pattern.

In `replace-regexp`, the newstring need not be constant: it can refer to all or part of what is matched by the regexp. `\&' in newstring stands for the entire match being replaced. `\d' in newstring, where d is a digit, stands for whatever matched the dth parenthesized grouping in regexp. To include a `\' in the text to replace with, you must enter `\\'. For example,

```
M-x replace-regexp RET c[ad]+r RET \&-safe RET
```

replaces (for example) `cadr' with `cadr-safe' and `cddr' with `cddr-safe'.


```
M-x replace-regexp RET \(c[ad]+r\)-safe RET \1 RET
```

performs the inverse transformation.

Replace Commands and Case

If the arguments to a replace command are in lower case, it preserves case when it makes a replacement. Thus, the command

```
M-x replace-string RET foo RET bar RET
```

replaces a lower case `foo' with a lower case `bar', an all-caps `FOO' with `BAR', and a capitalized `Foo' with `Bar'. (These three alternatives--lower case, all caps, and capitalized, are the only ones that `replace-string` can distinguish.)

If upper case letters are used in the second argument, they remain upper case every time that argument is inserted. If upper case letters are used in the first argument, the second argument is always substituted exactly as given, with no case conversion. Likewise, if the variable `case-replace` is set to `nil`, replacement is done without case conversion. If `case-fold-search` is set to `nil`, case is significant in matching occurrences of `foo' to replace; this also inhibits case conversion of the replacement string.

Query Replace

M-% string RET newstring RET

M-x query-replace RET string RET newstring RET

Replace some occurrences of string with newstring.

M-x query-replace-regexp RET regexp RET newstring RET

Replace some matches for regexp with newstring.

If you want to change only some of the occurrences of `foo' to `bar', not all of them, then you cannot use an ordinary `replace-string`. Instead, use M-% (`query-replace`). This command finds occurrences of `foo' one by one, displays each occurrence and asks you whether to replace it. A numeric argument to `query-replace` tells it to consider only occurrences that are bounded by word-delimiter characters. This preserves case, just like `replace-string`, provided `case-replace` is non-`nil`, as it normally is.

Aside from querying, `query-replace` works just like `replace-string`, and `query-replace-regexp` works just like `replace-regexp`. The shortest way to type this command name is M-x que SPC SPC SPC RET.

The things you can type when you are shown an occurrence of string or a match for regexp are:

SPC

to replace the occurrence with newstring.

DEL

to skip to the next occurrence without replacing this one.

, (Comma)

to replace this occurrence and display the result. You are then asked for another input character to say what to do next. Since the replacement has already been made, DEL and SPC are equivalent in this situation; both move to the next occurrence.

You could type C-r at this point (see below) to alter the replaced text. You could also type C-x u to undo the replacement; this exits the `query-replace`, so if you want to do further replacement you must use C-x ESC ESC RET to restart (see section Repeating Minibuffer Commands).

RET

to exit without doing any more replacements.

. (Period)

to replace this occurrence and then exit without searching for more occurrences.

!

to replace all remaining occurrences without asking again.

^

to go back to the position of the previous occurrence (or what used to be an occurrence), in case you changed it by mistake. This works by popping the mark ring. Only one ^ in a row is meaningful, because only one previous replacement position is kept during `query-replace`.

C-r

to enter a recursive editing level, in case the occurrence needs to be edited rather than just replaced with newstring. When you are done, exit the recursive editing level with C-M-c to proceed to the next occurrence. See section Recursive Editing Levels.

C-w

to delete the occurrence, and then enter a recursive editing level as in C-r. Use the recursive edit to insert text to replace the deleted occurrence of string. When done, exit the recursive editing level with C-M-c to proceed to the next occurrence.

C-l

to redisplay the screen. Then you must type another character to specify what to do with this occurrence.

C-h

to display a message summarizing these options. Then you must type another character to specify what to do with this occurrence.

Some other characters are aliases for the ones listed above: y, n and q are equivalent to SPC, DEL and RET.

Aside from this, any other character exits the `query-replace`, and is then reread as part of a key sequence. Thus, if you type C-k, it exits the `query-replace` and then kills to end of line.

To restart a `query-replace` once it is exited, use C-x ESC ESC, which repeats the `query-replace` because it used the minibuffer to read its arguments. See section Repeating Minibuffer Commands.

See also section Transforming File Names in Dired, for Dired commands to rename, copy, or link fil by replacing regexp matches in file names.

Other Search-and-Loop Commands

Here are some other commands that find matches for a regular expression. They all operate from point to the end of the buffer.

M-x occur RET regexp RET

> Display a list showing each line in the buffer that contains a match for regexp. A numeric argument specifies the number of context lines to print before and after each matching line; the default is none. To limit the search to part of the buffer, narrow to that part (see section Narrowing).
>
> The buffer `*Occur*' containing the output serves as a menu for finding the occurrences in their original context. Click Mouse-2 on an occurrence listed in `*Occur*', or position point there and type RET; this switches to the buffer that was searched and moves point to the original of the chosen occurrence.

M-x list-matching-lines

> Synonym for M-x occur.

M-x count-matches RET regexp RET

> Print the number of matches for regexp after point.

M-x flush-lines RET regexp RET

> Delete each line that follows point and contains a match for regexp.

M-x keep-lines RET regexp RET

> Delete each line that follows point and *does not* contain a match for regexp.

Go to the previous, next section.

Go to the previous, next section.

Commands for Fixing Typos

In this chapter we describe the commands that are especially useful for the times when you catch a mistake in your text just after you have made it, or change your mind while composing text on the fly.

The most fundamental command for correcting erroneous editing is the undo command, C-x u or C-_. This command undoes a single command (usually), a part of a command (in the case of `query-replace`), or several consecutive self-inserting characters. Consecutive repetitions of C-_ or C-x u undo earlier and earlier changes, back to the limit of the undo information available. See section Undoing Changes, for for more information.

Killing Your Mistakes

DEL
: Delete last character (`delete-backward-char`).

M-DEL
: Kill last word (`backward-kill-word`).

C-x DEL
: Kill to beginning of sentence (`backward-kill-sentence`).

The DEL character (`delete-backward-char`) is the most important correction command. It deletes the character before point. When DEL follows a self-inserting character command, you can think of it as canceling that command. However, avoid the mistake of thinking of DEL as a general way to cancel a command!

When your mistake is longer than a couple of characters, it might be more convenient to use M-DEL or C-x DEL. M-DEL kills back to the start of the last word, and C-x DEL kills back to the start of the last sentence. C-x DEL is particularly useful when you change your mind about the phrasing of the text you are writing. M-DEL and C-x DEL save the killed text for C-y and M-y to retrieve. See section Yanking.

M-DEL is often useful even when you have typed only a few characters wrong, if you know you are confused in your typing and aren't sure exactly what you typed. At such a time, you cannot correct with DEL except by looking at the screen to see what you did. Often it requires less thought to kill the whole word and start again.

Transposing Text

C-t
: Transpose two characters (`transpose-chars`).

M-t

Transpose two words (`transpose-words`).

C-M-t

Transpose two balanced expressions (`transpose-sexps`).

C-x C-t

Transpose two lines (`transpose-lines`).

The common error of transposing two characters can be fixed, when they are adjacent, with the C-t command (`transpose-chars`). Normally, C-t transposes the two characters on either side of point. When given at the end of a line, rather than transposing the last character of the line with the newline, which would be useless, C-t transposes the last two characters on the line. So, if you catch your transposition error right away, you can fix it with just a C-t. If you don't catch it so fast, you must move the cursor back to between the two transposed characters. If you transposed a space with the last character of the word before it, the word motion commands are a good way of getting there. Otherwise, a reverse search (C-r) is often the best way. See section Searching and Replacement.

M-t (`transpose-words`) transposes the word before point with the word after point. It moves point forward over a word, dragging the word preceding or containing point forward as well. The punctuation characters between the words do not move. For example, `FOO, BAR' transposes into `BAR, FOO' rather than `BAR FOO,'.

C-M-t (`transpose-sexps`) is a similar command for transposing two expressions (see section Lists and Sexps), and C-x C-t (`transpose-lines`) exchanges lines. They work like M-t except in determining the division of the text into syntactic units.

A numeric argument to a transpose command serves as a repeat count: it tells the transpose command to move the character (word, sexp, line) before or containing point across several other characters (words, sexps, lines). For example, C-u 3 C-t moves the character before point forward across three other characters. It would change `f-!-oobar' into `oobf-!-ar'. This is equivalent to repeating C-t three times. C-u - 4 M-t moves the word before point backward across four words. C-u - C-M-t would cancel the effect of plain C-M-t.

A numeric argument of zero is assigned a special meaning (because otherwise a command with a repeat count of zero would do nothing): to transpose the character (word, sexp, line) ending after point with the one ending after the mark.

Case Conversion

M-- M-l

Convert last word to lower case. Note Meta-- is Meta-minus.

M-- M-u

Convert last word to all upper case.

M-- M-c

Convert last word to lower case with capital initial.

A very common error is to type words in the wrong case. Because of this, the word case-conversion commands M-l, M-u and M-c have a special feature when used with a negative argument: they do not move the cursor. As soon as you see you have mistyped the last word, you can simply case-convert it and go on typing. See section Case Conversion Commands.

Checking and Correcting Spelling

This section describes the commands to check the spelling of a single word or of a portion of a buffer. These commands work with the spelling checker program Ispell, which is not part of Emacs.

M-$
: Check and correct spelling of word at point (`ispell-word`).

M-TAB
: Complete the word before point based on the spelling dictionary (`ispell-complete-word`).

M-x ispell-buffer
: Check and correct spelling of each word in the buffer.

M-x ispell-region
: Check and correct spelling of each word in the region.

M-x ispell-message
: Check and correct spelling of each word in a draft mail message, excluding cited material.

M-x ispell-change-dictionary RET dict RET
: Restart the ispell process, using dict as the dictionary.

M-x ispell-kill-ispell
: Kill the Ispell subprocess.

To check the spelling of the word around or next to point, and optionally correct it as well, use the command M-$ (`ispell-word`). If the word is not correct, the command offers you various alternatives for what to do about it.

To check the entire current buffer, use M-x ispell-buffer. Use M-x ispell-region to check just the current region. To check spelling in an email message you are writing, use M-x ispell-message; that checks the whole buffer, but does not check material that is indented or appears to be cited from other messages.

Each time these commands encounter an incorrect word, they ask you what to do. It displays a list of alternatives, usually including several "near-misses"---words that are close to the word being checked. Then you must type a character. Here are the valid responses:

SPC
: Skip this word--continue to consider it incorrect, but don't change it here.

r new RET
: Replace the word (just this time) with new.

R new RET
: Replace the word with new, and do a `query-replace` so you can replace it elsewhere in the

buffer if you wish.

digit

Replace the word (just this time) with one of the displayed near-misses. Each near-miss is listed with a digit; type that digit to select it.

a

Accept the incorrect word--treat it as correct, but only in this editing session.

A

Accept the incorrect word--treat it as correct, but only in this editing session and for this buffer.

i

Insert this word in your private dictionary file so that Ispell will consider it correct it from now on, even in future sessions.

u

Insert a lower-case version of this word in your private dictionary file.

m

Like i, but you can also specify dictionary completion information.

l word RET

Look in the dictionary for words that match word. These words become the new list of "near-misses"; you can select one of them to replace with by typing a digit. You can use `*' in word as a wildcard.

C-g

Quit interactive spell checking. You can restart it again afterward with C-u M-$.

X

Same as C-g.

x

Quit interactive spell checking and move point back to where it was when you started spell checking.

q

Quit interactive spell checking and kill the Ispell subprocess.

C-l

Refresh the screen.

C-z

This key has its normal command meaning (suspend Emacs or iconify this frame).

The command `ispell-complete-word`, which is bound to the key M-TAB in Text mode and related modes, shows a list of completions based on spelling correction. Insert the beginning of a word, and then type M-TAB; the command displays a completion list window. To choose one of the completions listed, click Mouse-2 on it, or move the cursor there in the completions window and type RET. See section Text Mode.

Once started, the Ispell subprocess continues to run (waiting for something to do), so that subsequent

spell checking commands complete more quickly. If you want to get rid of the Ispell process, use M-ispell-kill-ispell. This is not usually necessary, since the process uses no time except when you do spelling correction.

Ispell uses two dictionaries: the standard dictionary and your private dictionary. The variable `ispell-dictionary` specifies the file name of the standard dictionary to use. A value of `nil` says to use the default dictionary. The command M-x ispell-change-dictionary sets this variable and then restarts the Ispell subprocess, so that it will use a different dictionary.

Go to the previous, next section.

Go to the previous, next section.

File Handling

The operating system stores data permanently in named files. So most of the text you edit with Emacs comes from a file and is ultimately stored in a file.

To edit a file, you must tell Emacs to read the file and prepare a buffer containing a copy of the file's text. This is called visiting the file. Editing commands apply directly to text in the buffer; that is, to the copy inside Emacs. Your changes appear in the file itself only when you save the buffer back into the file.

In addition to visiting and saving files, Emacs can delete, copy, rename, and append to files, keep multiple versions of them, and operate on file directories.

File Names

Most Emacs commands that operate on a file require you to specify the file name. (Saving and reverting are exceptions; the buffer knows which file name to use for them.) You enter the file name using the minibuffer (see section The Minibuffer). Completion is available, to make it easier to specify long file names. See section Completion.

For most operations, there is a default file name which is used if you type just RET to enter an empty argument. Normally the default file name is the name of the file visited in the current buffer; this makes it easy to operate on that file with any of the Emacs file commands.

Each buffer has a default directory, normally the same as the directory of the file visited in that buffer. When you enter a file name without a directory, the default directory is used. If you specify a directory in a relative fashion, with a name that does not start with a slash, it is interpreted with respect to the default directory. The default directory is kept in the variable `default-directory`, which has a separate value in every buffer.

For example, if the default file name is `` `/u/rms/gnu/gnu.tasks' `` then the default directory is `` `/u/rms/gnu/' ``. If you type just `foo', which does not specify a directory, it is short for `` `/u/rms/gnu/foo' ``. `../.login' would stand for `` `/u/rms/.login' ``. `new/foo' would stand for the file name `` `/u/rms/gnu/new/foo' ``.

The command M-x pwd prints the current buffer's default directory, and the command M-x cd sets it (to a value read using the minibuffer). A buffer's default directory changes only when the `cd` command is used. A file-visiting buffer's default directory is initialized to the directory of the file that is visited there. If you create a buffer with C-x b, its default directory is copied from that of the buffer that was current at the time.

The default directory actually appears in the minibuffer when the minibuffer becomes active to read a file name. This serves two purposes: it *shows* you what the default is, so that you can type a relative file name and know with certainty what it will mean, and it allows you to *edit* the default to specify a different directory. This insertion of the default directory is inhibited if the variable

`insert-default-directory` is set to `nil`.

Note that it is legitimate to type an absolute file name after you enter the minibuffer, ignoring the presence of the default directory name as part of the text. The final minibuffer contents may look invalid, but that is not so. For example, if the minibuffer starts out with `/usr/tmp/' and you add `/x1/rms/foo', you get `/usr/tmp//x1/rms/foo'; but Emacs ignores everything through the first slash in the double slash; the result is `/x1/rms/foo'. See section Minibuffers for File Names.

You can refer to files on other machines using a special file name syntax:

```
/host:filename
/user@host:filename
```

When you do this, Emacs uses the FTP program to read and write files on the specified host. It logs in through FTP using your user name or the name user. It may ask you for a password from time to time; this is used for logging in on host.

You can turn off the FTP file name feature by setting the variable `file-name-handler-alist` to `nil`.

`$' in a file name is used to substitute environment variables. For example, if you have used the shell command `export FOO=rms/hacks' to set up an environment variable named `FOO`, then you can use `/u/$FOO/test.c'` or `/u/${FOO}/test.c'` as an abbreviation for `/u/rms/hacks/test.c'`. The environment variable name consists of all the alphanumeric characters after the `$'; alternatively, it may be enclosed in braces after the `$'. Note that shell commands to set environment variables affect Emacs only if done before Emacs is started.

To access a file with `$' in its name, type `$$'. This pair is converted to a single `$' at the same time as variable substitution is performed for single `$'. The Lisp function that performs the substitution is called `substitute-in-file-name`. The substitution is performed only on file names read as such using the minibuffer.

Visiting Files

C-x C-f
: Visit a file (`find-file`).

C-x C-r
: Visit a file for viewing, without allowing changes to it (`find-file-read-only`).

C-x C-v
: Visit a different file instead of the one visited last (`find-alternate-file`).

C-x 4 C-f
: Visit a file, in another window (`find-file-other-window`). Don't change the selected window.

C-x 5 C-f

Visit a file, in a new frame (`find-file-other-frame`). Don't change the selected frame.

M-x auto-compression-mode

Toggle automatic uncompression and recompression for compressed files.

Visiting a file means copying its contents into an Emacs buffer so you can edit them. Emacs makes a new buffer for each file that you visit. We say that this buffer is visiting the file that it was created to hold. Emacs constructs the buffer name from the file name by throwing away the directory, keeping just the name proper. For example, a file named `` `/usr/rms/emacs.tex' `` would get a buffer named `emacs.tex'. If there is already a buffer with that name, a unique name is constructed by appending `<2>', `<3>', or so on, using the lowest number that makes a name that is not already in use.

Each window's mode line shows the name of the buffer that is being displayed in that window, so you can always tell what buffer you are editing.

The changes you make with editing commands are made in the Emacs buffer. They do not take effect in the file that you visited, or any place permanent, until you save the buffer. Saving the buffer means that Emacs writes the current contents of the buffer into its visited file. See section Saving Files.

If a buffer contains changes that have not been saved, we say the buffer is modified. This is important because it implies that some changes will be lost if the buffer is not saved. The mode line displays two stars near the left margin to indicate that the buffer is modified.

To visit a file, use the command C-x C-f (`find-file`). Follow the command with the name of the file you wish to visit, terminated by a RET.

The file name is read using the minibuffer (see section The Minibuffer), with defaulting and completion in the standard manner (see section File Names). While in the minibuffer, you can abort C-x C-f by typing C-g.

Your confirmation that C-x C-f has completed successfully is the appearance of new text on the screen and a new buffer name in the mode line. If the specified file does not exist and could not be created, or cannot be read, then you get an error, with an error message displayed in the echo area.

If you visit a file that is already in Emacs, C-x C-f does not make another copy. It selects the existing buffer containing that file. However, before doing so, it checks that the file itself has not changed since you visited or saved it last. If the file has changed, a warning message is printed. See section Protection against Simultaneous Editing.

What if you want to create a new file? Just visit it. Emacs prints `(New File)' in the echo area, but in other respects behaves as if you had visited an existing empty file. If you make any changes and save them, the file is created.

If the file you specify is actually a directory, C-x C-f invokes Dired, the Emacs directory browser so that you can "edit" the contents of the directory (see section Dired, the Directory Editor). Dired is a convenient way to delete, look at, or operate on the files in the directory. However, if the variable `find-file-run-dired` is `nil`, then it is an error to try to visit a directory.

If you visit a file that the operating system won't let you modify, Emacs makes the buffer read-only, so

that you won't go ahead and make changes that you'll have trouble saving afterward. You can make t buffer writable with C-x C-q (`vc-toggle-read-only`). See section Miscellaneous Buffer Operations.

Occasionally you might want to visit a file as read-only in order to protect yourself from entering changes accidentally; do so by visiting the file with the command C-x C-r (`find-file-read-only`).

If you visit a nonexistent file unintentionally (because you typed the wrong file name), use the C-x C-v command (`find-alternate-file`) to visit the file you really wanted. C-x C-v is similar to C-x C-f, but it kills the current buffer (after first offering to save it if it is modified). When it reads the file name to visit, it inserts the entire default file name in the buffer, with point just after the directory part; this is convenient if you made a slight error in typing the name.

C-x 4 f (`find-file-other-window`) is like C-x C-f except that the buffer containing the specified file is selected in another window. The window that was selected before C-x 4 f continues to show the same buffer it was already showing. If this command is used when only one window is being displayed, that window is split in two, with one window showing the same buffer as before, and the other one showing the newly requested file. See section Multiple Windows.

C-x 5 f (`find-file-other-frame`) is similar, but opens a new frame, or makes visible any existing frame showing the file you seek. This feature is available only when you are using a window system. See section Frames and X Windows.

Two special hook variables allow extensions to modify the operation of visiting files. Visiting a file that does not exist runs the functions in the list `find-file-not-found-hooks`; this variable holds a list of functions, and the functions are called one by one until one of them returns non-`nil`. Any visiting of a file, whether extant or not, expects `find-file-hooks` to contain a list of functions and calls them all, one by one. In both cases the functions receive no arguments. Of these two variables, `find-file-not-found-hooks` takes effect first. These variables are *not* normal hooks, and their names end in `-hooks' rather than `-hook' to indicate that fact. See section Hooks.

The command M-x auto-compression-mode toggles a mode in which visiting a compressed file automatically uncompresses it. (Editing the file and saving it automatically recompresses it.)

There are several ways to specify automatically the major mode for editing the file (see section How Major Modes are Chosen), and to specify local variables defined for that file (see section Local Variables in Files).

Saving Files

Saving a buffer in Emacs means writing its contents back into the file that was visited in the buffer.

C-x C-s

: Save the current buffer in its visited file (`save-buffer`).

C-x s

: Save any or all buffers in their visited files (`save-some-buffers`).

M-~
: Forget that the current buffer has been changed (`not-modified`).

C-x C-w
: Save the current buffer in a specified file (`write-file`).

M-x set-visited-file-name
: Change file the name under which the current buffer will be saved.

When you wish to save the file and make your changes permanent, type C-x C-s (`save-buffer`). After saving is finished, C-x C-s displays a message like this:

```
Wrote /u/rms/gnu/gnu.tasks
```

If the selected buffer is not modified (no changes have been made in it since the buffer was created or last saved), saving is not really done, because it would have no effect. Instead, C-x C-s displays a message like this in the echo area:

```
(No changes need to be saved)
```

The command C-x s (`save-some-buffers`) offers to save any or all modified buffers. It asks you what to do with each buffer. The possible responses are analogous to those of `query-replace`:

y
: Save this buffer and ask about the rest of the buffers.

n
: Don't save this buffer, but ask about the rest of the buffers.

!
: Save this buffer and all the rest with no more questions.

RET
: Terminate `save-some-buffers` without any more saving.

.
: Save this buffer, then exit `save-some-buffers` without even asking about other buffers.

C-r
: View the buffer that you are currently being asked about. When you exit View mode, you get back to `save-some-buffers`, which asks the question again.

C-h
: Display a help message about these options.

C-x C-c, the key sequence to exit Emacs, invokes `save-some-buffers` and therefore asks the same questions.

If you have changed a buffer but you do not want to save the changes, you should take some action to prevent it. Otherwise, each time you use C-x s or C-x C-c, you are liable to save this buffer by mistake. One thing you can do is type M-~ (`not-modified`), which clears out the indication that the buffer is

modified. If you do this, none of the save commands will believe that the buffer needs to be saved. (`~' is often used as a mathematical symbol for `not'; thus M-~ is `not', metafied.) You could also use `set-visited-file-name' (see below) to mark the buffer as visiting a different file name, one which is not in use for anything important. Alternatively, you can cancel all the changes made since the file was visited or saved, by reading the text from the file again. This is called reverting. See section Reverting a Buffer. You could also undo all the changes by repeating the undo command C-x u until you have undone all the changes; but reverting is easier.

M-x set-visited-file-name alters the name of the file that the current buffer is visiting. It reads the new file name using the minibuffer. Then it specifies the visited file name and changes the buffer name correspondingly (as long as the new name is not in use). `set-visited-file-name` does not save the buffer in the newly visited file; it just alters the records inside Emacs in case you do save later. It also marks the buffer as "modified" so that C-x C-s in that buffer *will* save.

If you wish to mark the buffer as visiting a different file and save it right away, use C-x C-w (`write-file`). It is precisely equivalent to `set-visited-file-name` followed by C-x C-s. C-x C-s used on a buffer that is not visiting with a file has the same effect as C-x C-w; that is, it reads a file name, marks the buffer as visiting that file, and saves it there. The default file name in a buffer that is not visiting a file is made by combining the buffer name with the buffer's default directory.

If Emacs is about to save a file and sees that the date of the latest version on disk does not match what Emacs last read or wrote, Emacs notifies you of this fact, because it probably indicates a problem caused by simultaneous editing and requires your immediate attention. See section Protection against Simultaneous Editing.

If the variable `require-final-newline` is non-`nil`, Emacs puts a newline at the end of any file that doesn't already end in one, every time a file is saved or written.

Backup Files

On most operating systems, rewriting a file automatically destroys all record of what the file used to contain. Thus, saving a file from Emacs throws away the old contents of the file--or it would, except that Emacs carefully copies the old contents to another file, called the backup file, before actually saving. (This assumes that the variable `make-backup-files` is non-`nil`. Backup files are not written if this variable is `nil`.) Emacs does not normally make backup files for files in `/tmp'.

At your option, Emacs can keep either a single backup file or a series of numbered backup files for each file that you edit.

Emacs makes a backup for a file only the first time the file is saved from one buffer. No matter how many times you save a file, its backup file continues to contain the contents from before the file was visited. Normally this means that the backup file contains the contents from before the current editing session; however, if you kill the buffer and then visit the file again, a new backup file will be made by the next save.

Single or Numbered Backups

If you choose to have a single backup file (this is the default), the backup file's name is constructed by appending `~' to the file name being edited; thus, the backup file for `eval.c' would be `eval.c~'.

If you choose to have a series of numbered backup files, backup file names are made by appending `.~', the number, and another `~' to the original file name. Thus, the backup files of `eval.c' would be called `eval.c.~1~', `eval.c.~2~', and so on, through names like `eval.c.~259~' and beyond.

If protection stops you from writing backup files under the usual names, the backup file is written as `%backup%~' in your home directory. Only one such file can exist, so only the most recently made such backup is available.

The choice of single backup or numbered backups is controlled by the variable `version-control`. Its possible values are

`t`
: Make numbered backups.

`nil`
: Make numbered backups for files that have numbered backups already. Otherwise, make single backups.

`never`
: Do not in any case make numbered backups; always make single backups.

You can set `version-control` locally in an individual buffer to control the making of backups for that buffer's file. For example, Rmail mode locally sets `version-control` to `never` to make sure that there is only one backup for an Rmail file. See section Local Variables.

If you set the environment variable `VERSION_CONTROL`, to tell various GNU utilities what to do with backup files, Emacs also obeys the environment variable by setting the Lisp variable `version-control` accordingly at startup. If the environment variable's value is `t' or `numbered', then `version-control` becomes `t`; if the value is `nil' or `existing', then `version-control` becomes `nil`; if it is `never' or `simple', then `version-control` becomes `never`.

For files under version control (see section Version Control), the variable `vc-make-backup-files` determines whether to make backup files. By default, it is `nil`, since backup files are redundant when you store all the previous versions in a version control system. See section Editing with Version Control.

Automatic Deletion of Backups

To prevent unlimited consumption of disk space, Emacs can delete numbered backup versions automatically. Generally Emacs keeps the first few backups and the latest few backups, deleting any in between. This happens every time a new backup is made.

The two variables `kept-old-versions` and `kept-new-versions` control this deletion. Their values are, respectively the number of oldest (lowest-numbered) backups to keep and the number of newest (highest-numbered) ones to keep, each time a new backup is made. Recall that these values are used just after a new backup version is made; that newly made backup is included in the count in

`kept-new-versions`. By default, both variables are 2.

If `delete-old-versions` is non-`nil`, the excess middle versions are deleted without a murmur. If it is `nil`, the default, then you are asked whether the excess middle versions should really be deleted.

Dired's . (Period) command can also be used to delete old versions. See section Deleting Files with Dired.

Copying vs. Renaming

Backup files can be made by copying the old file or by renaming it. This makes a difference when the old file has multiple names. If the old file is renamed into the backup file, then the alternate names become names for the backup file. If the old file is copied instead, then the alternate names remain names for the file that you are editing, and the contents accessed by those names will be the new contents.

The method of making a backup file may also affect the file's owner and group. If copying is used, these do not change. If renaming is used, you become the file's owner, and the file's group becomes the default (different operating systems have different defaults for the group).

Having the owner change is usually a good idea, because then the owner always shows who last edited the file. Also, the owners of the backups show who produced those versions. Occasionally there is a file whose owner should not change; it is a good idea for such files to contain local variable lists to set `backup-by-copying-when-mismatch` locally (see section Local Variables in Files).

The choice of renaming or copying is controlled by three variables. Renaming is the default choice. If the variable `backup-by-copying` is non-`nil`, copying is used. Otherwise, if the variable `backup-by-copying-when-linked` is non-`nil`, then copying is used for files that have multiple names, but renaming may still used when the file being edited has only one name. If the variable `backup-by-copying-when-mismatch` is non-`nil`, then copying is used if renaming would cause the file's owner or group to change.

Protection against Simultaneous Editing

Simultaneous editing occurs when two users visit the same file, both make changes, and then both save them. If nobody were informed that this was happening, whichever user saved first would later find that his changes were lost. On some systems, Emacs notices immediately when the second user starts to change the file, and issues an immediate warning.

For the sake of systems where that is not possible, and in case someone else proceeds to change the file despite the warning, Emacs also checks when the file is saved, and issues a second warning if you are about to overwrite a file containing another user's changes. You can prevent loss of the other user's work by taking the proper corrective action at that time.

When you make the first modification in an Emacs buffer that is visiting a file, Emacs records that the file is locked by you. (It does this by writing another file in a directory reserved for this purpose.) The lock is removed when you save the changes. The idea is that the file is locked whenever an Emacs buffer visiting it has unsaved changes.

If you begin to modify the buffer while the visited file is locked by someone else, this constitutes a collision. When Emacs detects a collision, it asks you what to do, by calling the Lisp function `ask-user-about-lock`. You can redefine this function for the sake of customization. The standard definition of this function asks you a question and accepts three possible answers:

s
: Steal the lock. Whoever was already changing the file loses the lock, and you gain the lock.

p
: Proceed. Go ahead and edit the file despite its being locked by someone else.

q
: Quit. This causes an error (`file-locked`) and the modification you were trying to make in the buffer does not actually take place.

Note that locking works on the basis of a file name; if a file has multiple names, Emacs does not realize that the two names are the same file and cannot prevent two users from editing it simultaneously under different names. However, basing locking on names means that Emacs can interlock the editing of new files that will not really exist until they are saved.

Some systems are not configured to allow Emacs to make locks. On these systems, Emacs cannot detect trouble in advance, but it still can detect the collision when you try to save a file and overwrite someone else's changes.

Every time Emacs saves a buffer, it first checks the last-modification date of the existing file on disk to verify that it has not changed since the file was last visited or saved. If the date does not match, it implies that changes were made in the file in some other way, and these changes are about to be lost if Emacs actually does save. To prevent this, Emacs prints a warning message and asks for confirmation before saving. Occasionally you will know why the file was changed and know that it does not matter; then you can answer yes and proceed. Otherwise, you should cancel the save with C-g and investigate the situation.

The first thing you should do when notified that simultaneous editing has already taken place is to list the directory with C-u C-x C-d (see section File Directories). This shows the file's current author. You should attempt to contact him to warn him not to continue editing. Often the next step is to save the contents of your Emacs buffer under a different name, and use `diff` to compare the two files.

Simultaneous editing checks are also made when you visit with C-x C-f a file that is already visited and when you start to modify a file. This is not strictly necessary, but it can cause you to find out about the collision earlier, when perhaps correction takes less work.

Reverting a Buffer

If you have made extensive changes to a file and then change your mind about them, you can get rid of them by reading in the previous version of the file. To do this, use M-x revert-buffer, which operates on the current buffer. Since reverting a buffer unintentionally could lose a lot of work, you must confirm this command with yes.

`revert-buffer` keeps point at the same distance (measured in characters) from the beginning of the file. If the file was edited only slightly, you will be at approximately the same piece of text after reverting as before. If you have made drastic changes, the same value of point in the old file may address a totally different piece of text.

Reverting marks the buffer as "not modified" until another change is made.

Some kinds of buffers whose contents reflect data bases other than files, such as Dired buffers, can also be reverted. For them, reverting means recalculating their contents from the appropriate data base. Buffers created randomly with C-x b cannot be reverted; `revert-buffer` reports an error when asked to do so.

Auto-Saving: Protection Against Disasters

Emacs saves all the visited files from time to time (based on counting your keystrokes) without being asked. This is called auto-saving. It prevents you from losing more than a limited amount of work if the system crashes.

When Emacs determines that it is time for auto-saving, each buffer is considered, and is auto-saved if auto-saving is turned on for it and it has been changed since the last time it was auto-saved. The message `Auto-saving...' is displayed in the echo area during auto-saving, if any files are actually auto-saved. Errors occurring during auto-saving are caught so that they do not interfere with the execution of commands you have been typing.

Auto-Save Files

Auto-saving does not normally save in the files that you visited, because it can be very undesirable to save a program that is in an inconsistent state when you have made half of a planned change. Instead, auto-saving is done in a different file called the auto-save file, and the visited file is changed only when you request saving explicitly (such as with C-x C-s).

Normally, the auto-save file name is made by appending `#' to the front and rear of the visited file name. Thus, a buffer visiting file `` `foo.c' `` is auto-saved in a file `` `#foo.c#' ``. Most buffers that are not visiting files are auto-saved only if you request it explicitly; when they are auto-saved, the auto-save file name is made by appending `#%' to the front and `#' to the rear of buffer name. For example, the `*mail*' buffer in which you compose messages to be sent is auto-saved in a file named `` `#%*mail*#' ``. Auto-save file names are made this way unless you reprogram parts of Emacs to do something different (the functions `make-auto-save-file-name` and `auto-save-file-name-p`). The file name to be used for auto-saving in a buffer is calculated when auto-saving is turned on in that buffer.

When you delete a substantial part of the text in a large buffer, auto save turns off temporarily in that buffer. This is because if you deleted the text unintentionally, you might find the auto-save file more useful if it contains the deleted text. To reenable auto-saving after this happens, save the buffer with C-x C-s, or use C-u 1 M-x auto-save.

If you want auto-saving to be done in the visited file, set the variable `auto-save-visited-file-name` to be non-`nil`. In this mode, there is really no difference

between auto-saving and explicit saving.

A buffer's auto-save file is deleted when you save the buffer in its visited file. To inhibit this, set the variable `delete-auto-save-files` to `nil`. Changing the visited file name with C-x C-w or `set-visited-file-name` renames any auto-save file to go with the new visited name.

Controlling Auto-Saving

Each time you visit a file, auto-saving is turned on for that file's buffer if the variable `auto-save-default` is non-`nil` (but not in batch mode; see section Entering and Exiting Emacs). The default for this variable is `t`, so auto-saving is the usual practice for file-visiting buffers. Auto-saving can be turned on or off for any existing buffer with the command M-x auto-save-mode. Like other minor mode commands, M-x auto-save-mode turns auto-saving on with a positive argument, off with a zero or negative argument; with no argument, it toggles.

Emacs does auto-saving periodically based on counting how many characters you have typed since the last time auto-saving was done. The variable `auto-save-interval` specifies how many characters there are between auto-saves. By default, it is 300.

Auto-saving also takes place when you stop typing for a while. The variable `auto-save-timeout` says how many seconds Emacs should wait before it does an auto save (and perhaps also a garbage collection). (The actual time period is longer if the current buffer is long; this is a heuristic which aims to keep out of your way when you are editing long buffers in which auto-save takes an appreciable amount of time.) Auto-saving during idle periods accomplishes two things: first, it makes sure all your work is saved if you go away from the terminal for a while; second, it may avoid some auto-saving while you are actually typing.

Emacs also does auto-saving whenever it gets a fatal error. This includes killing the Emacs job with a shell command such as `kill %emacs', or disconnecting a phone line or network connection.

You can request an auto-save explicitly with the command M-x do-auto-save.

Recovering Data from Auto-Saves

You can use the contents of an auto-save file to recover from a loss of data with the command M-x recover-file RET file RET. This visits file and then (after your confirmation) restores the contents from from its auto-save file `` `#file#' ``. You can then save with C-x C-s to put the recovered text into file itself. For example, to recover file `` `foo.c' `` from its auto-save file `` `#foo.c#' ``, do:

```
M-x recover-file RET foo.c RET
yes RET
C-x C-s
```

Before asking for confirmation, M-x recover-file displays a directory listing describing the specified file and the auto-save file, so you can compare their sizes and dates. If the auto-save file is older, M-x recover-file does not offer to read it.

If Emacs or the computer crashes, you can recover all the files you were editing from their auto save with the command M-x recover-session. This first shows you a list of recorded interrupted sessions. Move point to the one you choose, and type C-c C-c.

Then `recover-session` asks about each of the files that were being edited during that session, asking whether to recover that file. If you answer y, it calls `recover-file`, which works in its normal fashion. It shows the dates of the original file and its auto-save file, and asks once again whether to recover that file.

When `recover-session` is done, the files you've chosen to recover are present in Emacs buffers. You should then save them. Only this--saving them--updates the files themselves.

Interrupted sessions are recorded for later recovery in files named `` `~/.saves-pid-hostname' ``. The `~/.saves' portion of these names comes from the value of `auto-save-list-file-prefix`. You can arrange to record sessions in a different place by setting that variable in your `` `.emacs' `` file, but you'll have to redefine `recover-session` as well to make it look in the new place. If you set `auto-save-list-file-prefix` to `nil` in your `` `.emacs' `` file, sessions are not recorded for recovery.

File Name Aliases

Symbolic links and hard links both make it possible for several file names to refer to the same file. Hard links are alternate names that refer directly to the file; all the names are equally valid, and no one of them is preferred. By contrast, a symbolic link is a kind of defined alias: when `` `foo' `` is a symbolic link to `` `bar' ``, you can use either name to refer to the file, but `` `bar' `` is the real name, while `` `foo' `` is just an alias. More complex cases occur when symbolic links point to directories.

If you visit two names for the same file, normally Emacs makes two different buffers, but it warns you about the situation.

If you wish to avoid visiting the same file in two buffers under different names, set the variable `find-file-existing-other-name` to a non-`nil` value. Then `find-file` uses the existing buffer visiting the file, no matter which of the file's names you specify.

If the variable `find-file-visit-truename` is non-`nil`, then the file name recorded for a buffer is the file's truename (made by replacing all symbolic links with their target names), rather than the name you specify. Setting `find-file-visit-truename` also implies the effect of `find-file-existing-other-name`.

Version Control

Version control systems are packages that can record multiple versions of a source file, usually storing the unchanged parts of the file just once. Version control systems also record history information such as the creation time of each version, who created it, and a description of what was changed in that version.

The Emacs version control commands work with three version control systems--RCS, CVS and SCCS. The GNU project recommends RCS and CVS, which are free software and available from the Free

Software Foundation.

Supported Version Control Systems

VC currently works with three different version control systems or "back ends": RCS, CVS, and SCCS.

RCS is a free version control system that is available from the Free Software Foundation. It is perhaps the most mature of the supported back ends, and the VC commands are conceptually closest to RCS. Almost everything you can do with RCS can be done through VC.

CVS is built on top of RCS, and extends the features of RCS, allowing for more sophisticated release management, and concurrent multi-user development. VC supports basic editing operations under CVS, but for some less common tasks you still need to call CVS from the command line. Note also that before using CVS you must set up a repository, which is a subject too complex to treat here. See section Using VC with CVS.

SCCS is a proprietary but widely used version control system. In terms of capabilities, it is the weakest of the the three that VC supports. VC compensates for certain features missing in SCCS (snapshots, for example) by implementing them itself, but some other VC features, such as multiple branches, are not available with SCCS. You should use SCCS only if for some reason you cannot use RCS.

Concepts of Version Control

When a file is under version control, we also say that it is registered in the version control system. Each registered file has a corresponding master file which represents the file's present state plus its change history, so that you can reconstruct from it either the current version or any specified earlier version. Usually the master file also records a log entry for each version describing what was changed in that version.

The file that is maintained under version control is sometimes called the work file corresponding to its master file.

To examine a file, you check it out. This extracts a version of the source file (typically, the most recent) from the master file. If you want to edit the file, you must check it out locked. Only one user can do this at a time for any given source file. (This kind of locking is completely unrelated to the locking that Emacs uses to detect simultaneous editing of a file.)

When you are done with your editing, you must check in the new version. This records the new version in the master file, and unlocks the source file so that other people can lock it and thus modify it.

Checkin and checkout are the basic operations of version control. You can do both of them with a single Emacs command: C-x C-q (`vc-toggle-read-only`).

There are variants of this basic pattern, though. CVS, for example, has no such thing as locking, and therefore you can normally edit files right away, without having to check them out first. See section Using VC with CVS. With RCS, you can optionally select non-strict locking for a particular source file; then you can edit the file in Emacs without explicitly locking it.

A snapshot is a coherent collection of versions of the various files that make up a program. See sect
Snapshots.

Editing with Version Control

These are the commands for editing a file maintained with version control:

C-x C-q

C-x v v

Check the visited file in or out.

C-x v u

Revert the buffer and the file to the last checked in version.

C-x v c

Remove the last-entered change from the master for the visited file. This undoes your last check-in.

C-x v i

Register the visited file for version control.

(C-x v is the prefix key for version control commands; all of these commands except for C-x C-q start with C-x v.)

Check-Out

When you want to modify a file maintained with version control, type C-x C-q (`vc-toggle-read-only`). This checks out the file, and tells RCS or SCCS to lock the file. This means making the file writable for you (but not for anyone else).

If you specify a prefix argument (C-u C-x C-q) for checkout, Emacs asks you for a version number, and checks out that version *unlocked*. This lets you move to old versions, or existing branches of the file (see section Multiple Branches of a File). You can then start editing the selected version by typing C-x C-q again. (If you edit an old version of a file this way, checking it in again creates a new branch.)

Under CVS, you normally don't need to check out files explicitly. CVS does not have locking; multiple users can edit their copies of a file whenever they want. (If two users make conflicting changes, they need to reconcile their changes when checking them in.) We therefore say that an implicit check-out happens when you make the first change in the file.

CVS has an alternative mode in which explicit check-out is required. And RCS has an alternative mode called non-strict locking in which explicit check-out is not required. Selecting these modes is done outside of VC, but once you have selected them, VC obeys them. With RCS, you can select non-strict locking for a particular file using the `rcs -U' command. See section Using VC with CVS, for an explanation of how to do this with CVS.

Check-In

When you are finished editing the file, type C-x C-q again. When used on a file that is checked out, t command checks the file in. But check-in does not start immediately; first, you must enter the log entry---a description of the changes in the new version. C-x C-q pops up a buffer for you to enter this in. When you are finished typing in the log entry, type C-c C-c to terminate it; this is when actual check-in takes place. See section Log Entries.

With RCS and SCCS, a checked-out file is also locked, which means it is writable for you, but not for anyone else. As long as you own the lock on the file, nobody else can modify it, and nobody can check in any changes to that particular version. Checking in your changes unlocks the file, so that other users can lock it and modify it.

CVS, on the contrary, doesn't have a concept of locking. The working files are always modifiable, allowing concurrent development, with possible conflicts being resolved at check-in time. See section Using VC with CVS.

To specify the version number for the new version, type C-u C-x C-q to check in a file. Then Emacs asks you for the new version number in the minibuffer. This can be used to create a new branch of the file (see section Multiple Branches of a File), or to increment the file's major version number.

It is not impossible to lock a file that someone else has locked. If you try to check out a file that is locked, C-x C-q asks you whether you want to "steal the lock." If you say yes, the file becomes locked by you, but a message is sent to the person who had formerly locked the file, to inform him of what has happened. The mode line indicates that a file is locked by someone else by displaying the login name of that person, before the version number.

Registering a File for Version Control

C-x v i

Register the visited file for version control.

You can put any file under version control by simply visiting it, and then typing C-x v i (`vc-register`). After C-x v i, the file is unlocked and read-only. Type C-x C-q if you wish to start editing it.

When you register the file, Emacs must choose which version control system to use for it. You can specify your choice explicitly by setting `vc-default-back-end` to `RCS`, `CVS` or `SCCS`. Otherwise, if there is a subdirectory named `` `RCS' ``, `` `SCCS' ``, or `` `CVS' ``, Emacs uses the corresponding version control system. In the absence of any specification, the default choice is RCS if RCS is installed, otherwise SCCS.

After registering a file with CVS, you must subsequently commit the initial version by typing C-x C-q. See section Using VC with CVS.

The initial version number for a newly registered file is 1.1, by default. To specify a different number, give C-x v i a numeric argument; then it reads the initial version number using the minibuffer.

If `vc-initial-comment` is non-`nil`, C-x v i reads an initial comment (much like a log entry) to describe the purpose of this source file.

Undoing Version Control Actions

C-x v u
: Revert the buffer and the file to the last checked in version.

C-x v c
: Remove the last-entered change from the master for the visited file. This undoes your last checkin.

If you want to discard your current set of changes and revert to the last version checked in, use C-x v u (`vc-revert-buffer`). This cancels your last check-out, leaving the file unlocked. If you want to make a different set of changes, you must first check the file out again. C-x v u requires confirmation, unless it sees that you haven't made any changes since the last checked-in version.

C-x v u is also the command to use to unlock a file if you lock it and then decide not to change it.

You can cancel a change after checking it in, with C-x v c (`vc-cancel-version`). This command discards all record of the most recent checked in version. C-x v c also offers to revert your work file and buffer to the previous version (the one that precedes the version that is deleted). If you say no, then VC keeps your changes in the buffer and locks the file.

The no-revert option is useful when you have checked in a change and then discover a trivial error in it; you can cancel the erroneous check-in, fix the error, and check the file in again.

When C-x v c does not revert the buffer, it unexpands all version control headers in the buffer instead (see section Inserting Version Control Headers). This is because the buffer no longer corresponds to any existing version. If you check it in again, the checkin process will expand the headers properly for the new version number.

However, it is impossible to unexpand the RCS `Log' header automatically. If you use that header feature, you have to unexpand it by hand--by deleting the entry for the version that you just canceled.

Be careful when invoking C-x v c, as it is easy to throw away a lot of work with it. To help you be careful, this command always requires confirmation with yes. Note also that this command is disabled under CVS, because canceling versions is very dangerous and discouraged with this back end.

The VC Mode Line

When you visit a file that is under version control, the mode line indicates the current status of the file: the name of the version control back end system, the locking state, and the version.

The locking state is displayed as a single character, which can be either `-' or `:'. `-' means the file is not locked or not modified by you. Once you lock the file, the state indicator changes to `:'. If the file is locked by someone else, that user's name appears after the version number.

For example, `RCS-1.3' means you are looking at RCS version 1.3, which is not locked. `RCS:1.3' means that you have locked the file, and possibly already changed it. `RCS:jim:1.3' means that the file is locked by jim.

Using VC with CVS

In CVS, files are never locked. Two users can check out the same file at the same time; each user has a separate copy and can edit it. Work files are always writable; once you have one, you need not type a VC command to start editing the file. You can edit it at any time.

When using RCS and SCCS, you normally use C-x C-q twice for each change; once before the change, for checkout, and once after, for checkin. With CVS, it's different: you normally use C-x C-q just once for each change, to commit the change when it is done. The work file remains writable, so you can begin editing again with no special commands.

One way to understand this is that VC does an implicit check-out when you save the modified file for the first time. VC indicates this on the mode line: the status indicator changes from `-' to `:' as soon as you save a modified version, telling you that you are not in sync with the repository anymore (see section The VC Mode Line). The file stays "checked out" until you check it back in, even if you kill the buffer and visit the file again.

If, instead, you would like to use explicit check-out with CVS, set the `CVSREAD` environment variable to some value. (It does not matter what value you use.) CVS then makes your work files read-only by default, and VC requires you to check them out explicitly with C-x C-q. When setting `CVSREAD` for the first time, make sure to check out all your modules anew, so that the file protections are set correctly.

VC does not provide a way to check out a working copy of an existing file in the repository. You have to use the CVS shell commands to do that. Once you have a work file, you can start using VC for that file.

CVS terminology speaks of committing a change rather than checking it in. But in practical terms they work the same way: Emacs asks you to type in a log entry, and you finish it with C-c C-c.

When you try to commit a change in a file, but someone else has committed another change in the meanwhile, that creates a conflict. VC detects this situation and offers to merge your changes and those of the other user, creating a new local version of the file, which you can then commit to the repository. This works smoothly if the changes are in different parts of the file, although it is wise to check the resulting file for semantic consistency.

However, if you and the other user changed the same parts of the file, the conflict cannot be resolved automatically. In this case, CVS inserts both variants of the conflicting regions into your working file, and puts so-called conflict markers around them. They indicate how the region looks in the respective user's version. You must resolve the conflict manually, for example by choosing one of the two variants and deleting the other one (and the conflict markers). Then you can commit the resulting file into the repository. The example below shows how a conflict region looks; the file is called `name' and the current repository version with user B's changes in it is 1.11.

```
<<<<<<< name
  User A's version
=======
  User B's version
>>>>>>> 1.11
```

You can turn off use of VC for CVS-managed files by setting the variable `vc-handle-cvs` to `nil`. If

you do this, Emacs treats these files as if they were not managed, and the VC commands are not available for them. You must do all CVS operations manually.

Log Entries

When you're editing an initial comment or log entry for inclusion in a master file, finish your entry by typing C-c C-c.

C-c C-c

Finish the comment edit normally (`vc-finish-logentry`). This finishes check-in.

To abort check-in, just **don't** type C-c C-c in that buffer. You can switch buffers and do other editing. As long as you don't try to check in another file, the entry you were editing remains in its buffer, and you can go back to that buffer at any time to complete the check-in.

If you change several source files for the same reason, it is often convenient to specify the same log entry for many of the files. To do this, use the history of previous log entries. The commands M-n, M-p, M-s and M-r for doing this work just like the minibuffer history commands (except that these versions are used outside the minibuffer).

Each time you check in a file, the log entry buffer is put into VC Log mode, which involves running two hooks: `text-mode-hook` and `vc-log-mode-hook`. See section Hooks.

Change Logs and VC

If you use RCS for a program and also maintain a change log file for it (see section Change Logs), you can generate change log entries automatically from the version control log entries:

C-x v a

Visit the current directory's change log file and create new entries for versions checked in since the most recent entry in the change log file (`vc-update-change-log`).

This command works with RCS only; it does not work with CVS or SCCS.

For example, suppose the first line of `` `ChangeLog' `` is dated 10 April 1992, and that the only check-in since then was by Nathaniel Bowditch to `` `rcs2log' `` on 8 May 1992 with log text `Ignore log messages that start with `#'.'. Then C-x v a visits `` `ChangeLog' `` and inserts text like this:

@medbreak

```
Fri May  8 21:45:00 1992  Nathaniel Bowditch  <nat@apn.org>

        * rcs2log: Ignore log messages that start with `#'.
```

@medbreak

You can then edit the new change log entry further as you wish.

Normally, the log entry for file `` `foo' `` is displayed as `* foo: text of log entry'. The `:' after `` `foo' `` is

omitted if the text of the log entry starts with `(functionname): '. For example, if the log entry for `vc.el' is `(vc-do-command): Check call-process status.', then the text in `ChangeLog' looks like this:

@medbreak

```
Wed May  6 10:53:00 1992  Nathaniel Bowditch  <nat@apn.org>

        * vc.el (vc-do-command): Check call-process status.
```

@medbreak

When C-x v a adds several change log entries at once, it groups related log entries together if they all are checked in by the same author at nearly the same time. If the log entries for several such files all have the same text, it coalesces them into a single entry. For example, suppose the most recent checkins have the following log entries:

```
* For `vc.texinfo': `Fix expansion typos.'
* For `vc.el': `Don't call expand-file-name.'
* For `vc-hooks.el': `Don't call expand-file-name.'
```

They appear like this in `ChangeLog':

@medbreak

```
Wed Apr  1 08:57:59 1992  Nathaniel Bowditch  <nat@apn.org>

        * vc.texinfo: Fix expansion typos.

        * vc.el, vc-hooks.el: Don't call expand-file-name.
```

@medbreak

Normally, C-x v a separates log entries by a blank line, but you can mark several related log entries to be clumped together (without an intervening blank line) by starting the text of each related log entry with a label of the form `{clumpname} '. The label itself is not copied to `ChangeLog'. For example, suppose the log entries are:

```
* For `vc.texinfo': `{expand} Fix expansion typos.'
* For `vc.el': `{expand} Don't call expand-file-name.'
* For `vc-hooks.el': `{expand} Don't call expand-file-name.'
```

Then the text in `ChangeLog' looks like this:

@medbreak

```
Wed Apr  1 08:57:59 1992  Nathaniel Bowditch  <nat@apn.org>
```


```
        * vc.texinfo: Fix expansion typos.
        * vc.el, vc-hooks.el: Don't call expand-file-name.
```

@medbreak

A log entry whose text begins with `#' is not copied to `ChangeLog'. For example, if you merely fix some misspellings in comments, you can log the change with an entry beginning with `#' to avoid putting such trivia into `ChangeLog'.

Examining And Comparing Old Versions

C-u C-x C-q version RET

Select version version as the current work file version.

C-x v ~ version RET

Examine version version of the visited file, in a buffer of its own.

C-x v =

Compare the current buffer contents with the latest checked-in version of the file.

C-u C-x v = file RET oldvers RET newvers RET

Compare the specified two versions of file.

There are two ways to work with an old version of a file. You can make the old version your current work file, for example if you want to reproduce a former stage of development, or if you want to create a branch from the old version (see section Multiple Branches of a File). To do this, visit the file and type C-u C-x C-q version RET. (This works only with RCS.)

If you want only to examine an old version, without changing your work file, visit the file and then type C-x v ~ version RET (`vc-version-other-window`). This puts the text of version version in a file named `filename.~version~', and visits it in its own buffer in a separate window.

To compare two versions of a file, use the command C-x v = (`vc-diff`). Plain C-x v = compares the current buffer contents (saving them in the file if necessary) with the last checked-in version of the file. C-u C-x v =, with a numeric argument, reads a file name and two version numbers, then compares those versions of the specified file.

If you supply a directory name instead of the name of a work file, this command compares the two specified versions of all registered files in that directory and its subdirectories. You can also specify a snapshot name (see section Snapshots) instead of one or both version numbers.

You can specify a checked-in version by its number; an empty input specifies the current contents of the work file (which may be different from all the checked-in versions).

This command works by running the `diff` utility, getting the options from the variable `diff-switches`. It displays the output in a special buffer in another window. Unlike the M-x diff command, C-x v = does not try to locate the changes in the old and new versions. This is because normally one or both versions do not exist as files when you compare them; they exist only in the records of the master file. See section Comparing Files, for more information about M-x diff.

Multiple Branches of a File

One use of version control is to maintain multiple "current" versions of a file. For example, you might have different versions of a program in which you are gradually adding various unfinished new features. Each such independent line of development is called a branch. VC allows you to create branches, and switch between existing branches. Note, however, that branches are supported only with RCS.

A file's main line of development is usually called the trunk. The versions on the trunk are normally numbered 1.1, 1.2, 1.3, etc. At any such version, you may start an independent branch. A branch starting at version 1.2 would have version number 1.2.1.1. Consecutive versions on this branch would have numbers 1.2.1.2, 1.2.1.3, 1.2.1.4, and so on. If there is a second branch also starting at version 1.2; it would consist of versions 1.2.2.1, 1.2.2.2, 1.2.2.3, and so on.

If you omit the final component of a version number, that is called a branch number. It refers to the highest existing version on that branch. The branches in the example above have branch numbers 1.2.1 and 1.2.2.

A version which is the last in its branch is called a head version.

Switching between Branches

To switch between branches, type C-u C-x C-q and specify the version number you want to select. This version is then checked out *unlocked* (write-protected), so you can examine it before really checking it out. Switching branches in this way is allowed only when the file is not locked.

You may omit the minor version number, thus giving only the branch number; this takes you to the highest version on the indicated branch. If you only type RET, Emacs goes to the highest version on the trunk.

After you have switched to any branch (including the main branch), you stay on it for subsequent VC commands, until you explicitly select some other branch.

Creating New Branches

To create a new branch from a head version (one that is the latest in the branch that contains it), first select that version if necessary, lock it with C-x C-q, and make whatever changes you want. Then, when you check in the changes, use C-u C-x C-q. This lets you specify the version number for the new version. You should specify a suitable branch number for a branch starting at the current version. For example, if the current version is 2.5, the branch number should be 2.5.1, 2.5.2, and so on, depending on the number of existing branches at that point.

To create a new branch at an older version (one that is no longer the head of a branch), first select that version, then lock it with C-x C-q. You'll be asked to confirm, when you lock the old version, that you really mean to create a new branch--if you say no, you'll be offered a chance to lock the latest version instead.

Then make your changes and type C-x C-q again to check in a new version. This automatically creates a new branch starting from the selected version. You need not specially request a new branch, because

that's the only way to add a new version at a point that is not the head of a branch.

After the branch is created, you "stay" on it. That means that subsequent checkouts and checkins create new versions on that branch. To leave the branch, you must explicitly select a different version with C-u C-x C-q for checkout.

Multi-User Branching

It is sometimes useful for multiple developers to work simultaneously on different branches of a file. This is possible if you create multiple source directories. Each source directory should have a link named `RCS' which points to a common directory of RCS master files. Then each source directory can have its own choice of versions checked out, but all share the same common RCS records.

This technique works reliably and automatically, provided that the source files contain RCS version headers (see section Inserting Version Control Headers). The headers enable Emacs to be sure, at all times, which version number is present in the work file.

If the files do not have version headers, you must instead tell Emacs explicitly in each session which branch you are working on. To do this, first find the file, then type C-u C-x C-q and specify the correct branch number. This ensures that Emacs knows which branch it is using during this particular editing session.

VC Status Commands

To view the detailed version control status and history of a file, type C-x v l (`vc-print-log`). It displays the history of changes to the current file, including the text of the log entries. The output appears in a separate window.

When you are working on a large program, it's often useful to find all the files that are currently locked, or all the files maintained in version control at all. You can use C-x v d (`vc-directory`) to show all the locked files in or beneath a certain directory. This includes all files that are locked by any user. C-u C-x v d lists all files in or beneath the specified directory that are maintained with version control.

The list of files is displayed as a buffer that uses an augmented Dired mode. The names of the users locking various files are shown (in parentheses) in place of the owner and group. (With CVS, a more detailed status is shown for each file.) All the normal Dired commands work in this buffer. Most interactive VC commands work also, and apply to the file name on the current line.

The C-x v v command (`vc-next-action`), when used in the augmented Dired buffer, operates on all the marked files (or the file on the current line). If it operates on more than one file, it handles each file according to its current state; thus, it may check out one file and check in another (because it is already checked out). If it has to check in any files, it reads a single log entry, then uses that text for all the files being checked in. This can be convenient for registering or checking in several files at once, as part of the same change.

Renaming VC Work Files and Master Files

When you rename a registered file, you must also rename its master file correspondingly to get proper results. Use `vc-rename-file` to rename the source file as you specify, and rename its master file accordingly. It also updates any snapshots (see section Snapshots) that mention the file, so that they use the new name; despite this, the snapshot thus modified may not completely work (see section Snapshot Caveats).

You cannot use `vc-rename-file` on a file that is locked by someone else.

Snapshots

A snapshot is a named set of file versions (one for each registered file) that you can treat as a unit. One important kind of snapshot is a release, a (theoretically) stable version of the system that is ready for distribution to users.

Making and Using Snapshots

There are two basic commands for snapshots; one makes a snapshot with a given name, the other retrieves a named snapshot.

`C-x v s name RET`

Define the last saved versions of every registered file in or under the current directory as a snapshot named name (`vc-create-snapshot`).

`C-x v r name RET`

Check out all registered files at or below the current directory level using whatever versions correspond to the snapshot name (`vc-retrieve-snapshot`).

This command reports an error if any files are locked at or below the current directory, without changing anything; this is to avoid overwriting work in progress.

A snapshot uses a very small amount of resources--just enough to record the list of file names and which version belongs to the snapshot. Thus, you need not hesitate to create snapshots whenever they are useful.

You can give a snapshot name as an argument to C-x v = or C-x v ~ (see section Examining And Comparing Old Versions). Thus, you can use it to compare a snapshot against the current files, or two snapshots against each other, or a snapshot against a named version.

Snapshot Caveats

VC's snapshot facilities are modeled on RCS's named-configuration support. They use RCS's native facilities for this, so under VC snapshots made using RCS are visible even when you bypass VC.

For SCCS, VC implements snapshots itself. The files it uses contain name/file/version-number triples. These snapshots are visible only through VC.

A snapshot is a set of checked-in versions. So make sure that all the files are checked in and not locked when you make a snapshot.

File renaming and deletion can create some difficulties with snapshots. This is not a VC-specific problem, but a general design issue in version control systems that no one has solved very well yet.

If you rename a registered file, you need to rename its master along with it (the command `vc-rename-file` does this automatically). If you are using SCCS, you must also update the records of the snapshot, to mention the file by its new name (`vc-rename-file` does this, too). An old snapshot that refers to a master file that no longer exists under the recorded name is invalid; VC can no longer retrieve it. It would be beyond the scope of this manual to explain enough about RCS and SCCS to explain how to update the snapshots by hand.

Using `vc-rename-file` makes the snapshot remain valid for retrieval, but it does not solve all problems. For example, some of the files in the program probably refer to others by name. At the very least, the makefile probably mentions the file that you renamed. If you retrieve an old snapshot, the renamed file is retrieved under its new name, which is not the name that the makefile expects. So the program won't really work as retrieved.

Inserting Version Control Headers

Sometimes it is convenient to put version identification strings directly into working files. Certain special strings called version headers are replaced in each successive version by the number of that version.

If you are using RCS, and version headers are present in your working files, Emacs can use them to determine the current version and the locking state of the files. This is more reliable than referring to the master files, which is done when there are no version headers. Note that in a multi-branch environment, version headers are necessary to make VC behave correctly (see section Multi-User Branching).

Searching for version headers is controlled by the variable `vc-consult-headers`. If it is non-`nil`, Emacs searches for headers to determine the version number you are editing. Setting it to `nil` disables this feature.

You can use the C-x v h command (`vc-insert-headers`) to insert a suitable header string.

C-x v h

Insert headers in a file for use with your version-control system.

The default header string is `$Id$' for RCS and `%W%' for SCCS. You can specify other headers to insert by setting the variable `vc-header-alist`. Its value is a list of elements of the form `(program . string)` where program is `RCS` or `SCCS` and string is the string to use.

Instead of a single string, you can specify a list of strings; then each string in the list is inserted as a separate header on a line of its own.

It is often necessary to use "superfluous" backslashes when writing the strings that you put in this variable. This is to prevent the string in the constant from being interpreted as a header itself if the Emacs Lisp file containing it is maintained with version control.

Each header is inserted surrounded by tabs, inside comment delimiters, on a new line at the start of the buffer. Normally the ordinary comment start and comment end strings of the current mode are used, but for certain modes, there are special comment delimiters for this purpose; the variable `vc-comment-alist` specifies them. Each element of this list has the form `(mode starter ender)`.

The variable `vc-static-header-alist` specifies further strings to add based on the name of the buffer. Its value should be a list of elements of the form `(regexp . format)`. Whenever regexp matches the buffer name, format is inserted as part of the header. A header line is inserted for each element that matches the buffer name, and for each string specified by `vc-header-alist`. The header line is made by processing the string from `vc-header-alist` with the format taken from the element. The default value for `vc-static-header-alist` is as follows:

```
(("\\.c$" .
  "\n#ifndef lint\nstatic char vcid[] = \"\%s\";\n\
#endif /* lint */\n"))
```

It specifies insertion of text of this form:

```
#ifndef lint
static char vcid[] = "string";
#endif /* lint */
```

Note that the text above starts with a blank line.

If you use more than one version header in a file, put them close together in the file. The mechanism in `revert-buffer` that preserves markers may not handle markers positioned between two version headers.

Customizing VC

There are many ways of customizing VC. The variables that control its behavior fall into three categories, described in the following sections.

VC Workfile Handling

Emacs normally does not save backup files for source files that are maintained with version control. If you want to make backup files even for files that use version control, set the variable `vc-make-backup-files` to a non-`nil` value.

Normally the work file exists all the time, whether it is locked or not. If you set `vc-keep-workfiles` to `nil`, then checking in a new version with C-x C-q deletes the work file; but any attempt to visit the file with Emacs creates it again. (With CVS, work files are always kept.)

Editing a version-controlled file through a symbolic link can be dangerous. It bypasses the version control system--you can edit the file without checking it out, and fail to check your changes in. Also,

your changes might overwrite those of another user. To protect against this, VC checks each symbolic link that you visit, to see if it points to a file under version control.

The variable `vc-follow-symlinks` controls what to do when a symbolic link points to a version-controlled file. If it is `nil`, VC only displays a warning message. If it is `t`, VC automatically follows the link, and visits the real file instead, telling you about this in the echo area. If the value is `ask` (the default), VC asks you each time whether to follow the link.

VC Status Retrieval

When deducing the locked/unlocked state of a file, VC first looks for an RCS version header string in the file (see section Inserting Version Control Headers). If there is no header string (or if the backend system is SCCS), VC normally looks at the file permissions of the work file; this is fast. But there might be situations when the file permissions cannot be trusted. In this case the master file has to be consulted, which is rather expensive. Also the master file can only tell you *if* there's any lock on the file, but not whether your work file really contains that locked version.

You can tell VC not to use version headers to determine lock status by setting `vc-consult-headers` to `nil`. VC then always uses the file permissions (if it can trust them), or else checks the master file.

You can specify the criterion for whether to trust the file permissions by setting the variable `vc-mistrust-permissions`. Its value may be `t` (always mistrust the file permissions and check the master file), `nil` (always trust the file permissions), or a function of one argument which makes the decision. The argument is the directory name of the `` `RCS' ``, `` `CVS' `` or `` `SCCS' `` subdirectory. A non-`nil` value from the function says to mistrust the file permissions. If you find that the file permissions of work files are changed erroneously, set `vc-mistrust-permissions` to `t`. Then VC always checks the master file to determine the file's status.

VC Command Execution

If `vc-suppress-confirm` is non-`nil`, then C-x C-q and C-x v i can save the current buffer without asking, and C-x v u also operates without asking for confirmation. (This variable does not affect C-x v c; that operation is so drastic that it should always ask for confirmation.)

VC mode does much of its work by running the shell commands for RCS, CVS and SCCS. If `vc-command-messages` is non-`nil`, VC displays messages to indicate which shell commands it runs, and additional messages when the commands finish.

You can specify additional directories to search for version control programs by setting the variable `vc-path`. These directories are searched before the usual search path. But the proper files are usually found automatically.

File Directories

The file system groups files into directories. A directory listing is a list of all the files in a directory. Emacs provides commands to create and delete directories, and to make directory listings in brief format (file names only) and verbose format (sizes, dates, and authors included). There is also a directory

browser called Dired; see section Dired, the Directory Editor.

C-x C-d dir-or-pattern RET

Display a brief directory listing (`list-directory`).

C-u C-x C-d dir-or-pattern RET

Display a verbose directory listing.

M-x make-directory RET dirname RET

Create a new directory named dirname.

M-x delete-directory RET dirname RET

Delete the directory named dirname. It must be empty, or you get an error.

The command to display a directory listing is C-x C-d (`list-directory`). It reads using the minibuffer a file name which is either a directory to be listed or a wildcard-containing pattern for the files to be listed. For example,

```
C-x C-d /u2/emacs/etc RET
```

lists all the files in directory `` `/u2/emacs/etc' ``. Here is an example of specifying a file name pattern:

```
C-x C-d /u2/emacs/src/*.c RET
```

Normally, C-x C-d prints a brief directory listing containing just file names. A numeric argument (regardless of value) tells it to make a verbose listing including sizes, dates, and authors (like `ls -l').

The text of a directory listing is obtained by running `ls` in an inferior process. Two Emacs variables control the switches passed to `ls`: `list-directory-brief-switches` is a string giving the switches to use in brief listings (`"-CF"` by default), and `list-directory-verbose-switches` is a string giving the switches to use in a verbose listing (`"-l"` by default).

Comparing Files

The command M-x diff compares two files, displaying the differences in an Emacs buffer named `*Diff*'. It works by running the `diff` program, using options taken from the variable `diff-switches`, whose value should be a string.

The buffer `*Diff*' has Compilation mode as its major mode, so you can use C-x ` to visit successive changed locations in the two source files. You can also move to a particular hunk of changes and type RET or C-c C-c, or click Mouse-2 on it, to move to the corresponding source location. You can also use the other special commands of Compilation mode: SPC and DEL for scrolling, and M-p and M-n for cursor motion. See section Running Compilations under Emacs.

The command M-x diff-backup compares a specified file with its most recent backup. If you specify the name of a backup file, `diff-backup` compares it with the source file that it is a backup of.

The command M-x compare-windows compares the text in the current window with that in the next window. Comparison starts at point in each window, and each starting position is pushed on the mark

ring in its respective buffer. Then point moves forward in each window, a character at a time, until a mismatch between the two windows is reached. Then the command is finished. For more information about windows in Emacs, section Multiple Windows.

With a numeric argument, `compare-windows` ignores changes in whitespace. If the variable `compare-ignore-case` is non-`nil`, it ignores differences in case as well.

See also section Merging Files with Emerge, for convenient facilities for merging two similar files.

Miscellaneous File Operations

Emacs has commands for performing many other operations on files. All operate on one file; they do not accept wild card file names.

M-x view-file allows you to scan or read a file by sequential screenfuls. It reads a file name argument using the minibuffer. After reading the file into an Emacs buffer, `view-file` displays the beginning. You can then type SPC to scroll forward one windowful, or DEL to scroll backward. Various other commands are provided for moving around in the file, but none for changing it; type ? while viewing for a list of them. They are mostly the same as normal Emacs cursor motion commands. To exit from viewing, type q. The commands for viewing are defined by a special major mode called View mode.

A related command, M-x view-buffer, views a buffer already present in Emacs. See section Miscellaneous Buffer Operations.

M-x insert-file inserts a copy of the contents of the specified file into the current buffer at point, leaving point unchanged before the contents and the mark after them.

M-x write-region is the inverse of M-x insert-file; it copies the contents of the region into the specified file. M-x append-to-file adds the text of the region to the end of the specified file. See section Accumulating Text.

M-x delete-file deletes the specified file, like the `rm` command in the shell. If you are deleting many files in one directory, it may be more convenient to use Dired (see section Dired, the Directory Editor).

M-x rename-file reads two file names old and new using the minibuffer, then renames file old as new. If a file named new already exists, you must confirm with yes or renaming is not done; this is because renaming causes the old meaning of the name new to be lost. If old and new are on different file systems, the file old is copied and deleted.

The similar command M-x add-name-to-file is used to add an additional name to an existing file without removing its old name. The new name must belong on the same file system that the file is on.

M-x copy-file reads the file old and writes a new file named new with the same contents. Confirmation is required if a file named new already exists, because copying has the consequence of overwriting the old contents of the file new.

M-x make-symbolic-link reads two file names old and linkname, then creates a symbolic link named linkname and pointing at old. The effect is that future attempts to open file linkname will refer to

whatever file is named old at the time the opening is done, or will get an error if the name old is not in use at that time. This command does not expand the argument filename, so that it allows you to specify a relative name as the target of the link.

Confirmation is required when creating the link if linkname is in use. Note that not all systems support symbolic links.

Accessing Compressed Files

Emacs comes with a library that can automatically uncompress compressed files when you visit them, and automatically recompress them if you alter them and save them. To enable this feature, type the command M-x auto-compression-mode.

When automatic compression (which implies automatic uncompression as well) is enabled, Emacs recognizes compressed files by their file names. File names ending in `.gz' indicate a file compressed with `gzip`. Other endings indicate other compression programs.

Automatic uncompression and compression apply to all the operations in which Emacs uses the contents of a file. This includes visiting it, saving it, inserting its contents into a buffer, loading it, and byte compiling it.

Go to the previous, next section.

Go to the previous, next section.

Using Multiple Buffers

The text you are editing in Emacs resides in an object called a buffer. Each time you visit a file, a buffer is created to hold the file's text. Each time you invoke Dired, a buffer is created to hold the directory listing. If you send a message with C-x m, a buffer named `*mail*' is used to hold the text of the message. When you ask for a command's documentation, that appears in a buffer called `*Help*'.

At any time, one and only one buffer is selected. It is also called the current buffer. Often we say that a command operates on "the buffer" as if there were only one; but really this means that the command operates on the selected buffer (most commands do).

When Emacs has multiple windows, each window has a chosen buffer which is displayed there, but at any time only one of the windows is selected and its chosen buffer is the selected buffer. Each window's mode line displays the name of the buffer that the window is displaying (see section Multiple Windows).

Each buffer has a name, which can be of any length, and you can select any buffer by giving its name. Most buffers are made by visiting files, and their names are derived from the files' names. But you can also create an empty buffer with any name you want. A newly started Emacs has a buffer named `*scratch*' which can be used for evaluating Lisp expressions in Emacs. The distinction between upper and lower case matters in buffer names.

Each buffer records individually what file it is visiting, whether it is modified, and what major mode and minor modes are in effect in it (see section Major Modes). Any Emacs variable can be made local to a particular buffer, meaning its value in that buffer can be different from the value in other buffers. See section Local Variables.

Creating and Selecting Buffers

C-x b buffer RET

: Select or create a buffer named buffer (`switch-to-buffer`).

C-x 4 b buffer RET

: Similar, but select buffer in another window (`switch-to-buffer-other-window`).

C-x 5 b buffer RET

: Similar, but select buffer in a separate frame (`switch-to-buffer-other-frame`).

To select the buffer named bufname, type C-x b bufname RET. This runs the command `switch-to-buffer` with argument bufname. You can use completion on an abbreviation for the buffer name you want (see section Completion). An empty argument to C-x b specifies the most recently selected buffer that is not displayed in any window.

Most buffers are created by visiting files, or by Emacs commands that want to display some text, but you can also create a buffer explicitly by typing C-x b bufname RET. This makes a new, empty buffer which

is not visiting any file, and selects it for editing. Such buffers are used for making notes to yourself. I you try to save one, you are asked for the file name to use. The new buffer's major mode is determi by the value of `default-major-mode` (see section Major Modes).

Note that C-x C-f, and any other command for visiting a file, can also be used to switch to an existing file-visiting buffer. See section Visiting Files.

Emacs uses buffer names that start with a space for internal purposes. It treats these buffers specially in minor ways--for example, by default they do not record undo information. It is best to avoid using such buffer names yourself.

Listing Existing Buffers

C-x C-b

List the existing buffers (`list-buffers`).

To display a list of all the buffers that exist, type C-x C-b. Each line in the list shows one buffer's name, major mode and visited file. The buffers are listed in the order, most recently visited first.

`*' at the beginning of a line indicates the buffer is "modified". If several buffers are modified, it may be time to save some with C-x s (see section Saving Files). `%' indicates a read-only buffer. `.' marks the selected buffer. Here is an example of a buffer list:

```
 MR Buffer          Size  Mode            File
 -- ------          ----  ----            ----
.*  emacs.tex       383402 Texinfo        /u2/emacs/man/emacs.tex
    *Help*          1287  Fundamental
    files.el        23076 Emacs-Lisp      /u2/emacs/lisp/files.el
  % RMAIL           64042 RMAIL           /u/rms/RMAIL
 *% man             747   Dired           /u2/emacs/man/
    net.emacs       343885 Fundamental    /u/rms/net.emacs
    fileio.c        27691 C               /u2/emacs/src/fileio.c
    NEWS            67340 Text            /u2/emacs/etc/NEWS
    *scratch*       0     Lisp Interaction
```

Note that the buffer `*Help*' was made by a help request; it is not visiting any file. The buffer `man` was made by Dired on the directory `/u2/emacs/man/'.

Miscellaneous Buffer Operations

C-x C-q

Toggle read-only status of buffer (`vc-toggle-read-only`).

M-x rename-buffer RET name RET

Change the name of the current buffer.

M-x rename-uniquely

Rename the current buffer by adding `<number>' to the end.

M-x view-buffer RET buffer RET

Scroll through buffer buffer.

A buffer can be read-only, which means that commands to change its contents are not allowed. The mode line indicates read-only buffers with `%%' or `%*' near the left margin. Read-only buffers are usually made by subsystems such as Dired and Rmail that have special commands to operate on the text; also by visiting a file whose access control says you cannot write it.

If you wish to make changes in a read-only buffer, use the command C-x C-q (`vc-toggle-read-only`). It makes a read-only buffer writable, and makes a writable buffer read-only. In most cases, this works by setting the variable `buffer-read-only`, which has a local value in each buffer and makes the buffer read-only if its value is non-`nil`. If the file is maintained with version control, C-x C-q works through the version control system to change the read-only status of the file as well as the buffer. See section Version Control.

M-x rename-buffer changes the name of the current buffer. Specify the new name as a minibuffer argument. There is no default. If you specify a name that is in use for some other buffer, an error happens and no renaming is done.

M-x rename-uniquely renames the current buffer to a similar name with a numeric suffix added to make it both different and unique. This command does not need an argument. It is useful for creating multiple shell buffers: if you rename the `*Shell*' buffer, then do M-x shell again, it makes a new shell buffer named `*Shell*'; meanwhile, the old shell buffer continues to exist under its new name. This method is also good for mail buffers, compilation buffers, and most Emacs features that create special buffers with particular names.

M-x view-buffer is much like M-x view-file (see section Miscellaneous File Operations) except that it examines an already existing Emacs buffer. View mode provides commands for scrolling through the buffer conveniently but not for changing it. When you exit View mode, the value of point that resulted from your perusal remains in effect.

The commands M-x append-to-buffer and M-x insert-buffer can be used to copy text from one buffer to another. See section Accumulating Text.

Killing Buffers

If you continue an Emacs session for a while, you may accumulate a large number of buffers. You may then find it convenient to kill the buffers you no longer need. On most operating systems, killing a buffer releases its space back to the operating system so that other programs can use it. Here are some commands for killing buffers:

C-x k bufname RET

Kill buffer bufname (`kill-buffer`).

M-x kill-some-buffers

Offer to kill each buffer, one by one.

C-x k (`kill-buffer`) kills one buffer, whose name you specify in the minibuffer. The default, used if you type just RET in the minibuffer, is to kill the current buffer. If you kill the current buffer, another buffer is selected; one that has been selected recently but does not appear in any window now. If you ask to kill a file-visiting buffer that is modified (has unsaved editing), then you must confirm with yes before the buffer is killed.

The command M-x kill-some-buffers asks about each buffer, one by one. An answer of y means to kill the buffer. Killing the current buffer or a buffer containing unsaved changes selects a new buffer or asks for confirmation just like `kill-buffer`.

The buffer menu feature (see section Operating on Several Buffers) is also convenient for killing various buffers.

If you want to do something special every time a buffer is killed, you can add hook functions to the hook `kill-buffer-hook` (see section Hooks).

Operating on Several Buffers

The buffer-menu facility is like a "Dired for buffers"; it allows you to request operations on various Emacs buffers by editing an Emacs buffer containing a list of them. You can save buffers, kill them (here called deleting them, for consistency with Dired), or display them.

M-x buffer-menu

Begin editing a buffer listing all Emacs buffers.

The command `buffer-menu` writes a list of all Emacs buffers into the buffer `*Buffer List*', and selects that buffer in Buffer Menu mode. The buffer is read-only, and can be changed only through the special commands described in this section. The usual Emacs cursor motion commands can be used in the `*Buffer List*' buffer. The following commands apply to the buffer described on the current line.

d

Request to delete (kill) the buffer, then move down. The request shows as a `D' on the line, before the buffer name. Requested deletions take place when you type the x command.

C-d

Like d but move up afterwards instead of down.

s

Request to save the buffer. The request shows as an `S' on the line. Requested saves take place when you type the x command. You may request both saving and deletion for the same buffer.

x

Perform previously requested deletions and saves.

u

Remove any request made for the current line, and move down.

DEL

Move to previous line and remove any request made for that line.

The d, C-d, s and u commands to add or remove flags also move down (or up) one line. They accept a numeric argument as a repeat count.

These commands operate immediately on the buffer listed on the current line:

~
: Mark the buffer "unmodified". The command ~ does this immediately when you type it.

%
: Toggle the buffer's read-only flag. The command % does this immediately when you type it.

t
: Visit the buffer as a tags table. See section Selecting a Tags Table.

There are also commands to select another buffer or buffers:

q
: Quit the buffer menu--immediately display the most recent formerly visible buffer in its place.

RET

f
: Immediately select this line's buffer in place of the `*Buffer List*' buffer.

o
: Immediately select this line's buffer in another window as if by C-x 4 b, leaving `*Buffer List*' visible.

C-o
: Immediately display this line's buffer in another window, but don't select the window.

1
: Immediately select this line's buffer in a full-screen window.

2
: Immediately set up two windows, with this line's buffer in one, and the previously selected buffer (aside from the buffer `*Buffer List*') in the other.

m
: Mark this line's buffer to be displayed in another window if you exit with the v command. The request shows as a `>' at the beginning of the line. (A single buffer may not have both a delete request and a display request.)

v
: Immediately select this line's buffer, and also display in other windows any buffers previously marked with the m command. If you have not marked any buffers, this command is equivalent to 1.

All that `buffer-menu` does directly is create and switch to a suitable buffer, and turn on Buffer Menu mode. Everything else described above is implemented by the special commands provided in Buffer Menu mode. One consequence of this is that you can switch from the `*Buffer List*' buffer to another

Emacs buffer, and edit there. You can reselect the `*Buffer List*' buffer later, to perform the operati already requested, or you can kill it, or pay no further attention to it.

The only difference between `buffer-menu` and `list-buffers` is that `buffer-menu` switches to the `*Buffer List*' buffer in the selected window; `list-buffers` displays it in another window. If you run `list-buffers` (that is, type C-x C-b) and select the buffer list manually, you can use all of the commands described here.

The buffer `*Buffer List*' is not updated automatically when buffers are created and killed; its contents are just text. If you have created, deleted or renamed buffers, the way to update `*Buffer List*' to show what you have done is to type g (`revert-buffer`) or repeat the `buffer-menu` command.

Indirect Buffers

An indirect buffer shares the text of some other buffer, which is called the base buffer of the indirect buffer. In some ways it is the analogue, for buffers, of a symbolic link between files.

M-x make-indirect-buffer base-buffer RET indirect-name RET

> Create an indirect buffer named indirect-name whose base buffer is base-buffer.

The text of the indirect buffer is always identical to the text of its base buffer; changes made by editing either one are visible immediately in the other. But in all other respects, the indirect buffer and its base buffer are completely separate. They have different names, different values of point, different narrowing, different markers, different major modes, and different local variables.

An indirect buffer cannot visit a file, but its base buffer can. If you try to save the indirect buffer, that actually works by saving the base buffer. Killing the base buffer effectively kills the indirect buffer, but killing an indirect buffer has no effect on its base buffer.

One way to use indirect buffers is to display multiple views of an outline. See section Viewing One Outline in Multiple Views.

Go to the previous, next section.

Go to the previous, next section.

Multiple Windows

Emacs can split a frame into two or many windows. Multiple windows can display parts of different buffers, or different parts of one buffer. Multiple frames always imply multiple windows, because each frame has its own set of windows. Each window belongs to one and only one frame.

Concepts of Emacs Windows

Each Emacs window displays one Emacs buffer at any time. A single buffer may appear in more than one window; if it does, any changes in its text are displayed in all the windows where it appears. But the windows showing the same buffer can show different parts of it, because each window has its own value of point.

At any time, one of the windows is the selected window; the buffer this window is displaying is the current buffer. The terminal's cursor shows the location of point in this window. Each other window has a location of point as well, but since the terminal has only one cursor there is no way to show where those locations are. When multiple frames are visible in X Windows, each frame has a cursor which appears in the frame's selected window. The cursor in the selected frame is solid; the cursor in other frames is a hollow box.

Commands to move point affect the value of point for the selected Emacs window only. They do not change the value of point in any other Emacs window, even one showing the same buffer. The same is true for commands such as C-x b to change the selected buffer in the selected window; they do not affect other windows at all. However, there are other commands such as C-x 4 b that select a different window and switch buffers in it. Also, all commands that display information in a window, including (for example) C-h f (`describe-function`) and C-x C-b (`list-buffers`), work by switching buffers in a nonselected window without affecting the selected window.

When multiple windows show the same buffer, they can have different regions, because they can have different values of point. This means that in Transient Mark mode, each window highlights a different part of the buffer. The part that is highlighted in the selected window is the region that editing commands use.

Each window has its own mode line, which displays the buffer name, modification status and major and minor modes of the buffer that is displayed in the window. See section The Mode Line, for full details on the mode line.

@break

Splitting Windows

C-x 2

Split the selected window into two windows, one above the other (`split-window-vertically`).

C-x 3

Split the selected window into two windows positioned side by side (`split-window-horizontally`).

C-Mouse-2

In the mode line or scroll bar of a window, split that window.

The command C-x 2 (`split-window-vertically`) breaks the selected window into two windows, one above the other. Both windows start out displaying the same buffer, with the same value of point. By default the two windows each get half the height of the window that was split; a numeric argument specifies how many lines to give to the top window.

C-x 3 (`split-window-horizontally`) breaks the selected window into two side-by-side windows. A numeric argument specifies how many columns to give the one on the left. A line of vertical bars separates the two windows. Windows that are not the full width of the screen have mode lines, but they are truncated; also, they do not always appear in inverse video, because the Emacs display routines have not been taught how to display a region of inverse video that is only part of a line on the screen.

You can split a window horizontally or vertically by clicking C-Mouse-2 in the mode line or the scroll bar. The line of splitting goes through the place where you click: if you click on the mode line, the new scroll bar goes above the spot; if you click in the scroll bar, the mode line of the split window is side by side with your click.

When a window is less than the full width, text lines too long to fit are frequent. Continuing all those lines might be confusing. The variable `truncate-partial-width-windows` can be set non-`nil` to force truncation in all windows less than the full width of the screen, independent of the buffer being displayed and its value for `truncate-lines`. See section Continuation Lines.

Horizontal scrolling is often used in side-by-side windows. See section Controlling the Display.

If `split-window-keep-point` is non-nil, C-x 2 tries to avoid shifting any text on the screen by putting point in whichever window happens to contain the screen line the cursor is already on. The default is that `split-window-keep-point` is non-nil on slow terminals.

Using Other Windows

C-x o

Select another window (`other-window`). That is o, not zero.

C-M-v

Scroll the next window (`scroll-other-window`).

M-x compare-windows

Find next place where the text in the selected window does not match the text in the next window.

Mouse-1

Mouse-1, in a window's mode line, selects that window but does not move point in it (`mouse-select-region`).

To select a different window, click with Mouse-1 on its mode line. With the keyboard, you can switch windows by typing C-x o (`other-window`). That is an o, for `other', not a zero. When there are more than two windows, this command moves through all the windows in a cyclic order, generally top to bottom and left to right. After the rightmost and bottommost window, it goes back to the one at the upper left corner. A numeric argument means to move several steps in the cyclic order of windows. A negative argument moves around the cycle in the opposite order. When the minibuffer is active, the minibuffer is the last window in the cycle; you can switch from the minibuffer window to one of the other windows, and later switch back and finish supplying the minibuffer argument that is requested. See section Editing in the Minibuffer.

The usual scrolling commands (see section Controlling the Display) apply to the selected window only, but there is one command to scroll the next window. C-M-v (`scroll-other-window`) scrolls the window that C-x o would select. It takes arguments, positive and negative, like C-v. (In the minibuffer, C-M-v scrolls the window that contains the minibuffer help display, if any, rather than the next window in the standard cyclic order.)

The command `M-x compare-windows` lets you compare two files or buffers visible in two windows, by moving through them to the next mismatch. See section Comparing Files, for details.

Displaying in Another Window

C-x 4 is a prefix key for commands that select another window (splitting the window if there is only one) and select a buffer in that window. Different C-x 4 commands have different ways of finding the buffer to select.

C-x 4 b bufname RET

Select buffer bufname in another window. This runs `switch-to-buffer-other-window`.

C-x 4 C-o bufname RET

Display buffer bufname in another window, but don't select that buffer or that window. This runs `display-buffer`.

C-x 4 f filename RET

Visit file filename and select its buffer in another window. This runs `find-file-other-window`. See section Visiting Files.

C-x 4 d directory RET

Select a Dired buffer for directory directory in another window. This runs `dired-other-window`. See section Dired, the Directory Editor.

C-x 4 m

Start composing a mail message in another window. This runs `mail-other-window`; its same-window analogue is C-x m (see section Sending Mail).

C-x 4 .

Find a tag in the current tags table, in another window. This runs `find-tag-other-window`, the multiple-window variant of M-. (see section Tags Tables).

C-x 4 r filename RET

Visit file filename read-only, and select its buffer in another window. This runs `find-file-read-only-other-window`. See section Visiting Files.

Forcing Display in the Same Window

Certain Emacs commands switch to a specific buffer with special contents. For example, M-x shell switches to a buffer named `*Shell*'. By convention, all these commands are written to pop up the buffer in a separate window. But you can specify that certain of these buffers should appear in the selected window.

If you add a buffer name to the list `same-window-buffer-names`, the effect is that such commands display that particular buffer by switching to it in the selected window. For example, if you add the element `"*grep*"` to the list, the `grep` command will display its output buffer in the selected window.

The default value of `same-window-buffer-names` is not `nil`. It specifies the buffers `*info*', `*mail*' and `*shell*'. This is why M-x shell normally switches to the `*shell*' buffer in the selected window. If you delete this element from the value of `same-window-buffer-names`, the behavior of M-x shell will change--it will pop up the buffer in another window instead.

You can specify these buffers more generally with the variable `same-window-regexps`. Set it to a list of regular expressions; then any buffer whose name matches one of those regular expressions is displayed by switching to it in the selected window. (Once again, this applies only to buffers that normally get displayed for you in a separate window.) The default value of this variable specifies Telnet and rlogin buffers.

An analogous feature lets you specify buffers which should be displayed in their own individual frames. See section Special Buffer Frames.

Deleting and Rearranging Windows

C-x 0

Delete the selected window (`delete-window`). That is a zero.

C-x 1

Delete all windows in the selected frame except the selected window (`delete-other-windows`).

C-x ^

Make selected window taller (`enlarge-window`).

C-x }

Make selected window wider (`enlarge-window-horizontally`).

Drag-Mouse-1

Dragging a window's mode line up or down with Mouse-1 changes window heights.

Mouse-2

Mouse-2 in a window's mode line deletes all other windows in the frame (`mouse-delete-other-windows`).

Mouse-3

Mouse-3 in a window's mode line deletes that window (`mouse-delete-window`).

To delete a window, type C-x 0 (`delete-window`). (That is a zero.) The space occupied by the deleted window is given to an adjacent window (but not the minibuffer window, even if that is active at the time). Once a window is deleted, its attributes are forgotten; only restoring a window configuration can bring it back. Deleting the window has no effect on the buffer it used to display; the buffer continues to exist, and you can select it in any window with C-x b.

C-x 1 (`delete-other-windows`) is more powerful than C-x 0; it deletes all the windows except the selected one (and the minibuffer); the selected window expands to use the whole frame except for the echo area.

You can also delete a window by clicking on its mode line with Mouse-2, and expand a window to fill its frame by clicking on its mode line with Mouse-3.

The easiest way to adjust window heights is with a mouse. If you press Mouse-1 on a mode line, you can drag that mode line up or down, changing the heights of the windows above and below it.

To readjust the division of space among vertically adjacent windows, use C-x ^ (`enlarge-window`). It makes the currently selected window get one line bigger, or as many lines as is specified with a numeric argument. With a negative argument, it makes the selected window smaller. C-x } (`enlarge-window-horizontally`) makes the selected window wider by the specified number of columns. The extra screen space given to a window comes from one of its neighbors, if that is possible. If this makes any window too small, it is deleted and its space is given to an adjacent window. The minimum size is specified by the variables `window-min-height` and `window-min-width`.

See section Editing in the Minibuffer, for information about the Resize-Minibuffer mode, which automatically changes the size of the minibuffer window to fit the text in the minibuffer.

Go to the previous, next section.

Go to the previous, next section.

Frames and X Windows

When using the X Window System, you can create multiple windows at the X level in a single Emacs session. Each X window that belongs to Emacs displays a frame which can contain one or several Emacs windows. A frame initially contains a single general-purpose Emacs window which you can subdivide vertically or horizontally into smaller windows. A frame normally contains its own echo area and minibuffer, but you can make frames that don't have these--they use the echo area and minibuffer of another frame.

Editing you do in one frame also affects the other frames. For instance, if you put text in the kill ring in one frame, you can yank it in another frame. If you exit Emacs through C-x C-c in one frame, it terminates all the frames. To delete just one frame, use C-x 5 0.

To avoid confusion, we reserve the word "window" for the subdivisions that Emacs implements, and never use it to refer to a frame.

Mouse Commands for Editing

The mouse commands for selecting and copying a region are mostly compatible with the `xterm` program. You can use the same mouse commands for copying between Emacs and other X client programs.

Mouse-1

Move point to where you click (`mouse-set-point`). This is normally the left button.

Drag-Mouse-1

Set the region to the text you select by dragging, and copy it to the kill ring (`mouse-set-region`). You can specify both ends of the region with this single command.

If you move the mouse off the top or bottom of the window while dragging, the window scrolls at a steady rate until you move the mouse back into the window. This way, you can select regions that don't fit entirely on the screen. The number of lines scrolled per step depends on how far away from the window edge the mouse has gone; the variable `mouse-scroll-min-lines` specifies a minimum step size.

Mouse-2

Yank the last killed text, where you click (`mouse-yank-at-click`). This is normally the middle button.

Mouse-3

This command, `mouse-save-then-kill`, has several functions depending on where you click and the status of the region.

The most basic case is when you click Mouse-1 in one place and then Mouse-3 in another. This selects the text between those two positions as the region. It also copies the new region to the kill

ring, so that you can copy it to someplace else.

If you click Mouse-1, scroll with the scroll bar, and then click Mouse-3, it remembers where point was before scrolling (where you put it with Mouse-1), and uses that position as the other end of the region. This is so that you can select a region that doesn't fit entirely on the screen.

More generally, if you do not have a highlighted region, Mouse-3 selects the text between point and the click position as the region. It does this by setting the mark where point was, and moving point to where you click.

If you have a highlighted region, or if the region was set just before by dragging button 1, Mouse-3 adjusts the nearer end of the region by moving it to where you click. The adjusted region's text also replaces the old region's text in the kill ring.

If you originally specified the region using a double or triple Mouse-1, so that the region is defined to consist of entire words or lines, then adjusting the region with Mouse-3 also proceeds by entire words or lines.

If you use Mouse-3 a second time consecutively, at the same place, that kills the region already selected.

Double-Mouse-1

This key sets the region around the word which you click on. If you click on a character with "symbol" syntax (such as underscore, in C mode), it sets the region around the symbol surrounding that character.

If you click on a character with open-parenthesis or close-parenthesis syntax, it sets the region around the parenthetical grouping (sexp) which that character starts or ends. If you click on a character with string-delimiter syntax (such as a singlequote or doublequote in C), it sets the region around the string constant (using heuristics to figure out whether that character is the beginning or the end of it).

Double-Drag-Mouse-1

This key selects a region made up of the words that you drag across.

Triple-Mouse-1

This key sets the region around the line which you click on.

Triple-Drag-Mouse-1

This key selects a region made up of the lines that you drag across.

The simplest way to kill text with the mouse is to press Mouse-1 at one end, then press Mouse-3 twice at the other end. See section Deletion and Killing. To copy the text into the kill ring without deleting it from the buffer, press Mouse-3 just once--or just drag across the text with Mouse-1. Then you can copy it elsewhere by yanking it.

To yank the killed or copied text somewhere else, move the mouse there and press Mouse-2. See section Yanking. However, if `mouse-yank-at-point` is non-`nil`, Mouse-2 yanks at point. Then it does not matter precisely where you click; all that matters is which window you click on. The default value is `nil`. This variable also effects yanking the secondary selection.

To copy text to another X window, kill it or save it in the kill ring. Under X, this also sets the primary selection. Then use the "paste" or "yank" command of the program operating the other window to insert the text from the selection.

To copy text from another X window, use the "cut" or "copy" command of the program operating the other window, to select the text you want. Then yank it in Emacs with C-y or Mouse-2.

When Emacs puts text into the kill ring, or rotates text to the front of the kill ring, it sets the primary selection in the X server. This is how other X clients can access the text. Emacs also stores the text in the cut buffer, but only if the text is short enough (`x-cut-buffer-max` specifies the maximum number of characters); putting long strings in the cut buffer can be slow.

The commands to yank the first entry in the kill ring actually check first for a primary selection in another program; after that, they check for text in the cut buffer. If neither of those sources provides text to yank, the kill ring contents are used.

Secondary Selection

The secondary selection is another way of selecting text using X. It does not use point or the mark, so you can use it to kill text without setting point or the mark.

M-Drag-Mouse-1

> Set the secondary selection, with one end at the place where you press down the button, and the other end at the place where you release it (`mouse-set-secondary`). The highlighting appears and changes as you drag.
>
> If you move the mouse off the top or bottom of the window while dragging, the window scrolls at a steady rate until you move the mouse back into the window. This way, you can mark regions that don't fit entirely on the screen.

M-Mouse-1

> Set one endpoint for the secondary selection (`mouse-start-secondary`).

M-Mouse-3

> Make a secondary selection, using the place specified with M-Mouse-1 as the other end (`mouse-secondary-save-then-kill`). A second click at the same place kills the secondary selection just made.

M-Mouse-2

> Insert the secondary selection where you click (`mouse-kill-secondary`). This places point at the end of the yanked text.

Double or triple clicking of M-Mouse-1 operates on words and lines, much like Mouse-1.

If `mouse-yank-at-point` is non-`nil`, M-Mouse-2 yanks at point. Then it does not matter precisely where you click; all that matters is which window you click on. See section Mouse Commands for Editing.

Following References with the Mouse

Some Emacs buffers display lists of various sorts. These include lists of files, of buffers, of possible completions, of matches for a pattern, and so on.

Since yanking text into these buffers is not very useful, most of them define Mouse-2 specially, as a command to use or view the item you click on.

For example, if you click Mouse-2 on a file name in a Dired buffer, you visit the that file. If you click Mouse-2 on an error message in the `*Compilation*' buffer, you go to the source code for that error message. If you click Mouse-2 on a completion in the `*Completions*' buffer, you choose that completion.

You can usually tell when Mouse-2 has this special sort of meaning because the sensitive text highlights when you move the mouse over it.

Mouse Clicks for Menus

Mouse clicks modified with the CONTROL and SHIFT keys bring up menus.

C-Mouse-1

> This menu is for selecting a buffer.

C-Mouse-2

> This menu is for specifying faces and other text properties for editing formatted text. See section Editing Formatted Text.

C-Mouse-3

> This menu is mode-specific. For most modes, this menu has the same items as all the mode-specific menu bar menus put together. Some modes may specify a different menu for this button.(2)

S-mouse-1

> This menu is for specifying the frame's default font.

Mode Line Mouse Commands

You can use mouse clicks on window mode lines to select and manipulate windows.

Mouse-1

> Mouse-1 on a mode line selects the window above. By dragging Mouse-1 on the mode line, you can move it, thus changing the height of the windows above and below.

Mouse-2

> Mouse-2 on a mode line expands that window to fill its frame.

Mouse-3

> Mouse-3 on a mode line deletes the window above.

C-Mouse-2

C-Mouse-2 on a mode line splits the window above horizontally, above the place in the mode line where you click.

C-Mouse-2 on a scroll bar splits the corresponding window vertically. See section Splitting Windows.

Creating Frames

The prefix key C-x 5 is analogous to C-x 4, with parallel subcommands. The difference is that C-x 5 commands create a new frame rather than just a new window in the selected frame (See section Displaying in Another Window). If an existing visible or iconified frame already displays the requested material, these commands use the existing frame, after raising or deiconifying as necessary.

The various C-x 5 commands differ in how they find or create the buffer to select:

C-x 5 2

Create a new frame (`make-frame`).

C-x 5 b bufname RET

Select buffer bufname in another window. This runs `switch-to-buffer-other-frame`.

C-x 5 f filename RET

Visit file filename and select its buffer in another frame. This runs `find-file-other-frame`. See section Visiting Files.

C-x 5 d directory RET

Select a Dired buffer for directory directory in another frame. This runs `dired-other-frame`. See section Dired, the Directory Editor.

C-x 5 m

Start composing a mail message in another frame. This runs `mail-other-frame`. It is the other-frame variant of C-x m. See section Sending Mail.

C-x 5 .

Find a tag in the current tag table in another frame. This runs `find-tag-other-frame`, the multiple-frame variant of M-.. See section Tags Tables.

C-x 5 r filename RET

Visit file filename read-only, and select its buffer in another frame. This runs `find-file-read-only-other-frame`. See section Visiting Files.

You can control the appearance of new frames you create by setting the frame parameters in `default-frame-alist`. You can use the variable `initial-frame-alist` to specify parameters that affect only the initial frame. See section 'Initial Parameters' in The Emacs Lisp Manual, for more information.

Multiple Displays

A single Emacs can talk to more than one X Windows display. Initially, Emacs uses just one display--the one specified with the `DISPLAY` environment variable or with the `--display' option (see section Initial Options). To connect to another display, use the command `make-frame-on-display`:

M-x make-frame-on-display RET display RET

Create a new frame on display display.

A single X server can handle more than one screen. When you open frames on two screens belonging to one server, Emacs knows they share a single keyboard, and it treats all the commands arriving from these screens as a single stream of input.

When you open frames on different X servers, Emacs makes a separate input stream for each server. This way, two users can type simultaneously on the two displays, and Emacs will not garble their input. Each server also has its own selected frame. The commands you enter with a particular X server apply to that server's selected frame.

Despite these features, people using the same Emacs job from different displays can still interfere with each other if they are not careful. For example, if any one types C-x C-c, that exits the Emacs job for all of them!

Special Buffer Frames

You can make certain chosen buffers, for which Emacs normally creates a second window when you have just one window, appear in special frames of their own. To do this, set the variable `special-display-buffer-names` to a list of buffer names; any buffer whose name is in that list automatically gets a special frame, when an Emacs command wants to display it "in another window."

For example, if you set the variable this way,

```
(setq special-display-buffer-names
      '("*Completions*" "*grep*" "*tex-shell*"))
```

then completion lists, `grep` output and the TeX mode shell buffer get individual frames of their own. These frames, and the windows in them, are never automatically split or reused for any other buffers. They continue to show the buffers they were created for, unless you alter them by hand. Killing the special buffer deletes its frame automatically.

More generally, you can set `special-display-regexps` to a list of regular expressions; then a buffer gets its own frame if its name matches any of those regular expressions. (Once again, this applies only to buffers that normally get displayed for you in a separate window.)

The variable `special-display-frame-alist` specifies the frame parameters for these frames. It has a default value, so you don't need to set it.

For those who know Lisp, an element of `special-display-buffer-names` or

`special-display-regexps` can also be a list. Then the first element is the buffer name or regu expression; the rest of the list specifies how to create the frame. It can be an association list specifyi frame parameter values; these values take precedence over parameter values specified in `special-display-frame-alist`. Alternatively, it can have this form:

```
(function args...)
```

where function is a symbol. Then the frame is constructed by calling function; its first argument is the buffer, and its remaining arguments are args.

Setting Frame Parameters

This section describes commands for altering the display style and window management behavior of the selected frame.

M-x set-foreground-color RET color RET

Specify color color for the foreground of the selected frame.

M-x set-background-color RET color RET

Specify color color for the background of the selected frame. This changes the foreground color of the `modeline` face also, so that it remains in inverse video compared with the default.

M-x set-cursor-color RET color RET

Specify color color for the cursor of the selected frame.

M-x set-mouse-color RET color RET

Specify color color for the mouse cursor when it is over the selected frame.

M-x set-border-color RET color RET

Specify color color for the border of the selected frame.

M-x list-colors-display

Display the defined color names and show what the colors look like. This command is somewhat slow.

M-x auto-raise-mode

Toggle whether or not the selected frame should auto-raise. Auto-raise means that every time you move the mouse onto the frame, it raises the frame.

Note that this auto-raise feature is implemented by Emacs itself. Some window managers also implement auto-raise. If you enable auto-raise for Emacs frames in your X window manager, it should work, but it is beyond Emacs's control and therefore `auto-raise-mode` has no effect on it.

M-x auto-lower-mode

Toggle whether or not the selected frame should auto-lower. Auto-lower means that every time you move the mouse off of the frame, the frame moves to the bottom of the stack of X windows.

The command `auto-lower-mode` has no effect on auto-lower implemented by the X window manager. To control that, you must use the appropriate window manager features.

M-x set-default-font RET font RET

Specify font font as the default for the selected frame. See section Font Specification Options, for ways to list the available fonts on your system.

You can also set a frame's default font through a pop-up menu. Press S-Mouse-1 to activate this menu.

In Emacs versions that use an X toolkit, the color-setting and font-setting functions don't affect menus and the menu bar, since they are displayed by their own widget classes. To change the appearance of the menus and menu bar, you must use X resources (see section X Resources). See section Window Color Options, regarding colors. See section Font Specification Options, regarding choice of font.

For information on frame parameters and customization, see section 'Frame Parameters' in The Emacs Lisp Manual.

Scroll Bars

When using X, Emacs normally makes a scroll bar at the right of each Emacs window. The scroll bar runs the height of the window, and shows a moving rectangular inner box which represents the portion of the buffer currently displayed. The entire height of the scroll bar represents the entire length of the buffer.

You can use Mouse-2 (normally, the middle button) in the scroll bar to move or drag the inner box up and down. If you move it to the top of the scroll bar, you see the top of the buffer. If you move it to the bottom of the scroll bar, you see the bottom of the buffer.

The left and right buttons in the scroll bar scroll by controlled increments. Mouse-1 (normally, the left button) moves the line at the level where you click up to the top of the window. Mouse-3 (normally, the right button) moves the line at the top of the window down to the level where you click. By clicking repeatedly in the same place, you can scroll by the same distance over and over.

Aside from scrolling, you can also click C-Mouse-2 in the scroll bar to split a window vertically. The split occurs on the line where you click.

You can enable or disable Scroll Bar mode with the command M-x scroll-bar-mode. With no argument, it toggles the use of scroll bars. With an argument, it turns use of scroll bars on if and only if the argument is positive. This command applies to all frames, including frames yet to be created. You can use the X resource `verticalScrollBars' to control the initial setting of Scroll Bar mode. See section X Resources.

To enable or disable scroll bars for just the selected frame, use the M-x toggle-scroll-bar command.

Menu Bars

By default, each Emacs frame has a menu bar at the top which you can use to perform certain common operations. There's no need to describe them in detail here, as you can more easily see for yourself; also, we may change them and add to them in subsequent Emacs versions.

When you are using a window system, you can use the mouse to choose a command from the menu bar. On text-only terminals, you can use the menu bar by typing M-` (`tmm-menubar`). This enters a mode in which you can select a menu item from the keyboard. Either type the initial of the item you want, or use the left and right arrow keys to choose an item and use RET to finalize the choice.

Each of the operations in the menu bar is bound to an ordinary Emacs command which you can invoke equally well with M-x or with its own key bindings. The menu lists one equivalent key binding (if the command has any) at the right margin. To see the command's name and documentation, type C-h k and then select the menu bar item you are interested in.

You can turn display of menu bars on or off with M-x menu-bar-mode. With no argument, this command toggles Menu Bar mode, a minor mode. With an argument, the command turns Menu Bar mode on if the argument is positive, off if the argument is not positive. You can use the X resource `menuBarLines' to control the initial setting of Menu Bar mode. See section X Resources. Expert users often turn off the menu bar, especially on text-only terminals where this makes one additional line available for text.

Using Multiple Typefaces

When using Emacs with X, you can set up multiple styles of displaying characters. The aspects of style that you can control are the type font, the foreground color, the background color, and whether to underline. Emacs on MS-DOS supports faces partially by letting you control the foreground and background colors of each face (see section MS-DOS Issues).

The way you control display style is by defining named faces. Each face can specify a type font, a foreground color, a background color, and an underline flag; but it does not have to specify all of them.

The style of display used for a given character in the text is determined by combining several faces. Any aspect of the display style that isn't specified by overlays or text properties comes from the frame itself.

Enriched mode, the mode for editing formatted text, includes several commands and menus for specifying faces. See section Faces in Formatted Text, for how to specify the font for text in the buffer. See section Colors in Formatted Text, for how to specify the foreground and background color.

To see what faces are currently defined, and what they look like, type M-x list-faces-display. It's possible for a given face to look different in different frames; this command shows the appearance in the frame in which you type it. Here's a list of the standardly defined faces:

`default`

This face is used for ordinary text that doesn't specify any other face.

`modeline`

This face is used for mode lines. By default, it's set up as the inverse of the default face. See section Variables Controlling Display.

`highlight`
: This face is used for highlighting portions of text, in various modes.

`region`
: This face is used for displaying a selected region (when Transient Mark mode is enabled--see below).

`secondary-selection`
: This face is used for displaying a secondary selection (see section Secondary Selection).

`bold`
: This face uses a bold variant of the default font, if it has one.

`italic`
: This face uses an italic variant of the default font, if it has one.

`bold-italic`
: This face uses a bold italic variant of the default font, if it has one.

`underline`
: This face underlines text.

When Transient Mark mode is enabled, the text of the region is highlighted when the mark is active. This uses the face named `region`; you can control the style of highlighting by changing the style of this face (see section Modifying Faces). See section Transient Mark Mode, for more information about Transient Mark mode and activation and deactivation of the mark.

One easy way to use faces is to turn on Font Lock mode. This minor mode, which is always local to a particular buffer, arranges to choose faces according to the syntax of the text you are editing. It can recognize comments and strings in most languages; in several languages, it can also recognize and properly highlight various other important constructs. See section Font Lock mode, for more information about Font Lock mode and syntactic highlighting.

You can print out the buffer with the highlighting that appears on your screen using the command `ps-print-buffer-with-faces`. See section Postscript Hardcopy.

Modifying Faces

Here are the commands for changing the font of a face:

M-x set-face-font RET face RET font RET
: Change face face to use font font. See section Font Specification Options, for more information about font naming under X.

M-x make-face-bold RET face RET
: Convert face face to use a bold version of its current font.

M-x make-face-italic RET face RET

Convert face face to use a italic version of its current font.

M-x make-face-bold-italic RET face RET

Convert face face to use a bold-italic version of its current font.

M-x make-face-unbold RET face RET

Convert face face to use a non-bold version of its current font.

M-x make-face-unitalic RET face RET

Convert face face to use a non-italic version of its current font.

Here are the commands for setting the colors and underline flag of a face:

M-x set-face-foreground RET face RET color RET

Use color color for the foreground of characters in face face.

M-x set-face-background RET face RET color RET

Use color color for the background of characters in face face.

On a black-and-white display, the colors you can use for the background are `black', `white', `gray', `gray1' and `gray3'. Emacs supports the gray colors by using background stipple patterns instead of a color.

M-x set-face-stipple RET face RET pattern RET

Use stipple pattern pattern for the background of characters in face face.

M-x list-colors-display

Display the defined color names and show what the colors look like.

M-x set-face-underline-p RET face RET flag RET

Specify whether to underline characters in face face.

M-x invert-face RET face RET

Swap the foreground and background colors of face face.

M-x modify-face RET face RET attributes...

Change various attributes of face face. This command prompts for all the attribute of the face, one attribute at a time. For the color and stipple attributes, the attribute's current value is the default--type just RET if you don't want to change that attribute. Type `none' if you want to clear out the attribute.

You can also use X resources to specify attributes of particular faces. See section X Resources.

Font Lock mode

Font Lock mode is a minor mode, always local to a particular buffer, which highlights (or "fontifies") using various faces according to the syntax of the text you are editing. It can recognize comments and strings in most languages; in several languages, it can also recognize and properly highlight various other important constructs--for example, names of functions being defined or reserved keywords.

The command M-x font-lock-mode turns Font Lock mode on or off according to the argument, and toggles the mode when it has no argument. The function `turn-on-font-lock` unconditionally enables Font Lock mode. This is useful in mode-hook functions. For example, to enable Font Lock mode whenever you edit a C file, you can do this:

```
(add-hook 'c-mode-hook 'turn-on-font-lock)
```

To turn on Font Lock mode automatically in all modes which support it, use the function `global-font-lock-mode`, like this:

```
(global-font-lock-mode t)
```

In Font Lock mode, when you edit the text, the highlighting updates automatically in the line that you changed. Most changes don't affect the highlighting of subsequent lines, but occasionally they do. To rehighlight a range of lines, use the command C-M-g (`font-lock-fontify-block`).

In certain major modes, C-M-g refontifies the entire current function. (The variable `font-lock-mark-block-function` controls how to find the current function.) In other major modes, C-M-g refontifies 16 lines above and below point.

With a prefix argument n, C-M-g refontifies n lines above and below point, regardless of the mode.

To get the full benefit of Font Lock mode, you need to choose a default font which has bold, italic, and bold-italic variants; or else you need to have a color or grayscale screen. The variable `font-lock-display-type` specifies whether Font Lock mode should use font styles, colors, or shades of gray to distinguish the various kinds of text. Emacs chooses the default value according to the characteristics of your display.

The variable `font-lock-maximum-decoration` specifies the preferred level of fontification for modes that provide multiple levels. The normal default is 1; larger numbers request more fontification, and some modes support levels as high as 3. These variables can also specify different numbers for particular major modes; for example, to use level 3 for C/C++ modes, and the default level otherwise, use this:

```
(setq font-lock-maximum-decoration
      '((c-mode . 3) (c++-mode . 3)))
```

Fontification can be too slow for large buffers, so you can suppress it. The variable `font-lock-maximum-size` specifies a buffer size, beyond which buffer fontification is suppressed.

Font Lock Support Modes

Font Lock support modes make Font Lock mode faster for large buffers. There are two support modes: Fast Lock mode and Lazy Lock mode. They use two different methods of speeding up Font Lock mode.

Fast Lock Mode

To make Font Lock mode faster for buffers visiting large files, you can use Fast Lock mode. Fast Lock mode saves the font information for each file in a separate cache file; each time you visit the file, it rereads the font information from the cache file instead of refontifying the text from scratch.

The command M-x fast-lock-mode turns Fast Lock mode on or off, according to the argument (with no argument, it toggles). You can also arrange to enable Fast Lock mode whenever you use Font Lock mode, like this:

```
(setq font-lock-support-mode 'fast-lock-mode)
```

It is not worth writing a cache file for small buffers. Therefore, the variable `fast-lock-minimum-size` specifies a minimum file size for caching font information.

The variable `fast-lock-cache-directories` specifies where to put the cache files. Its value is a list of directories to try; `"."` means the same directory as the file being edited. The default value is `("." "~/.emacs-flc")`, which means to use the same directory if possible, and otherwise the directory `` `~/.emacs-flc' ``.

The variable `fast-lock-save-others` specifies whether Fast Lock mode should save cache files for files that you do not own. A non-`nil` value means yes (and that is the default).

Lazy Lock Mode

To make Font Lock mode faster for large buffers, you can use Lazy Lock mode to reduce the amount of text that is fontified. In Lazy Lock mode, buffer fontification is demand-driven; it happens to portions of the buffer that are about to be displayed. And fontification of your changes is deferred; it happens only when Emacs has been idle for a certain short period of time.

The command M-x lazy-lock-mode turns Lazy Lock mode on or off, according to the argument (with no argument, it toggles). You can also arrange to enable Lazy Lock mode whenever you use Font Lock mode, like this:

```
(setq font-lock-support-mode 'lazy-lock-mode)
```

It is not worth avoiding buffer fontification for small buffers. Therefore, the variable `lazy-lock-minimum-size` specifies a minimum buffer size for demand-driven buffer fontification. Buffers smaller than that are fontified all at once, as in plain Font Lock mode.

When you alter the buffer, Lazy Lock mode defers fontification of the text you changed. The variable `lazy-lock-defer-time` specifies how many seconds Emacs must be idle before it starts fontifying your changes. If the value is `nil`, then changes are fontified immediately, as in plain Font Lock mode.

Lazy Lock mode normally fontifies newly visible portions of the buffer before they are first displayed. However, if the value of `lazy-lock-defer-driven` is non-`nil`, newly visible text is fontified only when Emacs is idle for `lazy-lock-defer-time` seconds.

When Emacs is idle for a long time, Lazy Lock fontifies additional portions of the buffer, not yet displayed, in case you will display them later. This is called stealth fontification.

The variable `lazy-lock-stealth-time` specifies how many seconds Emacs has to be idle before stealth fontification starts. A value of `nil` means no stealth fontification. The variables `lazy-lock-stealth-lines` and `lazy-lock-stealth-verbose` specify the granularity and verbosity of stealth fontification.

Fast Lock or Lazy Lock?

Here is a simple guide to help you choose one of the Font Lock support modes.

- Fast Lock mode only intervenes during file visiting and buffer killing (and related events); therefore buffer editing and window scrolling are no faster or slower than plain Font Lock mode.
- Fast Lock mode is slower at reading a cache file than Lazy Lock mode is at fontifying a window; therefore Fast Lock mode is slower at visiting a file than Lazy Lock mode.
- Lazy Lock mode intervenes during window scrolling to fontify text that scrolls onto the screen; therefore, scrolling is slower than in plain Font Lock mode.
- Lazy Lock mode doesn't fontify during buffer editing (it defers fontification of changes); therefore, editing is faster than in plain Font Lock mode.
- Fast Lock mode can be fooled by a file that is kept under version control software; therefore buffer fontification may occur even when a cache file exists for the file.
- Fast Lock mode only works with a buffer visiting a file; Lazy Lock mode works with any buffer.
- Fast Lock mode generates cache files; Lazy Lock mode does not.

The variable `font-lock-support-mode` specifies which of these support modes to use; for example, to specify that Fast Lock mode is used for C/C++ modes, and Lazy Lock mode otherwise, set the variable like this:

```
(setq font-lock-support-mode
      '((c-mode . fast-lock-mode) (c++-mode . fast-lock-mode)
        (t . lazy-lock-mode)))
```

Miscellaneous X Window Features

The following commands let you create, delete and operate on frames:

C-z

To iconify the selected Emacs frame, type C-z (`iconify-or-deiconify-frame`). The normal meaning of C-z, to suspend Emacs, is not useful under a window system, so it has a different binding in that case.

If you type this command on an Emacs frame's icon, it deiconifies the frame.

C-x 5 0

To delete the selected frame, type C-x 5 0 (`delete-frame`). This is not allowed if there is only

one frame.

C-x 5 o

Select another frame, raise it, and warp the mouse to it so that it stays selected. If you repeat this command, it cycles through all the frames on your terminal.

M-x transient-mark-mode

Under X Windows, when Transient Mark mode is enabled, Emacs highlights the region when the mark is active. This feature is the main motive for using Transient Mark mode. To toggle the state of this mode, use the command M-x transient-mark-mode. See section The Mark and the Region.

Non-Window Terminals

If your terminal does not have a window system that Emacs supports, then it can display only one Emacs frame at a time. However, you can still create multiple Emacs frames, and switch between them. Switching frames on these terminals is much like switching between different window configurations.

Use C-x 5 2 to create a new frame and switch to it; use C-x 5 o to cycle through the existing frames; use C-x 5 0 to delete the current frame.

Each frame has a number to distinguish it. The selected frame's number appears in the mode line after `Emacs', except when frame 1 is selected.

Go to the previous, next section.

Go to the previous, next section.

Major Modes

Emacs provides many alternative major modes, each of which customizes Emacs for editing text of a particular sort. The major modes are mutually exclusive, and each buffer has one major mode at any time. The mode line normally shows the name of the current major mode, in parentheses (see section The Mode Line).

The least specialized major mode is called Fundamental mode. This mode has no mode-specific redefinitions or variable settings, so that each Emacs command behaves in its most general manner, and each option is in its default state. For editing text of a specific type that Emacs knows about, such as Lisp code or English text, you should switch to the appropriate major mode, such as Lisp mode or Text mode.

Selecting a major mode changes the meanings of a few keys to become more specifically adapted to the language being edited. The ones which are changed frequently are TAB, DEL, and LFD. The prefix key C-c normally contains mode-specific commands. In addition, the commands which handle comments use the mode to determine how comments are to be delimited. Many major modes redefine the syntactical properties of characters appearing in the buffer. See section The Syntax Table.

The major modes fall into three major groups. Lisp mode (which has several variants), C mode, Fortran mode and others are for specific programming languages. Text mode, Nroff mode, TeX mode and Outline mode are for editing English text. The remaining major modes are not intended for use on users' files; they are used in buffers created for specific purposes by Emacs, such as Dired mode for buffers made by Dired (see section Dired, the Directory Editor), and Mail mode for buffers made by C-x m (see section Sending Mail), and Shell mode for buffers used for communicating with an inferior shell process (see section Interactive Inferior Shell).

Most programming language major modes specify that only blank lines separate paragraphs. This is to make the paragraph commands useful. (See section Paragraphs.) They also cause Auto Fill mode to use the definition of TAB to indent the new lines it creates. This is because most lines in a program are usually indented. (See section Indentation.)

How Major Modes are Chosen

You can select a major mode explicitly for the current buffer, but most of the time Emacs determines which mode to use based on the file name or on special text in the file.

Explicit selection of a new major mode is done with a M-x command. From the name of a major mode, add `-mode` to get the name of a command to select that mode. Thus, you can enter Lisp mode by executing M-x lisp-mode.

When you visit a file, Emacs usually chooses the right major mode based on the file's name. For example, files whose names end in `.c' are edited in C mode. The correspondence between file names and

major modes is controlled by the variable `auto-mode-alist`. Its value is a list in which each ele has this form,

```
(regexp . mode-function)
```

or this form,

```
(regexp mode-function flag)
```

For example, one element normally found in the list has the form `("\\.c\\'" . c-mode)`, and it is responsible for selecting C mode for files whose names end in `.c'`. (Note that `\\' is needed in Lisp syntax to include a `\' in the string, which is needed to suppress the special meaning of `.' in regexps.) If the element has the form `(regexp mode-function flag)` and flag is non-nil, then after calling function, the suffix that matched regexp is deleted and the list is searched again for another match.

You can specify which major mode should be used for editing a certain file by a special sort of text in the first nonblank line of the file. The mode name should appear in this line both preceded and followed by `-*-'. Other text may appear on the line as well. For example,

```
;-*-Lisp-*-
```

tells Emacs to use Lisp mode. Such an explicit specification overrides any defaulting based on the file name. Note how the semicolon is used to make Lisp treat this line as a comment.

Another format of mode specification is

```
-*-Mode: modename;-*-
```

which allows you to specify local variables as well, like this:

```
-*- mode: modename; var: value; ... -*-
```

See section Local Variables in Files, for more information about this.

When a file's contents begin with `#!', it can serve as an executable shell command, which works by running an interpreter named on the file's first line. The rest of the file is used as input to the interpreter.

When you visit such a file in Emacs, if the file's name does not specify a major mode, Emacs uses the interpreter name on the first line to choose a mode. If the first line is the name of a recognized interpreter program, such as `perl' or `tcl', Emacs uses a mode appropriate for programs for that interpreter. The variable `interpreter-mode-alist` specifies the correspondence between interpreter program names and major modes.

When you visit a file that does not specify a major mode to use, or when you create a new buffer with C-x b, the variable `default-major-mode` specifies which major mode to use. Normally its value is the symbol `fundamental-mode`, which specifies Fundamental mode. If `default-major-mode` is `nil`, the major mode is taken from the previously selected buffer.

TOC

If you change the major mode of a buffer, you can go back to the major mode Emacs would choose automatically: use the command M-x normal-mode to do this. This is the same function that `find-file` calls to choose the major mode. It also processes the file's local variables list if any.

Go to the previous, next section.

Go to the previous, next section.

TOC

Indentation

This chapter describes the Emacs commands that add, remove, or adjust indentation.

TAB

Indent current line "appropriately" in a mode-dependent fashion.

LFD

Perform RET followed by TAB (`newline-and-indent`).

M-^

Merge two lines (`delete-indentation`). This would cancel out the effect of LFD.

C-M-o

Split line at point; text on the line after point becomes a new line indented to the same column that it now starts in (`split-line`).

M-m

Move (forward or back) to the first nonblank character on the current line (`back-to-indentation`).

C-M-\

Indent several lines to same column (`indent-region`).

C-x TAB

Shift block of lines rigidly right or left (`indent-rigidly`).

M-i

Indent from point to the next prespecified tab stop column (`tab-to-tab-stop`).

M-x indent-relative

Indent from point to under an indentation point in the previous line.

Most programming languages have some indentation convention. For Lisp code, lines are indented according to their nesting in parentheses. The same general idea is used for C code, though many details are different.

Whatever the language, to indent a line, use the TAB command. Each major mode defines this command to perform the sort of indentation appropriate for the particular language. In Lisp mode, TAB aligns the line according to its depth in parentheses. No matter where in the line you are when you type TAB, it aligns the line as a whole. In C mode, TAB implements a subtle and sophisticated indentation style that knows about many aspects of C syntax.

In Text mode, TAB runs the command `tab-to-tab-stop`, which indents to the next tab stop column. You can set the tab stops with M-x edit-tab-stops.

Indentation Commands and Techniques

To move over the indentation on a line, do M-m (`back-to-indentation`). This command, given anywhere on a line, positions point at the first nonblank character on the line.

To insert an indented line before the current line, do C-a C-o TAB. To make an indented line after the current line, use C-e LFD.

If you just want to insert a tab character in the buffer, you can type C-q TAB.

C-M-o (`split-line`) moves the text from point to the end of the line vertically down, so that the current line becomes two lines. C-M-o first moves point forward over any spaces and tabs. Then it inserts after point a newline and enough indentation to reach the same column point is on. Point remains before the inserted newline; in this regard, C-M-o resembles C-o.

To join two lines cleanly, use the M-^ (`delete-indentation`) command. It deletes the indentation at the front of the current line, and the line boundary as well, replacing them with a single space. As a special case (useful for Lisp code) the single space is omitted if the characters to be joined are consecutive open parentheses or closing parentheses, or if the junction follows another newline. To delete just the indentation of a line, go to the beginning of the line and use M-\ (`delete-horizontal-space`), which deletes all spaces and tabs around the cursor.

If you have a fill prefix, M-^ deletes the fill prefix if it appears after the newline that is deleted. See section The Fill Prefix.

There are also commands for changing the indentation of several lines at once. C-M-\ (`indent-region`) applies to all the lines that begin in the region; it indents each line in the "usual" way, as if you had typed TAB at the beginning of the line. A numeric argument specifies the column to indent to, and each line is shifted left or right so that its first nonblank character appears in that column. C-x TAB (`indent-rigidly`) moves all of the lines in the region right by its argument (left, for negative arguments). The whole group of lines moves rigidly sideways, which is how the command gets its name.

M-x indent-relative indents at point based on the previous line (actually, the last nonempty line). It inserts whitespace at point, moving point, until it is underneath an indentation point in the previous line. An indentation point is the end of a sequence of whitespace or the end of the line. If point is farther right than any indentation point in the previous line, the whitespace before point is deleted and the first indentation point then applicable is used. If no indentation point is applicable even then, `indent-relative` runs `tab-to-tab-stop` (see next section).

`indent-relative` is the definition of TAB in Indented Text mode. See section Commands for Human Languages.

See section Indentation in Formatted Text, for another way of specifying the indentation for part of your text.

Tab Stops

For typing in tables, you can use Text mode's definition of TAB, `tab-to-tab-stop`. This command inserts indentation before point, enough to reach the next tab stop column. If you are not in Text mode, this command can be found on the key M-i.

You can specify the tab stops used by M-i. They are stored in a variable called `tab-stop-list`, as a list of column-numbers in increasing order.

The convenient way to set the tab stops is with M-x edit-tab-stops, which creates and selects a buffer containing a description of the tab stop settings. You can edit this buffer to specify different tab stops, and then type C-c C-c to make those new tab stops take effect. In the tab stop buffer, C-c C-c runs the function `edit-tab-stops-note-changes` rather than its usual definition `save-buffer`. `edit-tab-stops` records which buffer was current when you invoked it, and stores the tab stops back in that buffer; normally all buffers share the same tab stops and changing them in one buffer affects all, but if you happen to make `tab-stop-list` local in one buffer then `edit-tab-stops` in that buffer will edit the local settings.

Here is what the text representing the tab stops looks like for ordinary tab stops every eight columns.

```
        :       :       :       :       :       :
0         1         2         3         4
0123456789012345678901234567890123456789012345678
To install changes, type C-c C-c
```

The first line contains a colon at each tab stop. The remaining lines are present just to help you see where the colons are and know what to do.

Note that the tab stops that control `tab-to-tab-stop` have nothing to do with displaying tab characters in the buffer. See section Variables Controlling Display, for more information on that.

Tabs vs. Spaces

Emacs normally uses both tabs and spaces to indent lines. If you prefer, all indentation can be made from spaces only. To request this, set `indent-tabs-mode` to `nil`. This is a per-buffer variable; altering the variable affects only the current buffer, but there is a default value which you can change as well. See section Local Variables.

There are also commands to convert tabs to spaces or vice versa, always preserving the columns of all nonblank text. M-x tabify scans the region for sequences of spaces, and converts sequences of at least three spaces to tabs if that can be done without changing indentation. M-x untabify changes all tabs in the region to appropriate numbers of spaces.

Go to the previous, next section.

Go to the previous, next section.

Commands for Human Languages

The term text has two widespread meanings in our area of the computer field. One is data that is a sequence of characters. Any file that you edit with Emacs is text, in this sense of the word. The other meaning is more restrictive: a sequence of characters in a human language for humans to read (possibly after processing by a text formatter), as opposed to a program or commands for a program.

Human languages have syntactic/stylistic conventions that can be supported or used to advantage by editor commands: conventions involving words, sentences, paragraphs, and capital letters. This chapter describes Emacs commands for all of these things. There are also commands for filling, which means rearranging the lines of a paragraph to be approximately equal in length. The commands for moving over and killing words, sentences and paragraphs, while intended primarily for editing text, are also often useful for editing programs.

Emacs has several major modes for editing human language text. If the file contains text pure and simple, use Text mode, which customizes Emacs in small ways for the syntactic conventions of text. Outline mode provides special commands for operating on text with an outline structure. See section Outline Mode.

For text which contains embedded commands for text formatters, Emacs has other major modes, each for a particular text formatter. Thus, for input to TeX, you would use TeX mode (see section TeX Mode). For input to nroff, use Nroff mode.

Instead of using a text formatter, you can edit formatted text in WYSIWYG style ("what you see is what you get"), with Enriched mode. Then the formatting appears on the screen in Emacs while you edit. See section Editing Formatted Text.

Words

Emacs has commands for moving over or operating on words. By convention, the keys for them are all Meta characters.

M-f
: Move forward over a word (`forward-word`).

M-b
: Move backward over a word (`backward-word`).

M-d
: Kill up to the end of a word (`kill-word`).

M-DEL
: Kill back to the beginning of a word (`backward-kill-word`).

M-@

Mark the end of the next word (`mark-word`).

M-t

Transpose two words or drag a word across other words (`transpose-words`).

Notice how these keys form a series that parallels the character-based C-f, C-b, C-d, C-t and DEL. M-@ is cognate to C-@, which is an alias for C-SPC.

The commands M-f (`forward-word`) and M-b (`backward-word`) move forward and backward over words. These Meta characters are thus analogous to the corresponding control characters, C-f and C-b, which move over single characters in the text. The analogy extends to numeric arguments, which serve as repeat counts. M-f with a negative argument moves backward, and M-b with a negative argument moves forward. Forward motion stops right after the last letter of the word, while backward motion stops right before the first letter.

M-d (`kill-word`) kills the word after point. To be precise, it kills everything from point to the place M-f would move to. Thus, if point is in the middle of a word, M-d kills just the part after point. If some punctuation comes between point and the next word, it is killed along with the word. (If you wish to kill only the next word but not the punctuation before it, simply do M-f to get the end, and kill the word backwards with M-DEL.) M-d takes arguments just like M-f.

M-DEL (`backward-kill-word`) kills the word before point. It kills everything from point back to where M-b would move to. If point is after the space in `FOO, BAR', then `FOO, ' is killed. (If you wish to kill just `FOO', do M-b M-d instead of M-DEL.)

M-t (`transpose-words`) exchanges the word before or containing point with the following word. The delimiter characters between the words do not move. For example, `FOO, BAR' transposes into `BAR, FOO' rather than `BAR FOO,'. See section Transposing Text, for more on transposition and on arguments to transposition commands.

To operate on the next n words with an operation which applies between point and mark, you can either set the mark at point and then move over the words, or you can use the command M-@ (`mark-word`) which does not move point, but sets the mark where M-f would move to. M-@ accepts a numeric argument that says how many words to scan for the place to put the mark. In Transient Mark mode, this command activates the mark.

The word commands' understanding of syntax is completely controlled by the syntax table. Any character can, for example, be declared to be a word delimiter. See section The Syntax Table.

Sentences

The Emacs commands for manipulating sentences and paragraphs are mostly on Meta keys, so as to be like the word-handling commands.

M-a

Move back to the beginning of the sentence (`backward-sentence`).

M-e

Move forward to the end of the sentence (`forward-sentence`).

M-k

Kill forward to the end of the sentence (`kill-sentence`).

C-x DEL

Kill back to the beginning of the sentence (`backward-kill-sentence`).

The commands M-a and M-e (`backward-sentence` and `forward-sentence`) move to the beginning and end of the current sentence, respectively. They were chosen to resemble C-a and C-e, which move to the beginning and end of a line. Unlike them, M-a and M-e if repeated or given numeric arguments move over successive sentences.

Moving backward over a sentence places point just before the first character of the sentence; moving forward places point right after the punctuation that ends the sentence. Neither one moves over the whitespace at the sentence boundary.

Just as C-a and C-e have a kill command, C-k, to go with them, so M-a and M-e have a corresponding kill command M-k (`kill-sentence`) which kills from point to the end of the sentence. With minus one as an argument it kills back to the beginning of the sentence. Larger arguments serve as a repeat count. There is also a command, C-x DEL (`backward-kill-sentence`), for killing back to the beginning of a sentence. This command is useful when you change your mind in the middle of composing text.

The sentence commands assume that you follow the American typist's convention of putting two spaces at the end of a sentence; they consider a sentence to end wherever there is a `.', `?' or `!' followed by the end of a line or two spaces, with any number of `)', `]', `', or `"' characters allowed in between. A sentence also begins or ends wherever a paragraph begins or ends.

The variable `sentence-end` controls recognition of the end of a sentence. It is a regexp that matches the last few characters of a sentence, together with the whitespace following the sentence. Its normal value is

```
"[.?!][]\"')]*\\($\\|\t\\|  \\)[ \t\n]*"
```

This example is explained in the section on regexps. See section Syntax of Regular Expressions.

If you want to use just one space between sentences, you should set `sentence-end` to this value:

```
"[.?!][]\"')]*\\($\\|\t\\| \\)[ \t\n]*"
```

You should also set the variable `sentence-end-double-space` to `nil` so that the fill commands expect and leave just one space at the end of a sentence. Note that this makes it impossible to distinguish between periods that end sentences and those that indicate abbreviations.

Paragraphs

The Emacs commands for manipulating paragraphs are also Meta keys.

M-{
: Move back to previous paragraph beginning (`backward-paragraph`).

M-}
: Move forward to next paragraph end (`forward-paragraph`).

M-h
: Put point and mark around this or next paragraph (`mark-paragraph`).

M-{ moves to the beginning of the current or previous paragraph, while M-} moves to the end of the current or next paragraph. Blank lines and text formatter command lines separate paragraphs and are not considered part of any paragraph. Also, an indented line starts a new paragraph.

In major modes for programs (as opposed to Text mode), paragraphs begin and end only at blank lines. This makes the paragraph commands continue to be useful even though there are no paragraphs per se.

When there is a fill prefix, then paragraphs are delimited by all lines which don't start with the fill prefix. See section Filling Text.

When you wish to operate on a paragraph, you can use the command M-h (`mark-paragraph`) to set the region around it. Thus, for example, M-h C-w kills the paragraph around or after point. The M-h command puts point at the beginning and mark at the end of the paragraph point was in. In Transient Mark mode, it activates the mark. If point is between paragraphs (in a run of blank lines, or at a boundary), the paragraph following point is surrounded by point and mark. If there are blank lines preceding the first line of the paragraph, one of these blank lines is included in the region.

The precise definition of a paragraph boundary is controlled by the variables `paragraph-separate` and `paragraph-start`. The value of `paragraph-start` is a regexp that should match any line that either starts or separates paragraphs. The value of `paragraph-separate` is another regexp that should match only lines that separate paragraphs without being part of any paragraph. Lines that start a new paragraph and are contained in it must match only `paragraph-start`, not `paragraph-separate`. For example, normally `paragraph-start` is `"[ \t\n\f]"` and `paragraph-separate` is `"[ \t\f]*$"`.

Normally it is desirable for page boundaries to separate paragraphs. The default values of these variables recognize the usual separator for pages.

Pages

Files are often thought of as divided into pages by the formfeed character (ASCII control-L, octal code 014). When you print hardcopy for a file, this character forces a page break; thus, each page of the file goes on a separate page on paper. Most Emacs commands treat the page-separator character just like any other character: you can insert it with C-q C-l, and delete it with DEL. Thus, you are free to paginate your file or not. However, since pages are often meaningful divisions of the file, Emacs provides

commands to move over them and operate on them.

C-x [
: Move point to previous page boundary (`backward-page`).

C-x]
: Move point to next page boundary (`forward-page`).

C-x C-p
: Put point and mark around this page (or another page) (`mark-page`).

C-x l
: Count the lines in this page (`count-lines-page`).

The C-x [(`backward-page`) command moves point to immediately after the previous page delimiter. If point is already right after a page delimiter, it skips that one and stops at the previous one. A numeric argument serves as a repeat count. The C-x] (`forward-page`) command moves forward past the next page delimiter.

The C-x C-p command (`mark-page`) puts point at the beginning of the current page and the mark at the end. The page delimiter at the end is included (the mark follows it). The page delimiter at the front is excluded (point follows it). C-x C-p C-w is a handy way to kill a page to move it elsewhere. If you move to another page delimiter with C-x [and C-x], then yank the killed page, all the pages will be properly delimited once again. The reason C-x C-p includes only the following page delimiter in the region is to ensure that.

A numeric argument to C-x C-p is used to specify which page to go to, relative to the current one. Zero means the current page. One means the next page, and -1 means the previous one.

The C-x l command (`count-lines-page`) is good for deciding where to break a page in two. It prints in the echo area the total number of lines in the current page, and then divides it up into those preceding the current line and those following, as in

```
Page has 96 (72+25) lines
```

Notice that the sum is off by one; this is correct if point is not at the beginning of a line.

The variable `page-delimiter` controls where pages begin. Its value is a regexp that matches the beginning of a line that separates pages. The normal value of this variable is `"^\f"`, which matches a formfeed character at the beginning of a line.

Filling Text

Filling text means breaking it up into lines that fit a specified width. Emacs does filling in two ways. In Auto Fill mode, inserting text with self-inserting characters also automatically fills it. There are also explicit fill commands that you can use when editing text leaves it unfilled. When you edit formatted text, you can specify a style of filling for each portion of the text (see section Editing Formatted Text).

Auto Fill Mode

Auto Fill mode is a minor mode in which lines are broken automatically when they become too wide. Breaking happens only when you type a SPC or RET.

M-x auto-fill-mode

Enable or disable Auto Fill mode.

SPC

RET

In Auto Fill mode, break lines when appropriate.

M-x auto-fill-mode turns Auto Fill mode on if it was off, or off if it was on. With a positive numeric argument it always turns Auto Fill mode on, and with a negative argument always turns it off. You can see when Auto Fill mode is in effect by the presence of the word `Fill' in the mode line, inside the parentheses. Auto Fill mode is a minor mode which is enabled or disabled for each buffer individually. See section Minor Modes.

In Auto Fill mode, lines are broken automatically at spaces when they get longer than the desired width. Line breaking and rearrangement takes place only when you type SPC or RET. If you wish to insert a space or newline without permitting line-breaking, type C-q SPC or C-q LFD (recall that a newline is really a linefeed). Also, C-o inserts a newline without line breaking.

Auto Fill mode works well with Lisp mode, because when it makes a new line in Lisp mode it indents that line with TAB. If a line ending in a comment gets too long, the text of the comment is split into two comment lines. Optionally new comment delimiters are inserted at the end of the first line and the beginning of the second so that each line is a separate comment; the variable `comment-multi-line` controls the choice (see section Manipulating Comments).

Adaptive filling (see the following section) works for Auto Filling as well as for explicit fill commands. It takes a fill prefix automatically from the second or first line of a paragraph.

Auto Fill mode does not refill entire paragraphs; it can break lines but cannot merge lines. So editing in the middle of a paragraph can result in a paragraph that is not correctly filled. The easiest way to make the paragraph properly filled again is usually with the explicit fill commands.

Many users like Auto Fill mode and want to use it in all text files. The section on init files says how to arrange this permanently for yourself. See section The Init File, `~/.emacs'.

Explicit Fill Commands

M-q

Fill current paragraph (`fill-paragraph`).

C-x f

Set the fill column (`set-fill-column`).

M-x fill-region

Fill each paragraph in the region (`fill-region`).

M-x fill-region-as-paragraph.

Fill the region, considering it as one paragraph.

M-s

Center a line.

To refill a paragraph, use the command M-q (`fill-paragraph`). This operates on the paragraph that point is inside, or the one after point if point is between paragraphs. Refilling works by removing all the line-breaks, then inserting new ones where necessary.

To refill many paragraphs, use M-x fill-region, which divides the region into paragraphs and fills each of them.

M-q and `fill-region` use the same criteria as M-h for finding paragraph boundaries (see section Paragraphs). For more control, you can use M-x fill-region-as-paragraph, which refills everything between point and mark. This command deletes any blank lines within the region, so separate blocks of text end up combined into one block.

A numeric argument to M-q causes it to justify the text as well as filling it. This means that extra spaces are inserted to make the right margin line up exactly at the fill column. To remove the extra spaces, use M-q with no argument. (Likewise for `fill-region`.) Another way to control justification, and choose other styles of filling, is with the `justification` text property; see section Justification in Formatted Text.

The fill commands can deduce the proper fill prefix for a paragraph automatically in certain cases: either whitespace or certain punctuation characters at the beginning of a line are treated as a fill prefix. They take the fill prefix from the paragraph's second line, unless the paragraph has just one line. You can turn off this feature by setting `adaptive-fill-mode` to `nil`.

Some major modes, including Text mode, treat whitespace at the beginning of a line as a signal that this line starts a new paragraph. It would be a mistake to copy text which implies the start of a paragraph onto each line of the paragraph when filling it. Therefore, adaptive filling does not accept a fill prefix from a line which is a paragraph-starter. In particular, adaptive filling in Text mode does not accept a fill prefix consisting of just whitespace.

However, other modes including Indented Text mode (see section Text Mode) do not consider whitespace as a signal to start a new paragraph. In these modes, adaptive filling does accept a fill prefix consisting of just whitespace, if the first or second line of a paragraph begins with whitespace.

The variable `adaptive-fill-regexp` determines what kinds of line beginnings can serve as a fill prefix: any characters at the start of the line which match this regular expression are used.

You can specify more complex ways of choosing a fill prefix automatically by setting the variable `adaptive-fill-function` to a function. This function is called with point after the left margin of a line, and it should return the appropriate fill prefix based on that line. If it returns `nil`, that means it sees no fill prefix in that line.

The command M-s (`center-line`) centers the current line within the current fill column. With an

argument n, it centers n lines individually and moves past them.

The maximum line width for filling is in the variable `fill-column`. Altering the value of `fill-column` makes it local to the current buffer; until that time, the default value is in effect. The default is initially 70. See section Local Variables. The easiest way to set `fill-column` is to use the command C-x f (`set-fill-column`). With a numeric argument, it uses that as the new fill column. With just C-u as argument, it sets `fill-column` to the current horizontal position of point.

Emacs commands normally consider a period followed by two spaces or by a newline as the end of a sentence; a period followed by just one space indicates an abbreviation and not the end of a sentence. To preserve the distinction between these two ways of using a period, the fill commands do not break a line after a period followed by just one space.

If the variable `sentence-end-double-space` is `nil`, the fill commands expect and leave just one space at the end of a sentence. Ordinarily this variable is `t`, so the fill commands insist on two spaces for the end of a sentence, as explained above. See section Sentences.

If the variable `colon-double-space` is non-`nil`, the fill commands put two spaces after a colon.

The Fill Prefix

To fill a paragraph in which each line starts with a special marker (which might be a few spaces, giving an indented paragraph), use the fill prefix feature. The fill prefix is a string which Emacs expects every line to start with, and which is not included in filling.

C-x .
: Set the fill prefix (`set-fill-prefix`).

M-q
: Fill a paragraph using current fill prefix (`fill-paragraph`).

M-x fill-individual-paragraphs
: Fill the region, considering each change of indentation as starting a new paragraph.

M-x fill-nonuniform-paragraphs
: Fill the region, considering only paragraph-separator lines as starting a new paragraph.

To specify a fill prefix, move to a line that starts with the desired prefix, put point at the end of the prefix, and give the command C-x . (`set-fill-prefix`). That's a period after the C-x. To turn off the fill prefix, specify an empty prefix: type C-x . with point at the beginning of a line.

When a fill prefix is in effect, the fill commands remove the fill prefix from each line before filling and insert it on each line after filling. Auto Fill mode also inserts the fill prefix automatically when it makes a new line. The C-o command inserts the fill prefix on new lines it creates, when you use it at the beginning of a line (see section Blank Lines). Conversely, the command M-^ deletes the prefix (if it occurs) after the newline that it deletes (see section Indentation).

For example, if `fill-column` is 40 and you set the fill prefix to `;; ', then M-q in the following text


```
;; This is an
;; example of a paragraph
;; inside a Lisp-style comment.
```

produces this:

```
;; This is an example of a paragraph
;; inside a Lisp-style comment.
```

Lines that do not start with the fill prefix are considered to start paragraphs, both in M-q and the paragraph commands; this gives good results for paragraphs with hanging indentation (every line indented except the first one). Lines which are blank or indented once the prefix is removed also separate or start paragraphs; this is what you want if you are writing multi-paragraph comments with a comment delimiter on each line.

You can use M-x fill-individual-paragraphs to set the fill prefix for each paragraph automatically. This command divides the region into paragraphs, treating every change in the amount of indentation as the start of a new paragraph, and fills each of these paragraphs. Thus, all the lines in one "paragraph" have the same amount of indentation. That indentation serves as the fill prefix for that paragraph.

M-x fill-nonuniform-paragraphs is a similar command that divides the region into paragraphs in a different way. It considers only paragraph-separating lines (as defined by `paragraph-separate`) as starting a new paragraph. Since this means that the lines of one paragraph may have different amounts of indentation, the fill prefix used is the smallest amount of indentation of any of the lines of the paragraph. This gives good results with styles that indent a paragraph's first line more or less that the rest of the paragraph.

The fill prefix is stored in the variable `fill-prefix`. Its value is a string, or `nil` when there is no fill prefix. This is a per-buffer variable; altering the variable affects only the current buffer, but there is a default value which you can change as well. See section Local Variables.

The `indentation` text property provides another way to control the amount of indentation paragraphs receive. See section Indentation in Formatted Text.

Case Conversion Commands

Emacs has commands for converting either a single word or any arbitrary range of text to upper case or to lower case.

M-l
: Convert following word to lower case (`downcase-word`).

M-u
: Convert following word to upper case (`upcase-word`).

M-c
: Capitalize the following word (`capitalize-word`).

C-x C-l

Convert region to lower case (`downcase-region`).

C-x C-u

Convert region to upper case (`upcase-region`).

The word conversion commands are the most useful. M-l (`downcase-word`) converts the word after point to lower case, moving past it. Thus, repeating M-l converts successive words. M-u (`upcase-word`) converts to all capitals instead, while M-c (`capitalize-word`) puts the first letter of the word into upper case and the rest into lower case. All these commands convert several words at once if given an argument. They are especially convenient for converting a large amount of text from all upper case to mixed case, because you can move through the text using M-l, M-u or M-c on each word as appropriate, occasionally using M-f instead to skip a word.

When given a negative argument, the word case conversion commands apply to the appropriate number of words before point, but do not move point. This is convenient when you have just typed a word in the wrong case: you can give the case conversion command and continue typing.

If a word case conversion command is given in the middle of a word, it applies only to the part of the word which follows point. This is just like what M-d (`kill-word`) does. With a negative argument, case conversion applies only to the part of the word before point.

The other case conversion commands are C-x C-u (`upcase-region`) and C-x C-l (`downcase-region`), which convert everything between point and mark to the specified case. Point and mark do not move.

The region case conversion commands `upcase-region` and `downcase-region` are normally disabled. This means that they ask for confirmation if you try to use them. When you confirm, you may enable the command, which means it will not ask for confirmation again. See section Disabling Commands.

Text Mode

When you edit files of text in a human language, it's more convenient to use Text mode rather than Fundamental mode. Invoke M-x text-mode to enter Text mode. In Text mode, TAB runs the function `tab-to-tab-stop`, which allows you to use arbitrary tab stops set with M-x edit-tab-stops (see section Tab Stops). Features concerned with comments in programs are turned off in Text mode except when explicitly invoked. The syntax table is changed so that periods are not considered part of a word, while apostrophes, backspaces and underlines are part of words.

A similar variant mode is Indented Text mode, intended for editing text in which most lines are indented. This mode defines TAB to run `indent-relative` (see section Indentation), and makes Auto Fill indent the lines it creates. The result is that normally a line made by Auto Filling, or by LFD, is indented just like the previous line. In Indented Text mode, only blank lines separate paragraphs--indented lines continue the current paragraph. Use M-x indented-text-mode to select this mode.

Text mode, and all the modes based on it, define M-TAB as the command `ispell-complete-word`, which performs completion of the partial word in the buffer before point, using the spelling dictionary as

the space of possible words. See section Checking and Correcting Spelling.

Entering Text mode or Indented Text mode runs the hook `text-mode-hook`. Other major modes related to Text mode also run this hook, followed by hooks of their own; this includes Nroff mode, TeX mode, Outline mode and Mail mode. Hook functions on `text-mode-hook` can look at the value of `major-mode` to see which of these modes is actually being entered. See section Hooks.

Outline Mode

Outline mode is a major mode much like Text mode but intended for editing outlines. It allows you to make parts of the text temporarily invisible so that you can see the outline structure. Type M-x outline-mode to switch to Outline mode as the major mode of the current buffer.

When Outline mode makes a line invisible, the line does not appear on the screen. The screen appears exactly as if the invisible line were deleted, except that an ellipsis (three periods in a row) appears at the end of the previous visible line (only one ellipsis no matter how many invisible lines follow).

All editing commands treat the text of the invisible line as part of the previous visible line. For example, C-n moves onto the next visible line. Killing an entire visible line, including its terminating newline, really kills all the following invisible lines along with it; yanking it all back yanks the invisible lines and they remain invisible.

Outline minor mode provides the same commands as the major mode, Outline mode, but you can use it in conjunction with other major modes. Type M-x outline-minor-mode to enable the Outline minor mode in the current buffer. You can also specify this in the text of a file, with a file local variable of the form `mode: outline-minor' (see section Local Variables in Files).

The major mode, Outline mode, provides special key bindings on the C-c prefix. Outline minor mode provides similar bindings with C-c @ as the prefix; this is to reduce the conflicts with the major mode's special commands. (The variable `outline-minor-mode-prefix` controls the prefix used.)

Entering Outline mode runs the hook `text-mode-hook` followed by the hook `outline-mode-hook` (see section Hooks).

Format of Outlines

Outline mode assumes that the lines in the buffer are of two types: heading lines and body lines. A heading line represents a topic in the outline. Heading lines start with one or more stars; the number of stars determines the depth of the heading in the outline structure. Thus, a heading line with one star is a major topic; all the heading lines with two stars between it and the next one-star heading are its subtopics; and so on. Any line that is not a heading line is a body line. Body lines belong with the preceding heading line. Here is an example:

```
* Food

This is the body,
```


```
which says something about the topic of food.

** Delicious Food

This is the body of the second-level header.

** Distasteful Food

This could have
a body too, with
several lines.

*** Dormitory Food

* Shelter

Another first-level topic with its header line.
```

A heading line together with all following body lines is called collectively an entry. A heading line together with all following deeper heading lines and their body lines is called a subtree.

You can customize the criterion for distinguishing heading lines by setting the variable `outline-regexp`. Any line whose beginning has a match for this regexp is considered a heading line. Matches that start within a line (not at the left margin) do not count. The length of the matching text determines the level of the heading; longer matches make a more deeply nested level. Thus, for example, if a text formatter has commands `@chapter', `@section' and `@subsection' to divide the document into chapters and sections, you could make those lines count as heading lines by setting `outline-regexp` to `"@chap\\|@\\(sub\\)*section"'. Note the trick: the two words `chapter' and `section' are equally long, but by defining the regexp to match only `chap' we ensure that the length of the text matched on a chapter heading is shorter, so that Outline mode will know that sections are contained in chapters. This works as long as no other command starts with `@chap'.

It is possible to change the rule for calculating the level of a heading line by setting the variable `outline-level`. The value of `outline-level` should be a function that takes no arguments and returns the level of the current heading. Some major modes such as C, Nroff, and Emacs Lisp mode set this variable in order to work with Outline minor mode.

Outline mode makes a line invisible by changing the newline before it into an ASCII control-M (code 015). Most editing commands that work on lines treat an invisible line as part of the previous line because, strictly speaking, it *is* part of that line, since there is no longer a newline in between. When you save the file in Outline mode, control-M characters are saved as newlines, so the invisible lines become ordinary lines in the file. But saving does not change the visibility status of a line inside Emacs.

Outline Motion Commands

Outline mode provides special motion commands that move backward and forward to heading lines

C-c C-n
: Move point to the next visible heading line (`outline-next-visible-heading`).

C-c C-p
: Move point to the previous visible heading line (`outline-previous-visible-heading`).

C-c C-f
: Move point to the next visible heading line at the same level as the one point is on (`outline-forward-same-level`).

C-c C-b
: Move point to the previous visible heading line at the same level (`outline-backward-same-level`).

C-c C-u
: Move point up to a lower-level (more inclusive) visible heading line (`outline-up-heading`).

C-c C-n (`outline-next-visible-heading`) moves down to the next heading line. C-c C-p (`outline-previous-visible-heading`) moves similarly backward. Both accept numeric arguments as repeat counts. The names emphasize that invisible headings are skipped, but this is not really a special feature. All editing commands that look for lines ignore the invisible lines automatically.

More powerful motion commands understand the level structure of headings. C-c C-f (`outline-forward-same-level`) and C-c C-b (`outline-backward-same-level`) move from one heading line to another visible heading at the same depth in the outline. C-c C-u (`outline-up-heading`) moves backward to another heading that is less deeply nested.

Outline Visibility Commands

The other special commands of outline mode are used to make lines visible or invisible. Their names all start with `hide` or `show`. Most of them fall into pairs of opposites. They are not undoable; instead, you can undo right past them. Making lines visible or invisible is simply not recorded by the undo mechanism.

C-c C-t
: Make all body lines in the buffer invisible (`hide-body`).

C-c C-a
: Make all lines in the buffer visible (`show-all`).

C-c C-d
: Make everything under this heading invisible, not including this heading itself (`hide-subtree`).

C-c C-s
: Make everything under this heading visible, including body, subheadings, and their bodies

(`show-subtree`).

C-c C-l

Make the body of this heading line, and of all its subheadings, invisible (`hide-leaves`).

C-c C-k

Make all subheadings of this heading line, at all levels, visible (`show-branches`).

C-c C-i

Make immediate subheadings (one level down) of this heading line visible (`show-children`).

C-c C-c

Make this heading line's body invisible (`hide-entry`).

C-c C-e

Make this heading line's body visible (`show-entry`).

C-c C-q

Hide everything except the top n levels of heading lines (`hide-sublevels`).

C-c C-o

Hide everything except for the heading or body that point is in, plus the headings leading up from there to the top level of the outline (`hide-other`).

Two commands that are exact opposites are C-c C-c (`hide-entry`) and C-c C-e (`show-entry`). They are used with point on a heading line, and apply only to the body lines of that heading. Subheadings and their bodies are not affected.

Two more powerful opposites are C-c C-d (`hide-subtree`) and C-c C-s (`show-subtree`). Both expect to be used when point is on a heading line, and both apply to all the lines of that heading's subtree: its body, all its subheadings, both direct and indirect, and all of their bodies. In other words, the subtree contains everything following this heading line, up to and not including the next heading of the same or higher rank.

Intermediate between a visible subtree and an invisible one is having all the subheadings visible but none of the body. There are two commands for doing this, depending on whether you want to hide the bodies or make the subheadings visible. They are C-c C-l (`hide-leaves`) and C-c C-k (`show-branches`).

A little weaker than `show-branches` is C-c C-i (`show-children`). It makes just the direct subheadings visible--those one level down. Deeper subheadings remain invisible, if they were invisible.

Two commands have a blanket effect on the whole file. C-c C-t (`hide-body`) makes all body lines invisible, so that you see just the outline structure. C-c C-a (`show-all`) makes all lines visible. These commands can be thought of as a pair of opposites even though C-c C-a applies to more than just body lines.

The command C-c C-q (`hide-sublevels`) hides all but the top level headings. With a numeric argument n, it hides everything except the top n levels of heading lines.

The command C-c C-o (`hide-other`) hides everything except the heading or body text that point is in, plus its parents (the headers leading up from there to top level in the outline).

You can turn off the use of ellipses at the ends of visible lines by setting `selective-display-ellipses` to `nil`. Then there is no visible indication of the presence of invisible lines.

Viewing One Outline in Multiple Views

You can display two views of a single outline at the same time, in different windows, by means of an alternative implementation of Outline mode called `noutline`.

To do this, first load the library `noutline` with M-x load-library RET noutline RET. This loads the alternative implementation of Outline mode. It provides the same command names and key bindings as regular Outline mode, but it implements them differently.

Then, to display a second view of an outline buffer, you must create an indirect buffer using M-x make-indirect-buffer. The first argument of this command is the existing outline buffer name, and its second argument is the name to use for the new indirect buffer. See section Indirect Buffers.

Once the indirect buffer exists, you can display it in a window in the normal fashion, with C-x 4 b or other Emacs commands. The Outline mode commands to show and hide parts of the text operate on each buffer independently; as a result, each buffer can have its own view. If you want more than two views on the same outline, create additional indirect buffers.

In a future Emacs version, the alternative `noutline` implementation will probably become the principal implementation.

TeX Mode

TeX is a powerful text formatter written by Donald Knuth; it is also free, like GNU Emacs. LaTeX is a simplified input format for TeX, implemented by TeX macros; it comes with TeX. SliTeX is a special form of LaTeX.

Emacs has a special TeX mode for editing TeX input files. It provides facilities for checking the balance of delimiters and for invoking TeX on all or part of the file.

TeX mode has three variants, Plain TeX mode, LaTeX mode, and SliTeX mode (these three distinct major modes differ only slightly). They are designed for editing the three different formats. The command M-x tex-mode looks at the contents of the buffer to determine whether the contents appear to be either LaTeX input or SliTeX input; if so, it selects the appropriate mode. If the file contents do not appear to be LaTeX or SliTeX, it selects Plain TeX mode. If the contents are insufficient to determine this, the variable `tex-default-mode` controls which mode is used.

When M-x tex-mode does not guess right, you can use the commands M-x plain-tex-mode, M-x latex-mode, and M-x slitex-mode to select explicitly the particular variants of TeX mode.

TeX Editing Commands

Here are the special commands provided in TeX mode for editing the text of the file.

"

Insert, according to context, either `"' or `"' or `"' (`tex-insert-quote').

LFD

Insert a paragraph break (two newlines) and check the previous paragraph for unbalanced braces or dollar signs (`tex-terminate-paragraph').

M-x validate-tex-region

Check each paragraph in the region for unbalanced braces or dollar signs.

C-c {

Insert `{}' and position point between them (`tex-insert-braces').

C-c }

Move forward past the next unmatched close brace (`up-list').

In TeX, the character `"' is not normally used; we use `"' to start a quotation and `"' to end one. To make editing easier under this formatting convention, TeX mode overrides the normal meaning of the key " with a command that inserts a pair of single-quotes or backquotes (`tex-insert-quote'). To be precise, this command inserts `"' after whitespace or an open brace, `"' after a backslash, and `"' after any other character.

If you need the character `"' itself in unusual contexts, use C-q to insert it. Also, " with a numeric argument always inserts that number of `"' characters.

In TeX mode, `$' has a special syntax code which attempts to understand the way TeX math mode delimiters match. When you insert a `$' that is meant to exit math mode, the position of the matching `$' that entered math mode is displayed for a second. This is the same feature that displays the open brace that matches a close brace that is inserted. However, there is no way to tell whether a `$' enters math mode or leaves it; so when you insert a `$' that enters math mode, the previous `$' position is shown as if it were a match, even though they are actually unrelated.

TeX uses braces as delimiters that must match. Some users prefer to keep braces balanced at all times, rather than inserting them singly. Use C-c { (`tex-insert-braces') to insert a pair of braces. It leaves point between the two braces so you can insert the text that belongs inside. Afterward, use the command C-c } (`up-list') to move forward past the close brace.

There are two commands for checking the matching of braces. LFD (`tex-terminate-paragraph') checks the paragraph before point, and inserts two newlines to start a new paragraph. It prints a message in the echo area if any mismatch is found. M-x validate-tex-region checks a region, paragraph by paragraph. When it finds a paragraph that contains a mismatch, it displays point at the beginning of the paragraph for a few seconds and sets the mark at that spot. Scanning continues until the whole buffer has been checked or until you type another key. Afterward, you can use the mark ring to find the last several paragraphs that had mismatches (see section The Mark Ring).

Note that Emacs commands count square brackets and parentheses in TeX mode, not just braces. This is

not strictly correct for the purpose of checking TeX syntax. However, parentheses and square brackets are likely to be used in text as matching delimiters and it is useful for the various motion commands and automatic match display to work with them.

LaTeX Editing Commands

LaTeX mode, and its variant, SliTeX mode, provide a few extra features not applicable to plain TeX.

C-c C-o

Insert `\begin' and `\end' for LaTeX block and position point on a line between them. (`tex-latex-block`).

C-c C-e

Close the last unended block for LaTeX (`tex-close-latex-block`).

In LaTeX input, `\begin' and `\end' commands are used to group blocks of text. To insert a `\begin' and a matching `\end' (on a new line following the `\begin'), use C-c C-o (`tex-latex-block`). A blank line is inserted between the two, and point is left there. You can use completion when you enter the block type; to specify additional block type names beyond the standard list, set the variable `latex-block-names`. For example, here's how to add `theorem', `corollary', and `proof':

```
(setq latex-block-names '("theorem" "corollary" "proof"))
```

In LaTeX input, `\begin' and `\end' commands must balance. You can use C-c C-e (`tex-close-latex-block`) to insert automatically a matching `\end' to match the last unmatched `\begin'. It indents the `\end' to match the corresponding `\begin'. It inserts a newline after `\end' if point is at the beginning of a line.

TeX Printing Commands

You can invoke TeX as an inferior of Emacs on either the entire contents of the buffer or just a region at a time. Running TeX in this way on just one chapter is a good way to see what your changes look like without taking the time to format the entire file.

C-c C-r

Invoke TeX on the current region, together with the buffer's header (`tex-region`).

C-c C-b

Invoke TeX on the entire current buffer (`tex-buffer`).

C-c TAB

Invoke BibTeX on the current file (`tex-bibtex-file`).

C-c C-f

Invoke TeX on the current file (`tex-file`).

C-c C-l

Recenter the window showing output from the inferior TeX so that the last line can be seen (`tex-recenter-output-buffer`).

C-c C-k
: Kill the TeX subprocess (`tex-kill-job`).

C-c C-p
: Print the output from the last C-c C-r, C-c C-b, or C-c C-f command (`tex-print`).

C-c C-v
: Preview the output from the last C-c C-r, C-c C-b, or C-c C-f command (`tex-view`).

C-c C-q
: Show the printer queue (`tex-show-print-queue`).

You can pass the current buffer through an inferior TeX by means of C-c C-b (`tex-buffer`). The formatted output appears in a temporary file; to print it, type C-c C-p (`tex-print`). Afterward, you can use C-c C-q (`tex-show-print-queue`) to view the progress of your output towards being printed. If your terminal has the ability to display TeX output files, you can preview the output on the terminal with C-c C-v (`tex-view`).

You can specify the directory to use for running TeX by setting the variable `tex-directory`. `"."` is the default value. If your environment variable `TEXINPUTS` contains relative directory names, or if your files contains `\input' commands with relative file names, then `tex-directory` *must* be `"."` or you will get the wrong results. Otherwise, it is safe to specify some other directory, such as `"/tmp"`.

If you want to specify which shell commands are used in the inferior TeX, you can do so by setting the values of the variables `tex-run-command`, `latex-run-command`, `slitex-run-command`, `tex-dvi-print-command`, `tex-dvi-view-command`, and `tex-show-queue-command`. You *must* set the value of `tex-dvi-view-command` for your particular terminal; this variable has no default value. The other variables have default values that may (or may not) be appropriate for your system.

Normally, the file name given to these commands comes at the end of the command string; for example, `latex filename'. In some cases, however, the file name needs to be embedded in the command; an example is when you need to provide the file name as an argument to one command whose output is piped to another. You can specify where to put the file name with `*' in the command string. For example,

```
(setq tex-dvi-print-command "dvips -f * | lpr")
```

The terminal output from TeX, including any error messages, appears in a buffer called `*tex-shell*'. If TeX gets an error, you can switch to this buffer and feed it input (this works as in Shell mode; see section Interactive Inferior Shell). Without switching to this buffer you can scroll it so that its last line is visible by typing C-c C-l.

Type C-c C-k (`tex-kill-job`) to kill the TeX process if you see that its output is no longer useful. Using C-c C-b or C-c C-r also kills any TeX process still running.

You can also pass an arbitrary region through an inferior TeX by typing C-c C-r (`tex-region`). This is tricky, however, because most files of TeX input contain commands at the beginning to set parameters and define macros, without which no later part of the file will format correctly. To solve this problem,

C-c C-r allows you to designate a part of the file as containing essential commands; it is included bef the specified region as part of the input to TeX. The designated part of the file is called the header.

To indicate the bounds of the header in Plain TeX mode, you insert two special strings in the file. Insert `%**start of header' before the header, and `%**end of header' after it. Each string must appear entirely on one line, but there may be other text on the line before or after. The lines containing the two strings are included in the header. If `%**start of header' does not appear within the first 100 lines of the buffer, C-c C-r assumes that there is no header.

In LaTeX mode, the header begins with `\documentstyle' and ends with `\begin{document}'. These are commands that LaTeX requires you to use in any case, so nothing special needs to be done to identify the header.

The commands (`tex-buffer`) and (`tex-region`) do all of their work in a temporary directory, and do not have available any of the auxiliary files needed by TeX for cross-references; these commands are generally not suitable for running the final copy in which all of the cross-references need to be correct. When you want the auxiliary files, use C-c C-f (`tex-file`) which runs TeX on the current buffer's file, in that file's directory. Before TeX runs, you will be asked about saving any modified buffers. Generally, you need to use (`tex-file`) twice to get cross-references correct.

For LaTeX files, you can use BibTeX to process the auxiliary file for the current buffer's file. BibTeX looks up bibliographic citations in a data base and prepares the cited references for the bibliography section. The command C-c TAB (`tex-bibtex-file`) runs the shell command (`tex-bibtex-command`) to produce a `.bbl' file for the current buffer's file. Generally, you need to do C-c C-f (`tex-file`) once to generate the `.aux' file, then do C-c TAB (`tex-bibtex-file`), and then repeat C-c C-f (`tex-file`) twice more to get the cross-references correct.

Entering any kind of TeX mode runs the hooks `text-mode-hook` and `tex-mode-hook`. Then it runs either `plain-tex-mode-hook` or `latex-mode-hook`, whichever is appropriate. For SliTeX files, it calls `slitex-mode-hook`. Starting the TeX shell runs the hook `tex-shell-hook`. See section Hooks.

Unix TeX Distribution

TeX for Unix systems can be obtained from the University of Washington for a distribution fee.

To order a full distribution, specify whether you prefer 1/4 inch QIC-24 or 4mm DAT tape (9-track reel-to-reel is no longer available) and send $210.00 for a (tar or cpio) cartridge, payable to the University of Washington to:

```
Pierre MacKay
Department of Classics
Denny Hall, Mail Stop DH-10
University of Washington
Seattle, Washington 98195
```

Purchase orders are acceptable, but there is an extra charge of $10.00, to pay for processing charges.

For overseas orders please add $20.00 to the base cost for shipment via air parcel post, or $30.00 for shipment via courier.

The normal distribution is a tar tape, blocked 20, 1600 bpi, on an industry standard 2400 foot half-inch reel. The physical format for the 1/4 inch streamer cartridges is QIC-24. System V tapes can be written in cpio format, blocked 5120 bytes, with ASCII headers.

Nroff Mode

Nroff mode is a mode like Text mode but modified to handle nroff commands present in the text. Invoke M-x nroff-mode to enter this mode. It differs from Text mode in only a few ways. All nroff command lines are considered paragraph separators, so that filling will never garble the nroff commands. Pages are separated by `.bp' commands. Comments start with backslash-doublequote. Also, three special commands are provided that are not in Text mode:

M-n

Move to the beginning of the next line that isn't an nroff command (`forward-text-line`). An argument is a repeat count.

M-p

Like M-n but move up (`backward-text-line`).

M-?

Prints in the echo area the number of text lines (lines that are not nroff commands) in the region (`count-text-lines`).

The other feature of Nroff mode is that you can turn on Electric Nroff mode. This is a minor mode that you can turn on or off with M-x electric-nroff-mode (see section Minor Modes). When the mode is on, each time you use RET to end a line that contains an nroff command that opens a kind of grouping, the matching nroff command to close that grouping is automatically inserted on the following line. For example, if you are at the beginning of a line and type . (b RET, this inserts the matching command `.)b' on a new line following point.

If you use Outline minor mode with Nroff mode (see section Outline Mode), heading lines are lines of the form `.H' followed by a number (the header level).

Entering Nroff mode runs the hook `text-mode-hook`, followed by the hook `nroff-mode-hook` (see section Hooks).

Editing Formatted Text

Enriched mode is a minor mode for editing files that contain formatted text in WYSIWYG fashion, as in a word processor. Currently, formatted text in Enriched mode can specify fonts, colors, underlining, margins, and types of filling and justification. In the future, we plan to implement other formatting features as well.

Enriched mode is a minor mode (see section Minor Modes). Typically it is used in conjunction with Text

mode (see section Text Mode). However, you can also use it with other major modes such as Outline mode and Indented Text mode.

Potentially, Emacs can store formatted text files in various file formats. Currently, only one format is implemented: text/enriched format, which is defined by the MIME protocol. See section 'Format Conversion' in the Emacs Lisp Reference Manual, for details of how Emacs recognizes and converts file formats.

The Emacs distribution contains a formatted text file that can serve as an example. Its name is `etc/enriched.doc'. It contains samples illustrating all the features described in this section. It also contains a list of ideas for future enhancements.

Requesting to Edit Formatted Text

Whenever you visit a file that Emacs saved in the text/enriched format, Emacs automatically converts the formatting information in the file into Emacs's own internal format (text properties), and turns on Enriched mode.

To create a new file of formatted text, first visit the nonexistent file, then type M-x enriched-mode before you start inserting text. This command turns on Enriched mode. Do this before you begin inserting text, to ensure that the text you insert is handled properly.

More generally, the command `enriched-mode` turns Enriched mode on if it was off, and off if it was on. With a prefix argument, this command turns Enriched mode on if the argument is positive, and turns the mode off otherwise.

When you save a buffer while Enriched mode is enabled in it, Emacs automatically converts the text to text/enriched format while writing it into the file. When you visit the file again, Emacs will automatically recognize the format, reconvert the text, and turn on Enriched mode again.

Normally, after reading a file in text/enriched format, Emacs refills each paragraph to fit the width of the window. You can turn off this refilling, to save time, by setting the variable `enriched-fill-after-visiting` to `nil` or to `ask`.

In any case, if the window width is the same as the width with which the file was saved, Emacs trusts that the file is already properly filled.

You can add annotations for saving additional text properties, which Emacs normally does not save, by adding to `enriched-translations`. Note that the text/enriched standard requires any non-standard annotations to have names starting with `x-', as in `x-read-only'. This ensures that they will not conflict with standard annotations that may be added later.

Hard and Soft Newlines

In formatted text, Emacs distinguishes between two different kinds of newlines, hard newlines and soft newlines.

Hard newlines are used to separate paragraphs, or items in a list, or anywhere that there should always be

a line break regardless of the margins. The RET command (`newline`) and C-o (`open-line`) insert hard newlines.

Soft newlines are used to make text fit between the margins. All the fill commands, including Auto Fill, insert soft newlines--and they delete only soft newlines.

Although hard and soft newlines look the same, it is important to bear the difference in mind. Do not use RET to break lines in the middle of filled paragraphs, or else you will get hard newlines that are barriers to further filling. Instead, let Auto Fill mode break lines, so that if the text or the margins change, Emacs can refill the lines properly. See section Auto Fill Mode.

On the other hand, in tables and lists, where the lines should always remain as you type them, you can use RET to end lines. For these lines, you may also want to set the justification style to `unfilled`. See section Justification in Formatted Text.

Editing Format Information

There are two ways to alter the formatting information for a formatted text file: with keyboard commands, and with the mouse.

The easiest way to add properties to your document is by using the Text Properties menu. You can get to this menu in two ways: from the Edit menu in the menu bar, or with C-mouse-2 (hold the CTRL key and press the middle mouse button).

Most of the items in the Text Properties menu lead to other submenus. These are described in the sections that follow. Some items run commands directly:

`Remove Properties`
: Delete from the region all the text properties that the Text Properties menu works with (`facemenu-remove-props`).

`Remove All`
: Delete *all* text properties from the region (`facemenu-remove-all`).

`List Properties`
: List all the text properties of the character following point (`list-text-properties-at`).

`Display Faces`
: Display a list of all the defined faces.

`Display Colors`
: Display a list of all the defined colors.

Faces in Formatted Text

The Faces submenu lists various Emacs faces including `bold`, `italic`, and `underline`. Selecting one of these adds the chosen face to the region. See section Using Multiple Typefaces. You can also specify a face with these keyboard commands:

M-g d

Set the region, or the next inserted character, to the `default` face (`facemenu-set-default`).

M-g b

Set the region, or the next inserted character, to the `bold` face (`facemenu-set-bold`).

M-g i

Set the region, or the next inserted character, to the `italic` face (`facemenu-set-italic`).

M-g l

Set the region, or the next inserted character, to the `bold-italic` face (`facemenu-set-bold-italic`).

M-g u

Set the region, or the next inserted character, to the `underline` face (`facemenu-set-underline`).

M-g o face RET

Set the region, or the next inserted character, to the face face (`facemenu-set-face`).

If you use these commands with a prefix argument--or, in Transient Mark mode, if the region is not active--then these commands specify a face to use for your next self-inserting input. See section Transient Mark Mode. This applies to both the keyboard commands and the menu commands.

Enriched mode defines two additional faces: `excerpt` and `fixed`. These correspond to codes used in the text/enriched file format.

The `excerpt` face is intended for quotations. This face is the same as `italic` unless you customize it (see section Modifying Faces).

The `fixed` face is meant to say, "Use a fixed-width font for this part of the text." Emacs currently supports only fixed-width fonts; therefore, the `fixed` annotation is not necessary now. However, we plan to support variable width fonts in future Emacs versions, and other systems that display text/enriched format may not use a fixed-width font as the default. So if you specifically want a certain part of the text to use a fixed-width font, you should specify the `fixed` face for that part.

The `fixed` face is normally defined to use a different font from the default. However, systems have different fonts installed, you may need to customize this.

If your terminal cannot display different faces, you will not be able to see them, but you can still edit documents containing faces. You can even add faces and colors to documents. They will be visible when the file is viewed on a terminal that can display them.

Colors in Formatted Text

You can specify foreground and background colors for portions of the text. There is a menu for specifying the foreground color and a menu for specifying the background color. Each color menu lists all the colors that you have used in Enriched mode in the current Emacs session.

If you specify a color with a prefix argument--or, in Transient Mark mode, if the region is not

active--then it applies to your next self-inserting input. See section Transient Mark Mode. Otherwise command applies to the region.

Each color menu contains one additional item: `Other'. You can use this item to specify a color that is not listed in the menu; it reads the color name with the minibuffer. To display list of available colors and their names, use the `Display Colors' menu item in the Text Properties menu (see section Editing Format Information).

Any color that you specify in this way, or that is mentioned in a formatted text file that you read in, is added to both color menus for the duration of the Emacs session.

There are no key bindings for specifying colors, but you can do so with the extended commands M-x facemenu-set-foreground and M-x facemenu-set-background. Both of these commands read the name of the color with the minibuffer.

Indentation in Formatted Text

When editing formatted text, you can specify different amounts of indentation for the right or left margin of an entire paragraph or a part of a paragraph. The margins you specify automatically affect the Emacs fill commands (see section Filling Text) and line-breaking commands.

The Indentation submenu provides a convenient interface for specifying these properties. The submenu contains four items:

`Indent More`

Indent the region by 4 columns (`increase-left-margin`). In Enriched mode, this command is also available on C-x TAB; if you supply a numeric argument, that says how many columns to add to the margin (a negative argument reduces the number of columns).

`Indent Less`

Remove 4 columns of indentation from the region.

`Indent Right More`

Make the text narrower by indenting 4 columns at the right margin.

`Indent Right Less`

Remove 4 columns of indentation from the right margin.

You can use these commands repeatedly to increase or decrease the indentation.

The most common way to use these commands is to change the indentation of an entire paragraph. However, that is not the only use. You can change the margins at any point; the new values take effect at the end of the line (for right margins) or the beginning of the next line (for left margins).

This makes it possible to format paragraphs with hanging indents, which means that the first line is indented less than subsequent lines. To set up a hanging indent, increase the indentation of the region starting after the first word of the paragraph and running until the end of the paragraph.

Indenting the first line of a paragraph is easier. Set the margin for the whole paragraph where you want it to be for the body of the paragraph, then indent the first line by inserting extra spaces or tabs.

Sometimes, as a result of editing, the filling of a paragraph becomes messed up--parts of the paragraph may extend past the left or right margins. When this happens, use M-q (`fill-paragraph`) to refill the paragraph.

The variable `standard-indent` specifies how many columns these commands should add to or subtract from the indentation. The default value is 4.

Enriched mode automatically sets the variable `fill-column` based on the window width: it leaves a certain number of columns for the right margin. The variable `enriched-default-right-margin` says how many columns. The default value is 10.

Justification in Formatted Text

When editing formatted text, you can specify various styles of justification for a paragraph. The style you specify automatically affects the Emacs fill commands.

The Justification submenu provides a convenient interface for specifying the style. The submenu contains five items:

`Flush Left`

This is the most common style of justification (at least for English). Lines are aligned at the left margin but left uneven at the right.

`Flush Right`

This aligns each line with the right margin. Spaces and tabs are added on the left, if necessary, to make lines line up on the right.

`Full`

This justifies the text, aligning both edges of each line. Justified text looks very nice in a printed book, where the spaces can all be adjusted equally, but it does not look as nice with a fixed-width font on the screen. Perhaps a future version of Emacs will be able to adjust the width of spaces in a line to achieve elegant justification.

`Center`

This centers every line between the current margins.

`None`

This turns off filling entirely. Each line will remain as you wrote it; the fill and auto-fill functions will have no effect on text which has this setting. You can, however, still indent the left margin. In unfilled regions, all newlines are treated as hard newlines (see section Hard and Soft Newlines) .

In Enriched mode, you can also specify justification from the keyboard using the M-j prefix character:

M-j l

Make the region left-filled (`set-justification-left`).

M-j r

Make the region right-filled (`set-justification-right`).

M-j f

Make the region fully-justified (`set-justification-full`).

M-j c

M-S

Make the region centered (`set-justification-center`).

M-j u

Make the region unfilled (`set-justification-none`).

Justification styles apply to entire paragraphs. All the justification-changing commands operate on the paragraph containing point, or, if the region is active, on all paragraphs which overlap the region.

The default justification style is specified by the variable `default-justification`. Its value should be one of the symbols `left`, `right`, `full`, `center`, or `none`.

Setting Other Text Properties

The Other Properties menu lets you add or remove three other useful text properties: `read-only`, `invisible` and `intangible`. The `intangible` property disallows moving point within the text, the `invisible` text property hides text from display, and the `read-only` property disallows alteration of the text.

Each of these special properties has a menu item to add it to the region. The last menu item, `Remove Special', removes all of these special properties from the text in the region.

Currently, the `invisible` and `intangible` properties are *not* saved in the text/enriched format. The `read-only` property is saved, but it is not a standard part of the text/enriched format, so other editors may not respect it.

Forcing Enriched Mode

Normally, Emacs knows when you are editing formatted text because it recognizes the special annotations used in the file that you visited. However, there are situations in which you must take special actions to convert file contents or turn on Enriched mode:

- When you visit a file that was created with some other editor, Emacs may not recognize the file as being in the text/enriched format. In this case, when you visit the file you will see the formatting commands rather than the formatted text. Type M-x format-decode-buffer to translate it.
- When you *insert* a file into a buffer, rather than visiting it. Emacs does the necessary conversions on the text which you insert, but it does not enable Enriched mode. If you wish to do that, type M-x enriched-mode.

The command `format-decode-buffer` translates text in various formats into Emacs's internal format. It asks you to specify the format to translate from; however, normally you can type just RET, which tells Emacs to guess the format.

If you wish to look at a file in text/enriched format in its raw form, as a sequence of characters with no formatting, use M-x format-find-file RET filename RET RET. The empty second argument means, "read without format conversion."

TOC

Go to the previous, next section.

Go to the previous, next section.

Editing Programs

Emacs has many commands designed to understand the syntax of programming languages such as Lisp and C. These commands can

- Move over or kill balanced expressions or sexps (see section Lists and Sexps).
- Move over or mark top-level expressions---defuns, in Lisp; functions, in C (see section Defuns).
- Show how parentheses balance (see section Automatic Display Of Matching Parentheses).
- Insert, kill or align comments (see section Manipulating Comments).
- Follow the usual indentation conventions of the language (see section Indentation for Programs).

The commands for words, sentences and paragraphs are very useful in editing code even though their canonical application is for editing human language text. Most symbols contain words (see section Words); sentences can be found in strings and comments (see section Sentences). Paragraphs per se don't exist in code, but the paragraph commands are useful anyway, because programming language major modes define paragraphs to begin and end at blank lines (see section Paragraphs). Judicious use of blank lines to make the program clearer will also provide useful chunks of text for the paragraph commands to work on.

The selective display feature is useful for looking at the overall structure of a function (see section Selective Display). This feature causes only the lines that are indented less than a specified amount to appear on the screen.

Major Modes for Programming Languages

Emacs also has major modes for the programming languages Lisp, Scheme (a variant of Lisp), Awk, C, C++, Fortran, Icon, Java, Objective-C, Pascal, Perl and Tcl. There is also a major mode for makefiles, called Makefile mode.

Ideally, a major mode should be implemented for each programming language that you might want to edit with Emacs; but often the mode for one language can serve for other syntactically similar languages. The language modes that exist are those that someone decided to take the trouble to write.

There are several forms of Lisp mode, which differ in the way they interface to Lisp execution. See section Executing Lisp Expressions.

Each of the programming language modes defines the TAB key to run an indentation function that knows the indentation conventions of that language and updates the current line's indentation accordingly. For example, in C mode TAB is bound to `c-indent-line`. LFD is normally defined to do RET followed by TAB; thus, it too indents in a mode-specific fashion.

In most programming languages, indentation is likely to vary from line to line. So the major modes for

those languages rebind DEL to treat a tab as if it were the equivalent number of spaces (using the command `backward-delete-char-untabify`). This makes it possible to rub out indentation one column at a time without worrying whether it is made up of spaces or tabs. Use C-b C-d to delete a tab character before point, in these modes.

Programming language modes define paragraphs to be separated only by blank lines, so that the paragraph commands remain useful. Auto Fill mode, if enabled in a programming language major mode, indents the new lines which it creates.

Turning on a major mode runs a normal hook called the mode hook, which is the value of a Lisp variable. Each major mode has a mode hook, and the hook's name is always made from the mode command's name by adding `-hook'. For example, turning on C mode runs the hook `c-mode-hook`, while turning on Lisp mode runs the hook `lisp-mode-hook`. See section Hooks.

Lists and Sexps

By convention, Emacs keys for dealing with balanced expressions are usually Control-Meta characters. They tend to be analogous in function to their Control and Meta equivalents. These commands are usually thought of as pertaining to expressions in programming languages, but can be useful with any language in which some sort of parentheses exist (including human languages).

These commands fall into two classes. Some deal only with lists (parenthetical groupings). They see nothing except parentheses, brackets, braces (whichever ones must balance in the language you are working with), and escape characters that might be used to quote those.

The other commands deal with expressions or sexps. The word `sexp' is derived from s-expression, the ancient term for an expression in Lisp. But in Emacs, the notion of `sexp' is not limited to Lisp. It refers to an expression in whatever language your program is written in. Each programming language has its own major mode, which customizes the syntax tables so that expressions in that language count as sexps.

Sexps typically include symbols, numbers, and string constants, as well as anything contained in parentheses, brackets or braces.

In languages that use prefix and infix operators, such as C, it is not possible for all expressions to be sexps. For example, C mode does not recognize `foo + bar' as a sexp, even though it *is* a C expression; it recognizes `foo' as one sexp and `bar' as another, with the `+' as punctuation between them. This is a fundamental ambiguity: both `foo + bar' and `foo' are legitimate choices for the sexp to move over if point is at the `f'. Note that `(foo + bar)' is a single sexp in C mode.

Some languages have obscure forms of expression syntax that nobody has bothered to make Emacs understand properly.

List And Sexp Commands

C-M-f

Move forward over a sexp (`forward-sexp`).

C-M-b
: Move backward over a sexp (`backward-sexp`).

C-M-k
: Kill sexp forward (`kill-sexp`).

C-M-DEL
: Kill sexp backward (`backward-kill-sexp`).

C-M-u
: Move up and backward in list structure (`backward-up-list`).

C-M-d
: Move down and forward in list structure (`down-list`).

C-M-n
: Move forward over a list (`forward-list`).

C-M-p
: Move backward over a list (`backward-list`).

C-M-t
: Transpose expressions (`transpose-sexps`).

C-M-@
: Put mark after following expression (`mark-sexp`).

To move forward over a sexp, use C-M-f (`forward-sexp`). If the first significant character after point is an opening delimiter (`(' in Lisp; `(', `[' or `{' in C), C-M-f moves past the matching closing delimiter. If the character begins a symbol, string, or number, C-M-f moves over that.

The command C-M-b (`backward-sexp`) moves backward over a sexp. The detailed rules are like those above for C-M-f, but with directions reversed. If there are any prefix characters (single-quote, backquote and comma, in Lisp) preceding the sexp, C-M-b moves back over them as well. The sexp commands move across comments as if they were whitespace in most modes.

C-M-f or C-M-b with an argument repeats that operation the specified number of times; with a negative argument, it moves in the opposite direction.

Killing a sexp at a time can be done with C-M-k (`kill-sexp`) or C-M-DEL (`backward-kill-sexp`). C-M-k kills the characters that C-M-f would move over, and C-M-DEL kills the characters that C-M-b would move over.

The list commands move over lists like the sexp commands but skip blithely over any number of other kinds of sexps (symbols, strings, etc). They are C-M-n (`forward-list`) and C-M-p (`backward-list`). The main reason they are useful is that they usually ignore comments (since the comments usually do not contain any lists).

C-M-n and C-M-p stay at the same level in parentheses, when that's possible. To move *up* one (or n) levels, use C-M-u (`backward-up-list`). C-M-u moves backward up past one unmatched opening delimiter. A positive argument serves as a repeat count; a negative argument reverses direction of motion

and also requests repetition, so it moves forward and up one or more levels.

To move *down* in list structure, use C-M-d (`down-list`). In Lisp mode, where `(' is the only opening delimiter, this is nearly the same as searching for a `('. An argument specifies the number of levels of parentheses to go down.

A somewhat random-sounding command which is nevertheless handy is C-M-t (`transpose-sexps`), which drags the previous sexp across the next one. An argument serves as a repeat count, and a negative argument drags backwards (thus canceling out the effect of C-M-t with a positive argument). An argument of zero, rather than doing nothing, transposes the sexps ending after point and the mark.

To set the region around the next sexp in the buffer, use C-M-@ (`mark-sexp`), which sets mark at the same place that C-M-f would move to. C-M-@ takes arguments like C-M-f. In particular, a negative argument is useful for putting the mark at the beginning of the previous sexp.

The list and sexp commands' understanding of syntax is completely controlled by the syntax table. Any character can, for example, be declared to be an opening delimiter and act like an open parenthesis. See section The Syntax Table.

Defuns

In Emacs, a parenthetical grouping at the top level in the buffer is called a defun. The name derives from the fact that most top-level lists in a Lisp file are instances of the special form `defun`, but any top-level parenthetical grouping counts as a defun in Emacs parlance regardless of what its contents are, and regardless of the programming language in use. For example, in C, the body of a function definition is a defun.

C-M-a
: Move to beginning of current or preceding defun (`beginning-of-defun`).

C-M-e
: Move to end of current or following defun (`end-of-defun`).

C-M-h
: Put region around whole current or following defun (`mark-defun`).

The commands to move to the beginning and end of the current defun are C-M-a (`beginning-of-defun`) and C-M-e (`end-of-defun`).

If you wish to operate on the current defun, use C-M-h (`mark-defun`) which puts point at the beginning and mark at the end of the current or next defun. For example, this is the easiest way to get ready to move the defun to a different place in the text. In C mode, C-M-h runs the function `mark-c-function`, which is almost the same as `mark-defun`; the difference is that it backs up over the argument declarations, function name and returned data type so that the entire C function is inside the region. See section Commands to Mark Textual Objects.

Emacs assumes that any open-parenthesis found in the leftmost column is the start of a defun. Therefore, **never put an open-parenthesis at the left margin in a Lisp file unless it is the start of a top level list. Never put an open-brace or other opening delimiter at the beginning of a line of C code unless it**

starts the body of a function. The most likely problem case is when you want an opening delimiter at the start of a line inside a string. To avoid trouble, put an escape character (`\', in C and Emacs Lisp, some other Lisp dialects) before the opening delimiter. It will not affect the contents of the string.

In the remotest past, the original Emacs found defuns by moving upward a level of parentheses until there were no more levels to go up. This always required scanning all the way back to the beginning of the buffer, even for a small function. To speed up the operation, Emacs was changed to assume that any `(' (or other character assigned the syntactic class of opening-delimiter) at the left margin is the start of a defun. This heuristic is nearly always right and avoids the costly scan; however, it mandates the convention described above.

Indentation for Programs

The best way to keep a program properly indented is to use Emacs to re-indent it as you change it. Emacs has commands to indent properly either a single line, a specified number of lines, or all of the lines inside a single parenthetical grouping.

Emacs also provides a Lisp pretty-printer in the library `pp`. This program prints a Lisp object with indentation chosen to look nice.

Basic Program Indentation Commands

TAB
: Adjust indentation of current line.

LFD
: Equivalent to RET followed by TAB (`newline-and-indent`).

The basic indentation command is TAB, which gives the current line the correct indentation as determined from the previous lines. The function that TAB runs depends on the major mode; it is `lisp-indent-line` in Lisp mode, `c-indent-line` in C mode, etc. These functions understand different syntaxes for different languages, but they all do about the same thing. TAB in any programming language major mode inserts or deletes whitespace at the beginning of the current line, independent of where point is in the line. If point is inside the whitespace at the beginning of the line, TAB leaves it at the end of that whitespace; otherwise, TAB leaves point fixed with respect to the characters around it.

Use C-q TAB to insert a tab at point.

When entering lines of new code, use LFD (`newline-and-indent`), which is equivalent to a RET followed by a TAB. LFD creates a blank line, and then gives it the appropriate indentation.

TAB indents the second and following lines of the body of a parenthetical grouping each under the preceding one; therefore, if you alter one line's indentation to be nonstandard, the lines below will tend to follow it. This behavior is convenient in cases where you have overridden the standard result of TAB because you find it unaesthetic for a particular line.

Remember that an open-parenthesis, open-brace or other opening delimiter at the left margin is assumed

by Emacs (including the indentation routines) to be the start of a function. Therefore, you must never have an opening delimiter in column zero that is not the beginning of a function, not even inside a string. This restriction is vital for making the indentation commands fast; you must simply accept it. See section Defuns, for more information on this.

Indenting Several Lines

When you wish to re-indent several lines of code which have been altered or moved to a different level in the list structure, you have several commands available.

C-M-q

Re-indent all the lines within one list (`indent-sexp`).

C-u TAB

Shift an entire list rigidly sideways so that its first line is properly indented.

C-M-\

Re-indent all lines in the region (`indent-region`).

You can re-indent the contents of a single list by positioning point before the beginning of it and typing C-M-q (`indent-sexp` in Lisp mode, `indent-c-exp` in C mode; also bound to other suitable commands in other modes). The indentation of the line the sexp starts on is not changed; therefore, only the relative indentation within the list, and not its position, is changed. To correct the position as well, type a TAB before the C-M-q.

If the relative indentation within a list is correct but the indentation of its first line is not, go to that line and type C-u TAB. TAB with a numeric argument reindents the current line as usual, then reindents by the same amount all the lines in the grouping starting on the current line. In other words, it reindents the whole grouping rigidly as a unit. It is clever, though, and does not alter lines that start inside strings, or C preprocessor lines when in C mode.

Another way to specify the range to be re-indented is with the region. The command C-M-\ (`indent-region`) applies TAB to every line whose first character is between point and mark.

Customizing Lisp Indentation

The indentation pattern for a Lisp expression can depend on the function called by the expression. For each Lisp function, you can choose among several predefined patterns of indentation, or define an arbitrary one with a Lisp program.

The standard pattern of indentation is as follows: the second line of the expression is indented under the first argument, if that is on the same line as the beginning of the expression; otherwise, the second line is indented underneath the function name. Each following line is indented under the previous line whose nesting depth is the same.

If the variable `lisp-indent-offset` is non-`nil`, it overrides the usual indentation pattern for the second line of an expression, so that such lines are always indented `lisp-indent-offset` more columns than the containing list.

The standard pattern is overridden for certain functions. Functions whose names start with `def` alwa indent the second line by `lisp-body-indent` extra columns beyond the open-parenthesis startin expression.

The standard pattern can be overridden in various ways for individual functions, according to the `lisp-indent-function` property of the function name. There are four possibilities for this property:

`nil`

This is the same as no property; the standard indentation pattern is used.

`defun`

The pattern used for function names that start with `def` is used for this function also.

a number, number

The first number arguments of the function are distinguished arguments; the rest are considered the body of the expression. A line in the expression is indented according to whether the first argument on it is distinguished or not. If the argument is part of the body, the line is indented `lisp-body-indent` more columns than the open-parenthesis starting the containing expression. If the argument is distinguished and is either the first or second argument, it is indented *twice* that many extra columns. If the argument is distinguished and not the first or second argument, the standard pattern is followed for that line.

a symbol, symbol

symbol should be a function name; that function is called to calculate the indentation of a line within this expression. The function receives two arguments:

state

The value returned by `parse-partial-sexp` (a Lisp primitive for indentation and nesting computation) when it parses up to the beginning of this line.

pos

The position at which the line being indented begins.

It should return either a number, which is the number of columns of indentation for that line, or a list whose car is such a number. The difference between returning a number and returning a list is that a number says that all following lines at the same nesting level should be indented just like this one; a list says that following lines might call for different indentations. This makes a difference when the indentation is being computed by C-M-q; if the value is a number, C-M-q need not recalculate indentation for the following lines until the end of the list.

Commands for C Indentation

Here are the commands for indentation in C mode and related modes:

`C-c C-q`

Reindent the current top-level function definition or aggregate type declaration (`c-indent-defun`).

`C-M-q`

Reindent each line in the balanced expression that follows point (`c-indent-exp`). A prefix argument inhibits error checking and warning messages about invalid syntax.

`TAB`

Reindent the current line, and/or in some cases insert a tab character (`c-indent-command`).

If `c-tab-always-indent` is `t`, this command always reindents the current line and does nothing else. This is the default.

If that variable is `nil`, this command reindents the current line only if point is at the left margin or in the line's indentation; otherwise, it inserts a tab.

Any other value (not `nil` or `t`) means always reindent the line, and also insert a tab if within a comment, a string, or a preprocessor directive.

`C-u TAB`

Reindent the current line according to its syntax; also rigidly reindent any other lines of the expression that starts on the current line. See section Indenting Several Lines.

To reindent the whole current buffer, type C-x h C-M-\. This first selects the whole buffer as the region, then reindents that region.

To reindent the current block, use C-M-u C-M-q. This moves to the front of the block and then reindents it all.

Customizing C Indentation

C mode and related modes use a simple yet flexible mechanism for customizing indentation. The mechanism works in two steps: first it classifies the line syntactically according to its contents and context; second, it associates each kind of syntactic construct with an indentation offset which you can customize.

Step 1--Syntactic Analysis

In the first step, the C indentation mechanism looks at the line you are currently indenting and determines the syntactic components of the construct on that line. It builds a list of these syntactic components, where each component on the list contains a syntactic symbol and a relative buffer position. Syntactic symbols describe grammatical elements such as statement, substatement, class-open, class-close, knr-argdecl, etc.

Conceptually, a line of C code is always indented relative to the indentation of some line higher up in the buffer. This is represented by the relative buffer positions in the syntactic component list.

Here is an example. Suppose we have the following code in a C++ mode buffer (the line numbers don't actually appear in the buffer):

```
1: void swap (int& a, int& b)
2: {
3:   int tmp = a;
```


```
4:    a = b;
5:    b = tmp;
6: }
```

If you type C-c C-s (which runs the command `c-show-syntactic-information`) on line 4, it shows the result of the indentation mechanism for that line:

```
((statement . 32))
```

This indicates that the line is a statement and it is indented relative to buffer position 32, which happens to be the `i' in `int` on line 3. If you move the cursor to line 3 and type C-c C-s, it displays this:

```
((defun-block-intro . 28))
```

This indicates that the `int` line is the first statement in a block, and is indented relative to buffer position 28, which is the brace just after the function header.

Here is another example:

```
1: int add (int val, int incr, int doit)
2: {
3:   if (doit)
4:     {
5:       return (val + incr);
6:     }
7:   return (val);
8: }
```

Typing C-c C-s on line 4 displays this:

```
((substatement-open . 43))
```

This says that the brace *opens* a substatement block. By the way, a substatement indicates the line after an `if`, `else`, `while`, `do`, `switch`, and `for` statements.

After a line has been analyzed syntactically for indentation, the global variable `c-syntactic-context` contains a list that describes the results. Each element in this list is a a syntactic component: a cons cell containing a syntactic symbol and (optionally) its corresponding buffer position. There may be several elements in a component list; typically only one element has a buffer position.

Step 2--Indentation Calculation

The C indentation mechanism calculates the indentation for the current line using the list of syntactic components, `c-syntactic-context`, derived from syntactic analysis. Each component is a cons cell that contains a syntactic symbol and may also contain a pointer to a location in the buffer.

Each component contributes to the final total indentation of the line in two ways. First, the syntactic

symbol identifies an element of `c-offsets-alist`, which is an association list mapping syntacti symbols into indentation offsets. Each syntactic symbol's offset adds to the total indentation. Second, if the component includes a buffer position, the column number of that position adds to the indentation. All these offsets and column numbers, added together, give the total indentation.

The following examples demonstrate the workings of the C indentation mechanism:

```
1: void swap (int& a, int& b)
2: {
3:   int tmp = a;
4:   a = b;
5:   b = tmp;
6: }
```

Suppose that point is on line 3 and you type TAB to reindent the line. As explained above (see section Step 1--Syntactic Analysis), the syntactic component list for that line is:

```
((defun-block-intro . 28))
```

In this case, the indentation calculation first looks up `defun-block-intro` in the `c-offsets-alist` alist. Suppose that it finds the integer 2; it adds this to the running total (initialized to zero), yielding a updated total indentation of 2 spaces.

The next step is to find the column number of buffer position 28. Since the brace at buffer position 28 is in column zero, this adds 0 to the running total. Since this line has only one syntactic component, the total indentation for the line is 2 spaces.

```
1: int add (int val, int incr, int doit)
2: {
3:   if (doit)
4:     {
5:       return(val + incr);
6:     }
7:   return(val);
8: }
```

If you type TAB on line 4, the same process is performed, but with different data. The syntactic component list for this line is:

```
((substatement-open . 43))
```

Here, the indentation calculation's first job is to look up the symbol `substatement-open` in `c-offsets-alist`. Let's assume that the offset for this symbol is 2. At this point the running total is 2 (0 + 2 = 2). Then it adds the column number of buffer position 43, which is the `i' in `if` on line 3. This character is in column 2 on that line. Adding this yields a total indentation of 4 spaces.

If a syntactic symbol in the analysis of a line does not appear in `c-offsets-alist`, it is ignored; if in

addition the variable `c-strict-syntax-p` is non-`nil`, it is an error.

Changing Indentation Style

There are two ways to customize the indentation style for the C modes. First, you can select one of several predefined styles, each of which specifies offsets for all the syntactic symbols. For more flexibility, you can customize the handling of individual syntactic symbols. See section Syntactic Symbols, for a list of all defined syntactic symbols.

M-x c-set-style RET style RET

> Select predefined indentation style style. Type ? when entering style to see a list of supported styles; to find out what a style looks like, select it and reindent some C code.

C-c C-o symbol RET offset RET

> Set the indentation offset for syntactic symbol symbol (`c-set-offset`). The second argument offset specifies the new indentation offset.

The `c-offsets-alist` variable controls the amount of indentation to give to each syntactic symbol. Its value is an association list, and each element of the list has the form `(syntactic-symbol . offset)`. By changing the offsets for various syntactic symbols, you can customize indentation in fine detail.

Each offset value in `c-offsets-alist` can be an integer, a function or variable name, or one of the following symbols: `+`, `-`, `++`, or `--`, indicating positive or negative multiples of the variable `c-basic-offset`. Thus, if you want to change the levels of indentation to be 3 spaces instead of 2 spaces, set `c-basic-offset` to 3.

Using a function as the offset value provides the ultimate flexibility in customizing indentation. The function is called with a single argument containing the `cons` of the syntactic symbol and the relative indent point. The function should return an integer offset.

The command C-c C-o (`c-set-offset`) is the easiest way to set offsets, both interactively or in your `` `~/.emacs' `` file. First specify the syntactic symbol, then the offset you want. See section Syntactic Symbols, for a list of valid syntactic symbols and their meanings.

The variable `c-offsets-alist-default` holds the default settings for the offsets of the syntactic symbols. *Do not change this value!*

Syntactic Symbols

Here is a table of valid syntactic symbols for C mode indentation, with their syntactic meanings. Normally, most of these symbols are assigned offsets in `c-offsets-alist`.

`string`

> Inside a multi-line string.

`c`

> Inside a multi-line C style block comment.

`defun-open`

On a brace that opens a function definition.

`defun-close`
: On a brace that closes a function definition.

`defun-block-intro`
: In the first line in a top-level defun.

`class-open`
: On a brace that opens a class definition.

`class-close`
: On a brace that closes a class definition.

`inline-open`
: On a brace that opens an in-class inline method.

`inline-close`
: On a brace that closes an in-class inline method.

`ansi-funcdecl-cont`
: In the nether region between an ANSI function declaration and the defun opening brace.

`knr-argdecl-intro`
: On the first line of a K&R C argument declaration.

`knr-argdecl`
: In one of the subsequent lines in a K&R C argument declaration.

`topmost-intro`
: On the first line in a topmost construct definition.

`topmost-intro-cont`
: On the topmost definition continuation lines.

`member-init-intro`
: On the first line in a member initialization list.

`member-init-cont`
: On one of the subsequent member initialization list lines.

`inher-intro`
: On the first line of a multiple inheritance list.

`inher-cont`
: On one of the subsequent multiple inheritance lines.

`block-open`
: On a statement block open brace.

`block-close`
: On a statement block close brace.

`brace-list-open`

On the opening brace of an `enum` or `static` array list.

`brace-list-close`
: On the closing brace of an `enum` or `static` array list.

`brace-list-intro`
: On the first line in an `enum` or `static` array list.

`brace-list-entry`
: On one of the subsequent lines in an `enum` or `static` array list.

`statement`
: On an ordinary statement.

`statement-cont`
: On a continuation line of a statement.

`statement-block-intro`
: On the first line in a new statement block.

`statement-case-intro`
: On the first line in a `case` "block".

`statement-case-open`
: On the first line in a `case` block starting with brace.

`substatement`
: On the first line after an `if`, `while`, `for`, `do`, or `else`.

`substatement-open`
: On the brace that opens a substatement block.

`case-label`
: On a `case` or `default` label.

`access-label`
: On a C++ `private`, `protected`, or `public` access label.

`label`
: On any ordinary label.

`do-while-closure`
: On the `while` that ends a `do-while` construct.

`else-clause`
: On the `else` of an `if-else` construct.

`comment-intro`
: On a line containing only a comment introduction.

`arglist-intro`
: On the first line in an argument list.

`arglist-cont`

On one of the subsequent argument list lines when no arguments follow on the same line as th arglist opening parenthesis.

`arglist-cont-nonempty`

On one of the subsequent argument list lines when at least one argument follows on the same line as the arglist opening parenthesis.

`arglist-close`

On the closing parenthesis of an argument list.

`stream-op`

On one of the lines continuing a stream operator construct.

`inclass`

On a construct which is nested inside a class definition.

`cpp-macro`

On the start of a cpp macro.

`friend`

On a C++ `friend` declaration.

`objc-method-intro`

On the first line of an Objective-C method definition.

`objc-method-args-cont`

On one of the lines continuing an Objective-C method definition.

`objc-method-call-cont`

On one of the lines continuing an Objective-C method call.

Variables for C Indentation

This section describes additional variables which control the indentation behavior of C mode and related mode.

`c-offsets-alist`

Association list of syntactic symbols and their indentation offsets. See section Changing Indentation Style, for details.

`c-offsets-alist-default`

Default settings for the offsets of the syntactic symbols. See section Changing Indentation Style.

`c-style-alist`

Variable for specifying styles of indentation; see below.

`c-basic-offset`

Amount of basic offset used by `+` and `-` symbols in `c-offsets-alist`.

`c-recognize-knr-p`

If this variable is non-`nil`, C mode and Objective C mode recognize K&R constructs. This variable is needed because of ambiguities in C syntax that make recognition of K&R constructs

problematic and slow. If you always use ANSI C prototype syntax, set this variable to `nil` to speed up C indentation.

This variable is `nil` by default in C++ mode, and `t` by default in C mode and Objective C mode.

`c-special-indent-hook`

Hook for user-defined special indentation adjustments. This hook is called after a line is indented by C mode and related modes.

The variable `c-style-alist` specifies the predefined indentation styles. Each element has form `(name variable-setting...)`, where name is the name of the style. Each variable-setting has the form `(variable . value)`; variable is one of the customization variables used by C mode, and value is the value for that variable when using the selected style.

When variable is `c-offsets-alist`, that is a special case: value is appended to the front of the value of `c-offsets-alist` instead of replacing that value outright. Therefore, it is not necessary for value to specify each and every syntactic symbol--only those for which the style differs from the default.

The indentation of lines containing only comments is also affected by the variable `c-comment-only-line-offset` (see section Comments in C Modes).

C Indentation Styles

A C style is a collection of indentation style customizations. Emacs comes with several predefined indentation styles for C code including `gnu`, `k&r`, `bsd`, `stroustrup` `whitesmith`, `ellemtel`, and `cc-mode`. The default style is `gnu`.

To choose the style you want, use the command M-x c-set-style. Specify a style name as an argument (case is not significant in C style names). The chosen style only affects newly visited buffers, not those you are already editing.

To define a new C indentation style, call the function `c-add-style`:

```
(c-add-style name values use-now)
```

Here name is the name of the new style (a string), and values is an alist whose elements have the form `(variable . value)`. The variables you specify should be among those documented in section Variables for C Indentation.

If use-now is non-`nil`, `c-add-style` switches to the new style after defining it.

Automatic Display Of Matching Parentheses

The Emacs parenthesis-matching feature is designed to show automatically how parentheses match in the text. Whenever you type a self-inserting character that is a closing delimiter, the cursor moves momentarily to the location of the matching opening delimiter, provided that is on the screen. If it is not on the screen, some text near it is displayed in the echo area. Either way, you can tell what grouping is being closed off.

In Lisp, automatic matching applies only to parentheses. In C, it applies to braces and brackets too. Emacs knows which characters to regard as matching delimiters based on the syntax table, which is set by the major mode. See section The Syntax Table.

If the opening delimiter and closing delimiter are mismatched--such as in `[x)'---a warning message is displayed in the echo area. The correct matches are specified in the syntax table.

Three variables control parenthesis match display. `blink-matching-paren` turns the feature on or off; `nil` turns it off, but the default is `t` to turn match display on. `blink-matching-delay` says how many seconds to wait; the default is 1, but on some systems it is useful to specify a fraction of a second. `blink-matching-paren-distance` specifies how many characters back to search to find the matching opening delimiter. If the match is not found in that far, scanning stops, and nothing is displayed. This is to prevent scanning for the matching delimiter from wasting lots of time when there is no match. The default is 12,000.

When using X Windows, you can request a more powerful kind of automatic parenthesis matching by loading the `paren` library. To load it, type M-x load-library RET paren RET. This library turns off the usual kind of matching parenthesis display and substitutes another: whenever point is after a close parenthesis, the close parenthesis and its matching open parenthesis are both highlighted; otherwise, if point is before an open parenthesis, the matching close parenthesis is highlighted. (There is no need to highlight the open parenthesis after point because the cursor appears on top of that character.)

Manipulating Comments

Because comments are such an important part of programming, Emacs provides special commands for editing and inserting comments.

Comment Commands

The comment commands insert, kill and align comments.

M-;
: Insert or align comment (`indent-for-comment`).

C-x ;
: Set comment column (`set-comment-column`).

C-u - C-x ;
: Kill comment on current line (`kill-comment`).

M-LFD
: Like RET followed by inserting and aligning a comment (`indent-new-comment-line`).

M-x comment-region
: Add or remove comment delimiters on all the lines in the region.

The command that creates a comment is M-; (`indent-for-comment`). If there is no comment already on the line, a new comment is created, aligned at a specific column called the comment column.

The comment is created by inserting the string Emacs thinks comments should start with (the value of `comment-start`; see below). Point is left after that string. If the text of the line extends past the comment column, then the indentation is done to a suitable boundary (usually, at least one space is inserted). If the major mode has specified a string to terminate comments, that is inserted after point, to keep the syntax valid.

M-; can also be used to align an existing comment. If a line already contains the string that starts comments, then M-; just moves point after it and re-indents it to the conventional place. Exception: comments starting in column 0 are not moved.

Some major modes have special rules for indenting certain kinds of comments in certain contexts. For example, in Lisp code, comments which start with two semicolons are indented as if they were lines of code, instead of at the comment column. Comments which start with three semicolons are supposed to start at the left margin. Emacs understands these conventions by indenting a double-semicolon comment using TAB, and by not changing the indentation of a triple-semicolon comment at all.

```
;; This function is just an example
;;; Here either two or three semicolons are appropriate.
(defun foo (x)
;;; And now, the first part of the function:
  ;; The following line adds one.
  (1+ x))           ; This line adds one.
```

In C code, a comment preceded on its line by nothing but whitespace is indented like a line of code.

Even when an existing comment is properly aligned, M-; is still useful for moving directly to the start of the comment.

C-u - C-x ; (`kill-comment`) kills the comment on the current line, if there is one. The indentation before the start of the comment is killed as well. If there does not appear to be a comment in the line, nothing is done. To reinsert the comment on another line, move to the end of that line, do C-y, and then do M-; to realign it. Note that C-u - C-x ; is not a distinct key; it is C-x ; (`set-comment-column`) with a negative argument. That command is programmed so that when it receives a negative argument it calls `kill-comment`. However, `kill-comment` is a valid command which you could bind directly to a key if you wanted to.

Multiple Lines of Comments

If you are typing a comment and wish to continue it on another line, you can use the command M-LFD (`indent-new-comment-line`). This terminates the comment you are typing, creates a new blank line afterward, and begins a new comment indented under the old one. When Auto Fill mode is on, going past the fill column while typing a comment causes the comment to be continued in just this fashion. If point is not at the end of the line when M-LFD is typed, the text on the rest of the line becomes part of the new comment line.

To turn existing lines into comment lines, use the M-x comment-region command. It adds comment delimiters to the lines that start in the region, thus commenting them out. With a negative argument, it

does the opposite--it deletes comment delimiters from the lines in the region.

With a positive argument, `comment-region` duplicates the last character of the comment start sequence it adds; the argument specifies how many copies of the character to insert. Thus, in Lisp mode, C-u 2 M-x comment-region adds `;;' to each line. Duplicating the comment delimiter is a way of calling attention to the comment. It can also affect how the comment is indented. In Lisp, for proper indentation, you should use an argument of two, if between defuns, and three, if within a defun.

Options Controlling Comments

The comment column is stored in the variable `comment-column`. You can set it to a number explicitly. Alternatively, the command C-x ; (`set-comment-column`) sets the comment column to the column point is at. C-u C-x ; sets the comment column to match the last comment before point in the buffer, and then does a M-; to align the current line's comment under the previous one. Note that C-u - C-x ; runs the function `kill-comment` as described above.

The variable `comment-column` is per-buffer: setting the variable in the normal fashion affects only the current buffer, but there is a default value which you can change with `setq-default`. See section Local Variables. Many major modes initialize this variable for the current buffer.

The comment commands recognize comments based on the regular expression that is the value of the variable `comment-start-skip`. Make sure this regexp does not match the null string. It may match more than the comment starting delimiter in the strictest sense of the word; for example, in C mode the value of the variable is `"/\\*+ *"`, which matches extra stars and spaces after the `/*' itself. (Note that `\\' is needed in Lisp syntax to include a `\' in the string, which is needed to deny the first star its special meaning in regexp syntax. See section Syntax of Regular Expressions.)

When a comment command makes a new comment, it inserts the value of `comment-start` to begin it. The value of `comment-end` is inserted after point, so that it will follow the text that you will insert into the comment. In C mode, `comment-start` has the value `"/* "` and `comment-end` has the value `" */"`.

The variable `comment-multi-line` controls how M-LFD (`indent-new-comment-line`) behaves when used inside a comment. If `comment-multi-line` is `nil`, as it normally is, then the comment on the starting line is terminated and a new comment is started on the new following line. If `comment-multi-line` is not `nil`, then the new following line is set up as part of the same comment that was found on the starting line. This is done by not inserting a terminator on the old line, and not inserting a starter on the new line. In languages where multi-line comments work, the choice of value for this variable is a matter of taste.

The variable `comment-indent-function` should contain a function that will be called to compute the indentation for a newly inserted comment or for aligning an existing comment. It is set differently by various major modes. The function is called with no arguments, but with point at the beginning of the comment, or at the end of a line if a new comment is to be inserted. It should return the column in which the comment ought to start. For example, in Lisp mode, the indent hook function bases its decision on how many semicolons begin an existing comment, and on the code in the preceding lines.

Editing Without Unbalanced Parentheses

M-(

Put parentheses around next sexp(s) (`insert-parentheses`).

M-)

Move past next close parenthesis and re-indent (`move-over-close-and-reindent`).

The commands M-((`insert-parentheses`) and M-) (`move-over-close-and-reindent`) are designed to facilitate a style of editing which keeps parentheses balanced at all times. M-(inserts a pair of parentheses, either together as in `()', or, if given an argument, around the next several sexps. It leaves point after the open parenthesis. The command M-) moves past the close parenthesis, deleting any indentation preceding it (in this example there is none), and indenting with LFD after it.

For example, instead of typing (F O O), you can type M-(F O O, which has the same effect except for leaving the cursor before the close parenthesis.

M-(may insert a space before the open parenthesis, depending on the syntax class of the preceding character. Set `parens-dont-require-spaces` to a non-`nil` value if you wish to inhibit this.

Completion for Symbol Names

Usually completion happens in the minibuffer. But one kind of completion is available in all buffers: completion for symbol names.

The character M-TAB runs a command to complete the partial symbol before point against the set of meaningful symbol names. Any additional characters determined by the partial name are inserted at point.

If the partial name in the buffer has more than one possible completion and they have no additional characters in common, a list of all possible completions is displayed in another window.

There are two ways of determining the set of legitimate symbol names to complete against. In most major modes, this uses a tags table (see section Tags Tables); the legitimate symbol names are the tag names listed in the tags table file. The command which implements this is `complete-tag`.

In Emacs-Lisp mode, the name space for completion normally consists of nontrivial symbols present in Emacs--those that have function definitions, values or properties. However, if there is an open-parenthesis immediately before the beginning of the partial symbol, only symbols with function definitions are considered as completions. The command which implements this is `lisp-complete-symbol`.

In Text mode and related modes, M-TAB completes words based on the spell-checker's dictionary. See section Checking and Correcting Spelling.

Documentation Commands

As you edit Lisp code to be run in Emacs, the commands C-h f (`describe-function`) and C-h v (`describe-variable`) can be used to print documentation of functions and variables that you want to call. These commands use the minibuffer to read the name of a function or variable to document, and display the documentation in a window.

For extra convenience, these commands provide default arguments based on the code in the neighborhood of point. C-h f sets the default to the function called in the innermost list containing point. C-h v uses the symbol name around or adjacent to point as its default.

Documentation on operating system commands, library functions and system calls can be obtained with the M-x manual-entry command. This reads a topic as an argument, and displays the "man page" on that topic. `manual-entry` starts a background process that formats the manual page, by running the `man` program. The result goes in a buffer named `*man topic*'. These buffers use a special major mode, Man mode, that facilitates scrolling and examining other manual pages. For details, type C-h m while in a man page buffer.

For a long man page, setting the faces properly can take substantial time. By default, Emacs uses faces in man pages if you are using X Windows. You can turn off use of faces in man pages by setting the variable `Man-fontify-manpage-flag` to `nil`.

If you insert the text of a man page into an Emacs buffer in some other fashion, you can use the command M-x Man-fontify-manpage to perform the same conversions that M-x manual-entry does.

Eventually the GNU project hopes to replace most man pages with better-organized manuals that you can browse with Info. See section Other Help Commands. Since this process is only partially completed, it is still useful to read manual pages.

Change Logs

The Emacs command C-x 4 a adds a new entry to the change log file for the file you are editing (`add-change-log-entry-other-window`).

A change log file contains a chronological record of when and why you have changed a program, consisting of a sequence of entries describing individual changes. Normally it is kept in a file called `` `ChangeLog' `` in the same directory as the file you are editing, or one of its parent directories. A single `` `ChangeLog' `` file can record changes for all the files in its directory and all its subdirectories.

A change log entry starts with a header line that contains your name, your email address (taken from the variable `user-mail-address`), and the current date and time. Aside from these header lines, every line in the change log starts with a space or a tab. The bulk of the entry consists of items, each of which starts with a line starting with whitespace and a star. Here are two entries, each with two items:

@medbreak

```
Wed May  5 14:11:45 1993  Richard Stallman  <rms@gnu.ai.mit.edu>
```


```
        * man.el: Rename symbols `man-*' to `Man-*'.
        (manual-entry): Make prompt string clearer.

        * simple.el (blink-matching-paren-distance):
        Change default to 12,000.

Tue May  4 12:42:19 1993  Richard Stallman  <rms@gnu.ai.mit.edu>

        * vc.el (minor-mode-map-alist): Don't use it if it's void.
        (vc-cancel-version): Doc fix.
```

One entry can describe several changes; each change should have its own item. Normally there should be a blank line between items. When items are related (parts of the same change, in different places), group them by leaving no blank line between them. The second entry above contains two items grouped in this way.

C-x 4 a visits the change log file and creates a new entry unless the most recent entry is for today's date and your name. It also creates a new item for the current file. For many languages, it can even guess the name of the function or other object that was changed.

The change log file is visited in Change Log mode. In this major mode, each bunch of grouped items counts as one paragraph, and each entry is considered a page. This facilitates editing the entries. LFD and auto-fill indent each new line like the previous line; this is convenient for entering the contents of an entry.

Version control systems are another way to keep track of changes in your program and keep a change log. See section Log Entries.

Tags Tables

A tags table is a description of how a multi-file program is broken up into files. It lists the names of the component files and the names and positions of the functions (or other named subunits) in each file. Grouping the related files makes it possible to search or replace through all the files with one command. Recording the function names and positions makes possible the M-. command which finds the definition of a function by looking up which of the files it is in.

Tags tables are stored in files called tags table files. The conventional name for a tags table file is `TAGS'.

Each entry in the tags table records the name of one tag, the name of the file that the tag is defined in (implicitly), and the position in that file of the tag's definition.

Just what names from the described files are recorded in the tags table depends on the programming language of the described file. They normally include all functions and subroutines, and may also include global variables, data types, and anything else convenient. Each name recorded is called a tag.

Source File Tag Syntax

- In Lisp code, any function defined with `defun`, any variable defined with `defvar` or `defconst`, and in general the first argument of any expression that starts with `(def' in column zero, is a tag.
- In Scheme code, tags include anything defined with `def` or with a construct whose name starts with `def'. They also include variables set with `set!` at top level in the file.
- In C code, any C function or typedef is a tag, and so are definitions of struct, union and enum. Any `#define` is also a tag, unless `--no-defines' is specified when the tags table is constructed, which sometimes makes the tags file much smaller. In C++ code, member functions are also recognized.
- In Yacc or Bison input files, each rule defines as a tag the nonterminal it constructs. The portions of the file that contain C code are parsed as C code.
- In Fortran code, functions and subroutines are tags.
- In Pascal code, the tags are the functions and procedures defined in the file.
- In Perl code, the tags are the procedures defined by the `sub` keyword.
- In Prolog code, a tag name appears at the left margin.
- In Erlang code, the tags are the functions, records, and macros defined in the file.
- In assembler code, labels appearing at the beginning of a line, followed by a colon, are tags.
- In LaTeX text, the argument of any of the commands `\chapter`, `\section`, `\subsection`, `\subsubsection`, `\eqno`, `\label`, `\ref`, `\cite`, `\bibitem`, `\part`, `\appendix`, `\entry`, or `\index`, is a tag.

  Other commands can make tags as well, if you specify them in the environment variable `TEXTAGS` before invoking `etags`. The value of this environment variable should be a colon-separated list of commands names. For example,

  ```
  TEXTAGS="def:newcommand:newenvironment"
  export TEXTAGS
  ```

  specifies (using Bourne shell syntax) that the commands `\def', `\newcommand' and `\newenvironment' also define tags.
- You can also generate tags based on regexp matching (see section Creating Tags Tables) for any text file.

Creating Tags Tables

The `etags` program is used to create a tags table file. It knows the syntax of several languages, as described in the previous section. Here is how to run `etags`:

```
etags inputfiles...
```

The `etags` program reads the specified files, and writes a tags table named `TAGS' in the current working directory. `etags` recognizes the language used in an input file based on its file name and

contents. You can specify the language with the `--language=name' option, described below.

If the tags table data become outdated due to changes in the files described in the table, the way to update the tags table is the same way it was made in the first place. It is not necessary to do this often.

If the tags table fails to record a tag, or records it for the wrong file, then Emacs cannot possibly find its definition. However, if the position recorded in the tags table becomes a little bit wrong (due to some editing in the file that the tag definition is in), the only consequence is a slight delay in finding the tag. Even if the stored position is very wrong, Emacs will still find the tag, but it must search the entire file for it.

So you should update a tags table when you define new tags that you want to have listed, or when you move tag definitions from one file to another, or when changes become substantial. Normally there is no need to update the tags table after each edit, or even every day.

One tags table can effectively include another. Specify the included tags file name with the `--include=file' option when creating the file that is to include it. The latter file then acts as if it contained all the files specified in the included file, as well as the files it directly contains.

If you specify the source files with relative file names when you run `etags`, the tags file will contain file names relative to the directory where the tags file was initially written. This way, you can move an entire directory tree containing both the tags file and the source files, and the tags file will still refer correctly to the source files.

If you specify absolute file names as arguments to `etags`, then the tags file will contain absolute file names. This way, the tags file will still refer to the same files even if you move it, as long as the source files remain in the same place. Absolute file names start with `/', or with `device:/' on MS-DOS and Windows.

When you want to make a tags table from a great number of files, you may have problems listing them on the command line, because some systems have a limit on its length. The simplest way to circumvent this limit is to tell `etags` to read the file names from its standard input, by typing a dash in place of the file names, like this:

```
find . -name "*.[chCH]" -print | etags -
```

Use the option `--language=name' to specify the language explicitly. You can intermix these options with file names; each one applies to the file names that follow it. Specify `--language=auto' to tell `etags` to resume guessing the language from the file names and file contents. Specify `--language=none' to turn off language-specific processing entirely; then `etags` recognizes tags by regexp matching alone. `etags --help' prints the list of the languages `etags` knows, and the file name rules for guessing the language.

The `--regex' option provides a general way of recognizing tags based on regexp matching. You can freely intermix it with file names. Each `--regex' option adds to the preceding ones, and applies only to the following files. The syntax is:

```
--regex=/tagregexp[/nameregexp]/
```

where tagregexp is used to match the lines to tag. It is always anchored, that is, it behaves as if preceded

by `^'. If you want to account for indentation, just match any initial number of blanks by beginning y regular expression with `[\t]*'. In the regular expressions, `\' quotes the next character, and `\t' stands the tab character. Note that `etags` does not handle the other C escape sequences for special characters.

You should not match more characters with tagregexp than that needed to recognize what you want to tag. If the match is such that more characters than needed are unavoidably matched by tagregexp, you may find useful to add a nameregexp, in order to narrow the tag scope. You can find some examples below.

The `-R' option deletes all the regexps defined with `--regex' options. It applies to the file names following it, as you can see from the following example:

```
etags --regex=/reg1/ voo.doo --regex=/reg2/ \
    bar.ber -R --lang=lisp los.er
```

Here `etags` chooses the parsing language for `voo.doo' and `bar.ber' according to their contents. `etags` also uses reg1 to recognize additional tags in `voo.doo', and both reg1 and reg2 to recognize additional tags in `bar.ber'. `etags` uses the Lisp tags rules, and no regexp matching, to recognize tags in `los.er'.

Here are some more examples. The regexps are quoted to protect them from shell interpretation.

Tag the `DEFVAR` macros in the emacs source files:

```
--regex='/[ \t]*DEFVAR_[A-Z_ \t(]+"\([^"]+\)"/'
```

Tag VHDL files (this example is a single long line, broken here for formatting reasons):

```
--language=none
--regex='/[ \t]*\(ARCHITECTURE\|CONFIGURATION\) +[^ ]* +OF/'
--regex='/[ \t]*\(ATTRIBUTE\|ENTITY\|FUNCTION\|PACKAGE\
\( BODY\)?\|PROCEDURE\|PROCESS\|TYPE\)[ \t]+\([^ \t(]+\)/\3/'
```

Tag Cobol files (every label starting in column seven):

```
--language=none --regex='/.......[a-zA-Z0-9-]+\./'
```

Tag Postscript files (every label starting in column one):

```
--language=none --regex='#/[^ \t{]+#/'
```

Tag TCL files (this last example shows the usage of a nameregexp):

```
--lang=none --regex='/proc[ \t]+\([^ \t]+\)/\1/'
```

For a list of the other available `etags` options, execute `etags --help`.

Selecting a Tags Table

Emacs has at any time one selected tags table, and all the commands for working with tags tables use the selected one. To select a tags table, type M-x visit-tags-table, which reads the tags table file name as an argument. The name `TAGS' in the default directory is used as the default file name.

All this command does is store the file name in the variable `tags-file-name`. Emacs does not actually read in the tags table contents until you try to use them. Setting this variable yourself is just as good as using `visit-tags-table`. The variable's initial value is `nil`; that value tells all the commands for working with tags tables that they must ask for a tags table file name to use.

Using `visit-tags-table` when a tags table is already loaded gives you a choice: you can add the new tags table to the current list of tags tables, or start a new list. The tags commands use all the tags tables in the current list. If you start a new list, the new tags table is used *instead* of others. If you add the new table to the current list, it is used *as well as* the others. When the tags commands scan the list of tags tables, they don't always start at the beginning of the list; they start with the first tags table (if any) that describes the current file, proceed from there to the end of the list, and then scan from the beginning of the list until they have covered all the tables in the list.

You can specify a precise list of tags tables by setting the variable `tags-table-list` to a list of strings, like this:

```
(setq tags-table-list
      '("~/emacs" "/usr/local/lib/emacs/src"))
```

This tells the tags commands to look at the `TAGS' files in your `~/emacs' directory and in the `/usr/local/lib/emacs/src' directory. The order depends on which file you are in and which tags table mentions that file, as explained above.

Do not set both `tags-file-name` and `tags-table-list`.

Finding a Tag

The most important thing that a tags table enables you to do is to find the definition of a specific tag.

M-. tag RET
: Find first definition of tag (`find-tag`).

C-u M-.
: Find next alternate definition of last tag specified.

C-u - M-.
: Go back to previous tag found.

C-M-. pattern RET
: Find a tag whose name matches pattern (`find-tag-regexp`).

C-u C-M-.
: Find the next tag whose name matches the last pattern used.

C-x 4 . tag RET

Find first definition of tag, but display it in another window (`find-tag-other-window`).

C-x 5 . tag RET

Find first definition of tag, and create a new frame to select the buffer (`find-tag-other-frame`).

M-. (`find-tag`) is the command to find the definition of a specified tag. It searches through the tags table for that tag, as a string, and then uses the tags table info to determine the file that the definition is in and the approximate character position in the file of the definition. Then `find-tag` visits that file, moves point to the approximate character position, and searches ever-increasing distances away to find the tag definition.

If an empty argument is given (just type RET), the sexp in the buffer before or around point is used as the tag argument. See section Lists and Sexps, for info on sexps.

You don't need to give M-. the full name of the tag; a part will do. This is because M-. finds tags in the table which contain tag as a substring. However, it prefers an exact match to a substring match. To find other tags that match the same substring, give `find-tag` a numeric argument, as in C-u M-.; this does not read a tag name, but continues searching the tags table's text for another tag containing the same substring last used. If you have a real META key, M-0 M-. is an easier alternative to C-u M-..

Like most commands that can switch buffers, `find-tag` has a variant that displays the new buffer in another window, and one that makes a new frame for it. The former is C-x 4 ., which invokes the command `find-tag-other-window`. The latter is C-x 5 ., which invokes `find-tag-other-frame`.

To move back to places you've found tags recently, use C-u - M-.; more generally, M-. with a negative numeric argument. This command can take you to another buffer. C-x 4 . with a negative argument finds the previous tag location in another window.

The command C-M-. (`find-tag-regexp`) visits the tags that match a specified regular expression. It is just like M-. except that it does regexp matching instead of substring matching.

Searching and Replacing with Tags Tables

The commands in this section visit and search all the files listed in the selected tags table, one by one. For these commands, the tags table serves only to specify a sequence of files to search.

M-x tags-search RET regexp RET

Search for regexp through the files in the selected tags table.

M-x tags-query-replace RET regexp RET replacement RET

Perform a `query-replace-regexp` on each file in the selected tags table.

M-,

Restart one of the commands above, from the current location of point (`tags-loop-continue`).

M-x tags-search reads a regexp using the minibuffer, then searches for matches in all the files in the

selected tags table, one file at a time. It displays the name of the file being searched so you can follow progress. As soon as it finds an occurrence, `tags-search` returns.

Having found one match, you probably want to find all the rest. To find one more match, type M-, (`tags-loop-continue`) to resume the `tags-search`. This searches the rest of the current buffer, followed by the remaining files of the tags table.

M-x tags-query-replace performs a single `query-replace-regexp` through all the files in the tags table. It reads a regexp to search for and a string to replace with, just like ordinary M-x query-replace-regexp. It searches much like M-x tags-search, but repeatedly, processing matches according to your input. See section Replacement Commands, for more information on query replace.

It is possible to get through all the files in the tags table with a single invocation of M-x tags-query-replace. But often it is useful to exit temporarily, which you can do with any input event that has no special query replace meaning. You can resume the query replace subsequently by typing M-,; this command resumes the last tags search or replace command that you did.

The commands in this section carry out much broader searches than the `find-tag` family. The `find-tag` commands search only for definitions of tags that match your substring or regexp. The commands `tags-search` and `tags-query-replace` find every occurrence of the regexp, as ordinary search commands and replace commands do in the current buffer.

These commands create buffers only temporarily for the files that they have to search (those which are not already visited in Emacs buffers). Buffers in which no match is found are quickly killed; the others continue to exist.

It may have struck you that `tags-search` is a lot like `grep`. You can also run `grep` itself as an inferior of Emacs and have Emacs show you the matching lines one by one. This works much like running a compilation; finding the source locations of the `grep` matches works like finding the compilation errors. See section Running Compilations under Emacs.

Tags Table Inquiries

M-x list-tags RET file RET

Display a list of the tags defined in the program file `` `file' ``.

M-x tags-apropos RET regexp RET

Display a list of all tags matching regexp.

M-x list-tags reads the name of one of the files described by the selected tags table, and displays a list of all the tags defined in that file. The "file name" argument is really just a string to compare against the file names recorded in the tags table; it is read as a string rather than as a file name. Therefore, completion and defaulting are not available, and you must enter the file name the same way it appears in the tags table. Do not include a directory as part of the file name unless the file name recorded in the tags table includes a directory.

M-x tags-apropos is like `apropos` for tags (see section Apropos). It reads a regexp, then finds all the tags in the selected tags table whose entries match that regexp, and displays the tag names found.

You can also perform completion in the buffer on the name space of tag names in the current tags tat
See section Completion for Symbol Names.

Merging Files with Emerge

It's not unusual for programmers to get their signals crossed and modify the same program in two different directions. To recover from this confusion, you need to merge the two versions. Emerge makes this easier. See also section Comparing Files.

Overview of Emerge

To start Emerge, run one of these four commands:

M-x emerge-files

Merge two specified files.

M-x emerge-files-with-ancestor

Merge two specified files, with reference to a common ancestor.

M-x emerge-buffers

Merge two buffers.

M-x emerge-buffers-with-ancestor

Merge two buffers with reference to a common ancestor in a third buffer.

The Emerge commands compare two files or buffers, and display the comparison in three buffers: one for each input text (the A buffer and the B buffer), and one (the merge buffer) where merging takes place. The merge buffer shows the full merged text, not just the differences. Wherever the two input texts differ, you can choose which one of them to include in the merge buffer.

The Emerge commands that take input from existing buffers use only the accessible portions of those buffers, if they are narrowed (see section Narrowing).

If a common ancestor version is available, from which the two texts to be merged were both derived, Emerge can use it to guess which alternative is right. Wherever one current version agrees with the ancestor, Emerge presumes that the other current version is a deliberate change which should be kept in the merged version. Use the `with-ancestor' commands if you want to specify a common ancestor text. These commands read three file or buffer names--variant A, variant B, and the common ancestor.

After the comparison is done and the buffers are prepared, the interactive merging starts. You control the merging by typing special merge commands in the merge buffer. The merge buffer shows you a full merged text, not just differences. For each run of differences between the input texts, you can choose which one of them to keep, or edit them both together.

The merge buffer uses a special major mode, Emerge mode, with commands for making these choices. But you can also edit the buffer with ordinary Emacs commands.

At any given time, the attention of Emerge is focused on one particular difference, called the selected

difference. This difference is marked off in the three buffers like this:

```
vvvvvvvvvvvvvvvvvvvv
text that differs
^^^^^^^^^^^^^^^^^^^^
```

Emerge numbers all the differences sequentially and the mode line always shows the number of the selected difference.

Normally, the merge buffer starts out with the A version of the text. But when the A version of a difference agrees with the common ancestor, then the B version is initially preferred for that difference.

Emerge leaves the merged text in the merge buffer when you exit. At that point, you can save it in a file with C-x C-w. If you give a numeric argument to `emerge-files` or `emerge-files-with-ancestor`, it reads the name of the output file using the minibuffer. (This is the last file name those commands read.) Then exiting from Emerge saves the merged text in the output file.

Normally, Emerge commands save the output buffer in its file when you exit. If you abort Emerge with C-], the Emerge command does not save the output buffer, but you can save it yourself if you wish.

Submodes of Emerge

You can choose between two modes for giving merge commands: Fast mode and Edit mode. In Fast mode, basic merge commands are single characters, but ordinary Emacs commands are disabled. This is convenient if you use only merge commands. In Edit mode, all merge commands start with the prefix key C-c C-c, and the normal Emacs commands are also available. This allows editing the merge buffer, but slows down Emerge operations.

Use e to switch to Edit mode, and C-c C-c f to switch to Fast mode. The mode line indicates Edit and Fast modes with `E' and `F'.

Emerge has two additional submodes that affect how particular merge commands work: Auto Advance mode and Skip Prefers mode.

If Auto Advance mode is in effect, the a and b commands advance to the next difference. This lets you go through the merge faster as long as you simply choose one of the alternatives from the input. The mode line indicates Auto Advance mode with `A'.

If Skip Prefers mode is in effect, the n and p commands skip over differences in states prefer-A and prefer-B (see section State of a Difference). Thus you see only differences for which neither version is presumed "correct". The mode line indicates Skip Prefers mode with `S'.

Use the command s a (`emerge-auto-advance-mode`) to set or clear Auto Advance mode. Use s s (`emerge-skip-prefers-mode`) to set or clear Skip Prefers mode. These commands turn on the mode with a positive argument, turns it off with a negative or zero argument, and toggle the mode with no argument.

State of a Difference

In the merge buffer, a difference is marked with lines of `v' and `^' characters. Each difference has one of these seven states:

A

The difference is showing the A version. The a command always produces this state; the mode line indicates it with `A'.

B

The difference is showing the B version. The b command always produces this state; the mode line indicates it with `B'.

default-A

default-B

The difference is showing the A or the B state by default, because you haven't made a choice. All differences start in the default-A state (and thus the merge buffer is a copy of the A buffer), except those for which one alternative is "preferred" (see below).

When you select a difference, its state changes from default-A or default-B to plain A or B. Thus, the selected difference never has state default-A or default-B, and these states are never displayed in the mode line.

The command d a chooses default-A as the default state, and d b chooses default-B. This chosen default applies to all differences which you haven't ever selected and for which no alternative is preferred. If you are moving through the merge sequentially, the differences you haven't selected are those following the selected one. Thus, while moving sequentially, you can effectively make the A version the default for some sections of the merge buffer and the B version the default for others by using d a and d b between sections.

prefer-A

prefer-B

The difference is showing the A or B state because it is preferred. This means that you haven't made an explicit choice, but one alternative seems likely to be right because the other alternative agrees with the common ancestor. Thus, where the A buffer agrees with the common ancestor, the B version is preferred, because chances are it is the one that was actually changed.

These two states are displayed in the mode line as `A*' and `B*'.

combined

The difference is showing a combination of the A and B states, as a result of the x c or x C commands.

Once a difference is in this state, the a and b commands don't do anything to it unless you give them a numeric argument.

The mode line displays this state as `comb'.

Merge Commands

Here are the Merge commands for Fast mode; in Edit mode, precede them with C-c C-c:

p
: Select the previous difference.

n
: Select the next difference.

a
: Choose the A version of this difference.

b
: Choose the B version of this difference.

C-u n j
: Select difference number n.

.
: Select the difference containing point. You can use this command in the merge buffer or in the A or B buffer.

q
: Quit--finish the merge.

C-]
: Abort--exit merging and do not save the output.

f
: Go into Fast mode. (In Edit mode, this is actually C-c C-c f.)

e
: Go into Edit mode.

l
: Recenter (like C-l) all three windows.

-
: Specify part of a prefix numeric argument.

digit
: Also specify part of a prefix numeric argument.

d a
: Choose the A version as the default from here down in the merge buffer.

d b
: Choose the B version as the default from here down in the merge buffer.

c a
: Copy the A version of this difference into the kill ring.

c b

Copy the B version of this difference into the kill ring.

i a

Insert the A version of this difference at point.

i b

Insert the B version of this difference at point.

m

Put point and mark around the difference.

^

Scroll all three windows down (like M-v).

v

Scroll all three windows up (like C-v).

<

Scroll all three windows left (like C-x <).

>

Scroll all three windows right (like C-x >).

|

Reset horizontal scroll on all three windows.

x 1

Shrink the merge window to one line. (Use C-u l to restore it to full size.)

x c

Combine the two versions of this difference (see section Combining the Two Versions).

x f

Show the names of the files/buffers Emerge is operating on, in a Help window. (Use C-u l to restore windows.)

x j

Join this difference with the following one. (C-u x j joins this difference with the previous one.)

x s

Split this difference into two differences. Before you use this command, position point in each of the three buffers at the place where you want to split the difference.

x t

Trim identical lines off top and bottom of the difference. Such lines occur when the A and B versions are identical but differ from the ancestor version.

Exiting Emerge

The q command (`emerge-quit`) finishes the merge, storing the results into the output file if you specified one. It restores the A and B buffers to their proper contents, or kills them if they were created by Emerge and you haven't changed them. It also disables the Emerge commands in the merge buffer,

since executing them later could damage the contents of the various buffers.

C-] aborts the merge. This means exiting without writing the output file. If you didn't specify an output file, then there is no real difference between aborting and finishing the merge.

If the Emerge command was called from another Lisp program, then its return value is `t` for successful completion, or `nil` if you abort.

Combining the Two Versions

Sometimes you want to keep *both* alternatives for a particular difference. To do this, use x c, which edits the merge buffer like this:

```
#ifdef NEW
version from A buffer
#else /* NEW */
version from B buffer
#endif /* NEW */
```

While this example shows C preprocessor conditionals delimiting the two alternative versions, you can specify the strings to use by setting the variable `emerge-combine-versions-template` to a string of your choice. In the string, `%a' says where to put version A, and `%b' says where to put version B. The default setting, which produces the results shown above, looks like this:

```
"#ifdef NEW\n%a#else /* NEW */\n%b#endif /* NEW */\n"
```

Fine Points of Emerge

During the merge, you mustn't try to edit the A and B buffers yourself. Emerge modifies them temporarily, but ultimately puts them back the way they were.

You can have any number of merges going at once--just don't use any one buffer as input to more than one merge at once, since the temporary changes made in these buffers would get in each other's way.

Starting Emerge can take a long time because it needs to compare the files fully. Emacs can't do anything else until `diff` finishes. Perhaps in the future someone will change Emerge to do the comparison in the background when the input files are large--then you could keep on doing other things with Emacs until Emerge is ready to accept commands.

After setting up the merge, Emerge runs the hook `emerge-startup-hook` (see section Hooks).

C Mode

This section describes special features available in C, C++, Objective-C and Java modes.

C Mode Motion Commands

This section commands for moving point, in C mode and related modes.

`C-c C-u`

Move point back to the containing preprocessor conditional, leaving the mark behind. A prefix argument acts as a repeat count. With a negative argument, move point forward to the end of the containing preprocessor conditional. When going backwards, `#elif` is treated like `#else` followed by `#if`. When going forwards, `#elif` is ignored.

`C-c C-p`

Move point back over a preprocessor conditional, leaving the mark behind. A prefix argument acts as a repeat count. With a negative argument, move forward.

`C-c C-n`

Move point forward across a preprocessor conditional, leaving the mark behind. A prefix argument acts as a repeat count. With a negative argument, move backward.

`M-a`

Move point to the beginning of the innermost C statement. If point is already at the beginning of a statement, move to the beginning of the preceding statement. With prefix argument n, move back n - 1 statements.

If point is within a string or comment, or next to a comment (only whitespace between them), this command moves by sentences instead of statements.

When called from a program, this function takes two optional arguments: the numeric prefix argument, and a buffer position limit (don't move back before that place).

`M-e`

Move point to the end of the innermost C statement. If point is at the end of a statement, move to the end of the next statement. With prefix argument n, move forward n - 1 statements.

If point is within a string or comment, or next to a comment (only whitespace between them), this command moves by sentences instead of statements.

When called from a program, this function takes two optional arguments: the numeric prefix argument, and a buffer position limit (don't move past that place).

`M-x c-backward-into-nomenclature`

Move point backward to beginning of a C++ nomenclature section or word. With prefix argument n, move n times. If n is negative, move forward. C++ nomenclature means a symbol name in the style of NamingSymbolsWithMixedCaseAndNoUnderlines; each capital letter begins a section or word.

In the GNU project, we recommend using underscores to separate words within an identifier in C or C++, rather than using case distinctions.

`M-x c-forward-into-nomenclature`

Move point forward to end of a C++ nomenclature section or word. With prefix argument n, move n times.

Electric C Characters

In C mode and related modes, certain printing characters are "electric"---in addition to inserting themselves, they also reindent the current line and may insert newlines. This feature is controlled by the variable `c-auto-newline`. The "electric" characters are {, }, :, #, ;, ,, <, >, / and *.

Electric characters insert newlines only when the auto-newline feature is enabled (indicated by `/a' in the mode line after the mode name). This feature is controlled by the variable `c-auto-newline`. You can turn this feature on or off with the command C-c C-a:

C-c C-a

> Toggle the auto-newline feature (`c-toggle-auto-state`). With a prefix argument, this command turns the auto-newline feature on if the argument is positive, and off if it is negative.

The colon character is electric because that is appropriate for a single colon. But when you want to insert a double colon in C++, the electric behavior of colon is inconvenient. You can insert a double colon with no reindentation or newlines by typing C-c ::

C-c :

> Insert a double colon scope operator at point, without reindenting the line or adding any newlines (`c-scope-operator`).

The electric # key reindents the line if it appears to be the beginning of a preprocessor directive. This happens when the value of `c-electric-pound-behavior` is `(alignleft)`. You can turn this feature off by setting `c-electric-pound-behavior` to `nil`.

The variable `c-hanging-braces-alist` controls the insertion of newlines before and after inserted braces. It is an association list with elements of the following form: `(syntactic-symbol . nl-list)`. Most of the syntactic symbols that appear in `c-offsets-alist` are meaningful here as well.

The list nl-list may contain either of the symbols `before` or `after`, or both; or it may be `nil`. When a brace is inserted, the syntactic context it defines is looked up in `c-hanging-braces-alist`; if it is found, the nl-list is used to determine where newlines are inserted: either before the brace, after, or both. If not found, the default is to insert a newline both before and after braces.

The variable `c-hanging-colons-alist` controls the insertion of newlines before and after inserted colons. It is an association list with elements of the following form: `(syntactic-symbol . nl-list)`. The list nl-list may contain either of the symbols `before` or `after`, or both; or it may be `nil`.

When a colon is inserted, the syntactic symbol it defines is looked up in this list, and if found, the nl-list is used to determine where newlines are inserted: either before the brace, after, or both. If the syntactic symbol is not found in this list, no newlines are inserted.

Electric characters can also delete newlines automatically when the auto-newline feature is enabled. This feature makes auto-newline more acceptable, by deleting the newlines in the most common cases where you do not want them. Emacs can recognize several cases in which deleting a newline might be

desirable; by setting the variable `c-cleanup-list`, you can specify *which* of these cases that sho happen. The variable's value is a list of symbols, each describing one case for possible deletion of a newline. Here are the meaningful symbols, and their meanings:

`brace-else-brace`

Clean up `} else {' constructs by placing entire construct on a single line. The clean-up occurs when you type the `{' after the `else`, but only if there is nothing but white space between the braces and the `else`.

`empty-defun-braces`

Clean up empty defun braces by placing the braces on the same line. Clean-up occurs when you type the closing brace.

`defun-close-semi`

Clean up the semicolon after a `struct` or similar type declaration, by placing the semicolon on the same line as the closing brace. Clean-up occurs when you type the semicolon.

`list-close-comma`

Clean up commas following braces in array and aggregate initializers. Clean-up occurs when you type the comma.

`scope-operator`

Clean up double colons which may designate a C++ scope operator, by placing the colons together. Clean-up occurs when you type the second colon, but only when the two colons are separated by nothing but whitespace.

Hungry Delete Feature in C

When the hungry-delete feature is enabled (indicated by `/h' or `/ah' in the mode line after the mode name), a single DEL command deletes all preceding whitespace, not just one space. To turn this feature on or off, use C-c C-d:

C-c C-d

Toggle the hungry-delete feature (`c-toggle-hungry-state`). With a prefix argument, this command turns the hungry-delete feature on if the argument is positive, and off if it is negative.

C-c C-t

Toggle the auto-newline and hungry-delete features, both at once (`c-toggle-auto-hungry-state`).

The variable `c-hungry-delete-key` controls whether the hungry-delete feature is enabled.

Other Commands for C Mode

C-M-h

Put mark at the end of a function definition, and put point at the beginning (`c-mark-function`).

M-q

Fill a paragraph, handling C and C++ comments (`c-fill-paragraph`). If any part of the current line is a comment or within a comment, this command fills the comment or the paragraph of it that point is in, preserving the comment indentation and comment delimiters.

C-c C-e

Run the C preprocessor on the text in the region, and show the result, which includes the expansion of all the macro calls (`c-macro-expand`). The buffer text before the region is also included in preprocessing, for the sake of macros defined there, but the output from this part isn't shown.

When you are debugging C code that uses macros, sometimes it is hard to figure out precisely how the macros expand. With this command, you don't have to figure it out; you can see the expansions.

C-c C-\

Insert or align `\' characters at the ends of the lines of the region (`c-backslash-region`). This is useful after writing or editing a C macro definition.

If a line already ends in `\', this command adjusts the amount of whitespace before it. Otherwise, it inserts a new `\'. However, the last line in the region is treated specially; no `\' is inserted on that line, and any `\' there is deleted.

M-x cpp-highlight-buffer

Highlight parts of the text according to its preprocessor conditionals. This command displays another buffer named `*CPP Edit*', which serves as a graphic menu for selecting how to display particular kinds of conditionals and their contents. After changing various settings, click on `[A]pply these settings' (or go to that buffer and type a) to rehighlight the C mode buffer accordingly.

C-c C-s

Display the syntactic information about the current source line (`c-show-syntactic-information`). This is the information that directs how the line is indented.

Comments in C Modes

C mode and related modes use a number of variables for controlling comment format.

`c-block-comments-indent-p`

This variable specifies how to reindent block comments. The C modes support five styles of block comments:

```
style 1:    style 2 (GNU):   style 3:    style 4:    style 5:
/*          /* Blah         /*          /*          /*
   blah        blah.  */      * blah     ** blah     blah
   blah                       * blah     ** blah     blah
   */                         */         */          */
```

For the styles 1 through 4, `c-block-comments-indent-p` should be `nil` (the default). If

you want to use style 5, set `c-block-comments-indent-p` to `t`.

This variable has no effect on the indentation of the comment-start itself or on insertion of asterisks when auto-filling C comments. It does not affect M-q either.

`c-comment-only-line-offset`

Extra offset for line which contains only the start of a comment. It can be either an integer or a cons cell of the form `(non-anchored-offset . anchored-offset)`, where non-anchored-offset is the amount of offset given to non-column-zero anchored comment-only lines, and anchored-offset is the amount of offset to give column-zero anchored comment-only lines. Just an integer as value is equivalent to `(val . 0)`.

`c-comment-start-regexp`

This buffer-local variable specifies how to recognize the start of a comment.

`c-hanging-comment-ender-p`

If this variable is `nil`, `c-fill-paragraph` leaves the comment terminator of a block comment on a line by itself. The default value is `t`, which always puts the comment-end delimiter `*/' at the end of the last line of the comment text.

Fortran Mode

Fortran mode provides special motion commands for Fortran statements and subprograms, and indentation commands that understand Fortran conventions of nesting, line numbers and continuation statements. Fortran mode has its own Auto Fill mode that breaks long lines into proper Fortran continuation lines.

Special commands for comments are provided because Fortran comments are unlike those of other languages. Built-in abbrevs optionally save typing when you insert Fortran keywords.

Use M-x fortran-mode to switch to this major mode. This command runs the hook `fortran-mode-hook` (see section Hooks).

Motion Commands

Fortran mode provides special commands to move by subprograms (functions and subroutines) and by statements. There is also a command to put the region around one subprogram, convenient for killing it or moving it.

C-M-a

Move to beginning of subprogram (`beginning-of-fortran-subprogram`).

C-M-e

Move to end of subprogram (`end-of-fortran-subprogram`).

C-M-h

Put point at beginning of subprogram and mark at end (`mark-fortran-subprogram`).

C-c C-n

Move to beginning of current or next statement (`fortran-next-statement`).

C-c C-p

Move to beginning of current or previous statement (`fortran-previous-statement`).

Fortran Indentation

Special commands and features are needed for indenting Fortran code in order to make sure various syntactic entities (line numbers, comment line indicators and continuation line flags) appear in the columns that are required for standard Fortran.

Fortran Indentation Commands

TAB

Indent the current line (`fortran-indent-line`).

LFD

Indent the current and start a new indented line (`fortran-indent-new-line`).

M-LFD

Break the current line and set up a continuation line.

C-M-q

Indent all the lines of the subprogram point is in (`fortran-indent-subprogram`).

Fortran mode redefines TAB to reindent the current line for Fortran (`fortran-indent-line`). This command indents Line numbers and continuation markers to their required columns, and independently indents the body of the statement based on its nesting in the program.

The key LFD runs the command `fortran-indent-new-line`, which reindents the current line then makes and indents a new line. This command is useful to reindent the closing statement of `do' loops and other blocks before starting a new line.

The key C-M-q runs `fortran-indent-subprogram`, a command to reindent all the lines of the Fortran subprogram (function or subroutine) containing point.

The key M-LFD runs `fortran-split-line`, which splits a line in the appropriate fashion for Fortran. In a non-comment line, the second half becomes a continuation line and is indented accordingly. In a comment line, both halves become separate comment lines.

Continuation Lines

Most modern Fortran compilers allow two ways of writing continuation lines. If the first non-space character on a line is in column 5, then that line is a continuation of the previous line. We call this fixed format. (In GNU Emacs we always count columns from 0.) The variable `fortran-continuation-string` specifies what character to put on column 5. A line that starts with a tab character followed by any digit except `0' is also a continuation line. We call this style of continuation tab format.

Fortran mode can make either style of continuation line, but you must specify which one you prefer. The value of the variable `indent-tabs-mode` controls the choice: `nil` for fixed format, and non-`nil` for tab format. You can tell which style is presently in effect by the presence or absence of the string `Tab' in the mode line.

If the text on a line starts with the conventional Fortran continuation marker `$', or if it begins with any non-whitespace character in column 5, Fortran mode treats it as a continuation line. When you indent a continuation line with TAB, it converts the line to the current continuation style. When you split a Fortran statement with M-LFD, the continuation marker on the newline is created according to the continuation style.

The setting of continuation style affects several other aspects of editing in Fortran mode. In fixed format mode, the minimum column number for the body of a statement is 6. Lines inside of Fortran blocks that are indented to larger column numbers always use only the space character for whitespace. In tab format mode, the minimum column number for the statement body is 8, and the whitespace before column 8 must always consist of one tab character.

When you enter Fortran mode for an existing file, it tries to deduce the proper continuation style automatically from the file contents. The first line that begins with either a tab character or six spaces determines the choice. The variable `fortran-analyze-depth` specifies how many lines to consider (at the beginning of the file); if none of those lines indicates a style, then the variable `fortran-tab-mode-default` specifies the style. If it is `nil`, that specifies fixed format, and non-`nil` specifies tab format.

Line Numbers

If a number is the first non-whitespace in the line, Fortran indentation assumes it is a line number and moves it to columns 0 through 4. (Columns always count from 0 in GNU Emacs.)

Line numbers of four digits or less are normally indented one space. The variable `fortran-line-number-indent` controls this; it specifies the maximum indentation a line number can have. Line numbers are indented to right-justify them to end in column 4 unless that would require more than this maximum indentation. The default value of the variable is 1.

Simply inserting a line number is enough to indent it according to these rules. As each digit is inserted, the indentation is recomputed. To turn off this feature, set the variable `fortran-electric-line-number` to `nil`. Then inserting line numbers is like inserting anything else.

Syntactic Conventions

Fortran mode assumes that you follow certain conventions that simplify the task of understanding a Fortran program well enough to indent it properly:

- Two nested `do' loops never share a `continue' statement.
- Fortran keywords such as `if', `else', `then', `do' and others are written without embedded whitespace or line breaks.

Fortran compilers generally ignore whitespace outside of string constants, but Fortran mode d not recognize these keywords if they are not contiguous. Constructs such as `else if' or `end d acceptable, but the second word should be on the same line as the first and not on a continuati line.

If you fail to follow these conventions, the indentation commands may indent some lines unaesthetically. However, a correct Fortran program retains its meaning when reindented even if the conventions are not followed.

Variables for Fortran Indentation

Several additional variables control how Fortran indentation works:

`fortran-do-indent`

Extra indentation within each level of `do' statement (default 3).

`fortran-if-indent`

Extra indentation within each level of `if' statement (default 3). This value is also used for extra indentation within each level of the Fortran 90 `where' statement.

`fortran-structure-indent`

Extra indentation within each level of `structure', `union', or `map' statements (default 3).

`fortran-continuation-indent`

Extra indentation for bodies of continuation lines (default 5).

`fortran-check-all-num-for-matching-do`

If this is `nil`, indentation assumes that each `do' statement ends on a `continue' statement. Therefore, when computing indentation for a statement other than `continue', it can save time by not checking for a `do' statement ending there. If this is non-`nil`, indenting any numbered statement must check for a `do' that ends there. The default is `nil`.

`fortran-blink-matching-if`

If this is `t`, indenting an `endif' statement moves the cursor momentarily to the matching `if' statement to show where it is. The default is `nil`.

`fortran-minimum-statement-indent-fixed`

Minimum indentation for fortran statements when using fixed format continuation line style. Statement bodies are never indented less than this much. The default is 6.

`fortran-minimum-statement-indent-tab`

Minimum indentation for fortran statements for tab format continuation line style. Statement bodies are never indented less than this much. The default is 8.

Fortran Comments

The usual Emacs comment commands assume that a comment can follow a line of code. In Fortran, the standard comment syntax requires an entire line to be just a comment. Therefore, Fortran mode replaces the standard Emacs comment commands and defines some new variables.

Fortran mode can also handle a nonstandard comment syntax where comments start with `!' and can follow other text. Because only some Fortran compilers accept this syntax, Fortran mode will not insert such comments unless you have said in advance to do so. To do this, set the variable `comment-start` to `"!"` (see section Variables).

M-;

Align comment or insert new comment (`fortran-comment-indent`).

C-x ;

Applies to nonstandard `!' comments only.

C-c ;

Turn all lines of the region into comments, or (with argument) turn them back into real code (`fortran-comment-region`).

M-; in Fortran mode is redefined as the command `fortran-comment-indent`. Like the usual M-; command, this recognizes any kind of existing comment and aligns its text appropriately; if there is no existing comment, a comment is inserted and aligned. But inserting and aligning comments are not the same in Fortran mode as in other modes.

When a new comment must be inserted, if the current line is blank, a full-line comment is inserted. On a non-blank line, a nonstandard `!' comment is inserted if you have said you want to use them. Otherwise a full-line comment is inserted on a new line before the current line.

Nonstandard `!' comments are aligned like comments in other languages, but full-line comments are different. In a standard full-line comment, the comment delimiter itself must always appear in column zero. What can be aligned is the text within the comment. You can choose from three styles of alignment by setting the variable `fortran-comment-indent-style` to one of these values:

`fixed`

Align the text at a fixed column, which is the sum of `fortran-comment-line-extra-indent` and the minimum statement indentation. This is the default.

The minimum statement indentation is `fortran-minimum-statement-indent-fixed` for fixed format continuation line style and `fortran-minimum-statement-indent-tab` for tab format style.

`relative`

Align the text as if it were a line of code, but with an additional `fortran-comment-line-extra-indent` columns of indentation.

`nil`

Don't move text in full-line columns automatically at all.

In addition, you can specify the character to be used to indent within full-line comments by setting the variable `fortran-comment-indent-char` to the single-character string you want to use.

Fortran mode introduces two variables `comment-line-start` and `comment-line-start-skip` which play for full-line comments the same roles played by `comment-start` and `comment-start-skip` for ordinary text-following comments. Normally these are set properly by

Fortran mode so you do not need to change them.

The normal Emacs comment command C-x ; has not been redefined. If you use `!' comments, this command can be used with them. Otherwise it is useless in Fortran mode.

The command C-c ; (`fortran-comment-region`) turns all the lines of the region into comments by inserting the string `C$$$' at the front of each one. With a numeric argument, it turns the region back into live code by deleting `C$$$' from the front of each line in it. The string used for these comments can be controlled by setting the variable `fortran-comment-region`. Note that here we have an example of a command and a variable with the same name; these two uses of the name never conflict because in Lisp and in Emacs it is always clear from the context which one is meant.

Fortran Auto Fill Mode

Fortran Auto Fill mode is a minor mode which automatically splits Fortran statements as you insert them when they become too wide. Splitting a statement involves making continuation lines using `fortran-continuation-string` (See section Continuation Lines). This splitting happens when you type SPC, RET, or TAB, and also in the Fortran indentation commands.

M-x fortran-auto-fill-mode turns Fortran Auto Fill mode on if it was off, or off if it was on. This command works the same as M-x auto-fill-mode does for normal Auto Fill mode (see section Filling Text). A positive numeric argument turns Fortran Auto Fill mode on, and a negative argument turns it off. You can see when Fortran Auto Fill mode is in effect by the presence of the word `Fill' in the mode line, inside the parentheses. Fortran Auto Fill mode is a minor mode, turned on or off for each buffer individually. See section Minor Modes.

Fortran Auto Fill mode breaks lines at spaces or delimiters when the lines get longer than the desired width (the value of `fill-column`). The delimiters that Fortran Auto Fill mode may break at are `,', `'', `+', `-', `/', `*', `=', and `)'. The line break comes after the delimiter if the variable `fortran-break-before-delimiters` is `nil`. Otherwise (and by default), the break comes before the delimiter.

By default, Fortran Auto Fill mode is not enabled. If you want this feature turned on permanently, add a hook function to `fortran-mode-hook` to execute `(fortran-auto-fill-mode 1)`. See section Hooks.

Checking Columns in Fortran

C-c C-r

Display a "column ruler" momentarily above the current line (`fortran-column-ruler`).

C-c C-w

Split the current window horizontally temporarily so that it is 72 columns wide. This may help you avoid making lines longer than the 72 character limit that some fortran compilers impose (`fortran-window-create-momentarily`).

The command C-c C-r (`fortran-column-ruler`) shows a column ruler momentarily above the

current line. The comment ruler is two lines of text that show you the locations of columns with spec significance in Fortran programs. Square brackets show the limits of the columns for line numbers, a curly brackets show the limits of the columns for the statement body. Column numbers appear above them.

Note that the column numbers count from zero, as always in GNU Emacs. As a result, the numbers may be one less than those you are familiar with; but the positions they indicate in the line are standard for Fortran.

The text used to display the column ruler depends on the value of the variable `indent-tabs-mode`. If `indent-tabs-mode` is `nil`, then the value of the variable `fortran-column-ruler-fixed` is used as the column ruler. Otherwise, the variable `fortran-column-ruler-tab` is displayed. By changing these variables, you can change the column ruler display.

For even more help, use C-c C-w (`fortran-window-create`), a command which splits the current window horizontally, making a window 72 columns wide. By editing in this window you can immediately see when you make a line too wide to be correct Fortran.

Fortran Keyword Abbrevs

Fortran mode provides many built-in abbrevs for common keywords and declarations. These are the same sort of abbrev that you can define yourself. To use them, you must turn on Abbrev mode. See section Abbrevs.

The built-in abbrevs are unusual in one way: they all start with a semicolon. You cannot normally use semicolon in an abbrev, but Fortran mode makes this possible by changing the syntax of semicolon to "word constituent."

For example, one built-in Fortran abbrev is `;c' for `continue'. If you insert `;c' and then insert a punctuation character such as a space or a newline, the `;c' expands automatically to `continue', provided Abbrev mode is enabled.

Type `;?' or `;C-h' to display a list of all the built-in Fortran abbrevs and what they stand for.

Asm Mode

Asm mode is a major mode for editing files of assembler code. It defines these commands:

TAB
: `tab-to-tab-stop`.

LFD
: Insert a newline and then indent using `tab-to-tab-stop`.

:
: Insert a colon and then remove the indentation from before the label preceding colon. Then do `tab-to-tab-stop`.

;

Insert or align a comment.

The variable `asm-comment-char` specifies which character starts comments in assembler syntax.

Go to the previous, next section.

Go to the previous, next section.

Compiling and Testing Programs

The previous chapter discusses the Emacs commands that are useful for making changes in programs. This chapter deals with commands that assist in the larger process of developing and maintaining programs.

Running Compilations under Emacs

Emacs can run compilers for noninteractive languages such as C and Fortran as inferior processes, feeding the error log into an Emacs buffer. It can also parse the error messages and show you the source lines where compilation errors occurred.

M-x compile

: Run a compiler asynchronously under Emacs, with error messages to `*compilation*' buffer.

M-x grep

: Run `grep` asynchronously under Emacs, with matching lines listed in the buffer named `*grep*'.

M-x kill-compilation

M-x kill-grep

: Kill the running compilation or `grep` subprocess.

C-x `

: Visit the locus of the next compiler error message or `grep` match.

RET

: Visit the locus of the error message that point is on. This command is used in the compilation buffer.

Mouse-2

: Visit the locus of the error message that you click on.

To run `make` or another compilation command, do M-x compile. This command reads a shell command line using the minibuffer, and then executes the command in an inferior shell, putting output in the buffer named `*compilation*'. The current buffer's default directory is used as the working directory for the execution of the command; normally, therefore, the compilation happens in this directory.

When the shell command line is read, the minibuffer appears containing a default command line, which is the command you used the last time you did M-x compile. If you type just RET, the same command line is used again. For the first M-x compile, the default is `make -k'. The default compilation command comes from the variable `compile-command`; if the appropriate compilation command for a file is something other than `make -k', it can be useful for the file to specify a local value for `compile-command` (see section Local Variables in Files).

Starting a compilation displays the buffer `*compilation*' in another window but does not select it. The

buffer's mode line tells you whether compilation is finished, with the word `run' or `exit' inside the parentheses. You do not have to keep this buffer visible; compilation continues in any case. While a compilation is going on, the string `Compiling' appears in the mode lines of all windows. When this string disappears, the compilation is finished.

If you want to watch the compilation transcript as it appears, switch to the `*compilation*' buffer and move point to the end of the buffer. When point is at the end, new compilation output is inserted above point, which remains at the end. If point is not at the end of the buffer, it remains fixed while more compilation output is added at the end of the buffer.

To kill the compilation process, do M-x kill-compilation. When the compiler process terminates, the mode line of the `*compilation*' buffer changes to say `signal' instead of `run'. Starting a new compilation also kills any running compilation, as only one can exist at any time. However, M-x compile asks for confirmation before actually killing a compilation that is running.

The `*compilation*' buffer uses a special major mode, Compilation mode. This mode provides the keys SPC and DEL to scroll by screenfuls, and M-n and M-p to move to the next or previous error message. You can also use M-{ and M-} to move up or down to an error message for a different source file.

You can visit the source for any particular error message by moving point in `*compilation*' to that error message and typing RET (`compile-goto-error`). Or click Mouse-2 on the error message; you need not switch to the `*compilation*' buffer first.

To parse the compiler error messages sequentially, type C-x ` (`next-error`). The character following the C-x is the backquote or "grave accent," not the single-quote. This command is available in all buffers, not just in `*compilation*'; it displays the next error message at the top of one window and source location of the error in another window.

The first time C-x ` is used after the start of a compilation, it moves to the first error's location. Subsequent uses of C-x ` advance down to subsequent errors. If you visit a specific error message with RET or Mouse-2, subsequent C-x ` commands advance from there. When C-x ` gets to the end of the buffer and finds no more error messages to visit, it fails and signals an Emacs error.

C-u C-x ` starts scanning from the beginning of the compilation buffer. This way, you can process the same set of errors again.

Just as you can run a compiler, you can also run `grep` and then visit the lines on which matches were found. To do this, type M-x grep with an argument line that contains the same arguments you would give `grep` when running it normally: a `grep`-style regexp (usually in single-quotes to quote the shell's special characters) followed by file names which may use wildcards. The output from `grep` goes in the `*grep*' buffer, and you can find the matching lines in the source with C-x ` and RET just like compiler errors.

Note: a shell is used to run the compile command, but the shell is told that it should be noninteractive. This means in particular that the shell starts up with no prompt. If you find your usual shell prompt making an unsightly appearance in the `*compilation*' buffer, it means you have made a mistake in your shell's init file by setting the prompt unconditionally. (The init file name may be `.profile', `.cshrc', `.shrc', or various other things, depending on the shell you use.) The shell init file should set the prompt only if there already is a prompt. In csh, here is how to do it:


```
if ($?prompt) set prompt = ...
```

And here's how to do it in bash:

```
if [ "${PS1+set}" = set ]
then prompt=...
fi
```

There may well be other things that your shell's init file ought to do only for an interactive shell. You can use the same method to conditionalize them.

The features of Compilation mode are also available in a minor mode called Compilation Minor mode. This lets you parse error messages in any buffer, not just a normal compilation output buffer. Type M-x compilation-minor-mode to enable the minor mode. This defines the keys RET and Mouse-2, as in the Compilation major mode. In an Rlogin buffer (see section Remote Host Shell), Compilation minor mode automatically accesses remote source files by FTP (see section File Names).

The MS-DOS "operating system" does not support asynchronous subprocesses; to work around this lack, M-x compile runs the compilation command synchronously on MS-DOS. As a consequence, you must wait until the command finishes before you can do anything else in Emacs. See section MS-DOS Issues.

Running Debuggers Under Emacs

The GUD (Grand Unified Debugger) library provides an interface to various symbolic debuggers from within Emacs. We recommend the debugger GDB, which is free software, but you can also run DBX, SDB or XDB if you have them. GUD can also serve as an interface to the Perl's debugging mode.

Starting GUD

There are five commands for starting a debugger, each corresponding to a particular debugger program.

M-x gdb RET file RET

> Run GDB as a subprocess of Emacs. This command creates a buffer for input and output to GDB, and switches to it. If a GDB buffer already exists, it just switches to that buffer.

M-x dbx RET file RET

> Similar, but run DBX instead of GDB.

M-x xdb RET file RET

> Similar, but run XDB instead of GDB. Use the variable `gud-xdb-directories` to specify directories to search for source files.

M-x sdb RET file RET

> Similar, but run SDB instead of GDB.
>
> Some versions of SDB do not mention source file names in their messages. When you use them, you need to have a valid tags table (see section Tags Tables) in order for GUD to find functions in

the source code. If you have not visited a tags table or the tags table doesn't list one of the functions, you get a message saying `The sdb support requires a valid tags table to work'. If this happens, generate a valid tags table in the working directory and try again.

M-x perldb RET file RET

Run the Perl interpreter in debug mode to debug file, a Perl program.

You can only run one debugger process at a time.

Each of these commands takes one argument: a command line to invoke the debugger. In the simplest case, specify just the name of the executable file you want to debug. You may also use options that the debugger supports. However, shell wild cards and variables are not allowed. GUD assumes that the first argument not preceded by a `-' is the executable file name.

Debugger Operation

When you run a debugger with GUD, the debugger uses an Emacs buffer for its ordinary input and output. This is called the GUD buffer. The debugger displays the source files of the program by visiting them in Emacs buffers. An arrow (`=>') in one of these buffers indicates the current execution line. Moving point in this buffer does not move the arrow.

You can start editing these source files at any time in the buffers that were made to display them. The arrow is not part of the file's text; it appears only on the screen. If you do modify a source file, keep in mind that inserting or deleting lines will throw off the arrow's positioning; GUD has no way of figuring out which line corresponded before your changes to the line number in a debugger message. Also, you'll typically have to recompile and restart the program for your changes to be reflected in the debugger's tables.

If you wish, you can control your debugger process entirely through the debugger buffer, which uses a variant of Shell mode. All the usual commands for your debugger are available, and you can use the Shell mode history commands to repeat them. See section Shell Mode.

Commands of GUD

The GUD interaction buffer uses a variant of Shell mode, so the commands of Shell mode are available (see section Shell Mode). GUD mode also provides commands for setting and clearing breakpoints, for selecting stack frames, and for stepping through the program. These commands are available both in the GUD buffer and globally, but with different key bindings.

The breakpoint commands are usually used in source file buffers, because that is the way to specify where to set or clear the breakpoint. Here's the global command to set a breakpoint:

C-x SPC

Set a breakpoint on the source line that point is on.

Here are the other special commands provided by GUD. The keys starting with C-c are available only in the GUD interaction buffer. The bindings that start with C-x C-a are available in the GUD buffer and also in source files.

C-c C-l

C-x C-a C-l

Display in another window the last line referred to in the GUD buffer (that is, the line indicated in the last location message). This runs the command `gud-refresh`.

C-c C-s

C-x C-a C-s

Execute a single line of code (`gud-step`). If the line contains a function call, execution stops after entering the called function.

C-c C-n

C-x C-a C-n

Execute a single line of code, stepping across entire function calls at full speed (`gud-next`).

C-c C-i

C-x C-a C-i

Execute a single machine instruction (`gud-stepi`).

C-c C-r

C-x C-a C-r

Continue execution without specifying any stopping point. The program will run until it hits a breakpoint, terminates, or gets a signal that the debugger is checking for (`gud-cont`).

C-c C-d

C-x C-a C-d

Delete the breakpoint(s) on the current source line, if any (`gud-remove`). If you use this command in the GUD interaction buffer, it applies to the line where the program last stopped.

C-c C-t

C-x C-a C-t

Set a temporary breakpoint on the current source line, if any. If you use this command in the GUD interaction buffer, it applies to the line where the program last stopped.

The above commands are common to all supported debuggers. If you are using GDB or (some versions of) DBX, these additional commands are available:

C-c <

C-x C-a <

Select the next enclosing stack frame (`gud-up`). This is equivalent to the `up' command.

C-c >

C-x C-a >

Select the next inner stack frame (`gud-down`). This is equivalent to the `down' command.

If you are using GDB, these additional key bindings are available:

TAB

With GDB, complete a symbol name (`gud-gdb-complete-command`). This key is available

only in the GUD interaction buffer, and requires GDB versions 4.13 and later.

C-c C-f

C-x C-a C-f

Run the program until the selected stack frame returns (or until it stops for some other reason).

These commands interpret a numeric argument as a repeat count, when that makes sense.

Because TAB serves as a completion command, you can't use it to enter a tab as input to the program you are debugging with GDB. Instead, type C-q TAB to enter a tab.

GUD Customization

On startup, GUD runs one of the following hooks: `gdb-mode-hook`, if you are using GDB; `dbx-mode-hook`, if you are using DBX; `sdb-mode-hook`, if you are using SDB; `xdb-mode-hook`, if you are using XDB; `perldb-mode-hook`, for Perl debugging mode. You can use these hooks to define custom key bindings for the debugger interaction buffer. See section Hooks.

Here is a convenient way to define a command that sends a particular command string to the debugger, and set up a key binding for it in the debugger interaction buffer:

```
(gud-def function cmdstring binding docstring)
```

This defines a command named function which sends cmdstring to the debugger process, and gives it the documentation string docstring. You can use the command thus defined in any buffer. If binding is non-`nil`, `gud-def` also binds the command to C-c binding in the GUD buffer's mode and to C-x C-a binding generally.

The command string cmdstring may contain certain `%'-sequences that stand for data to be filled in at the time function is called:

`%f'

The name of the current source file. If the current buffer is the GUD buffer, then the "current source file" is the file that the program stopped in.

`%l'

The number of the current source line. If the current buffer is the GUD buffer, then the "current source line" is the line that the program stopped in.

`%e'

The text of the C lvalue or function-call expression at or adjacent to point.

`%a'

The text of the hexadecimal address at or adjacent to point.

`%p'

The numeric argument of the called function, as a decimal number. If the command is used without a numeric argument, `%p' stands for the empty string.

If you don't use `%p' in the command string, the command you define ignores any numeric

argument.

Executing Lisp Expressions

Emacs has several different major modes for Lisp and Scheme. They are the same in terms of editing commands, but differ in the commands for executing Lisp expressions. Each mode has its own purpose.

Emacs-Lisp mode

The mode for editing source files of programs to run in Emacs Lisp. This mode defines C-M-x to evaluate the current defun. See section Libraries of Lisp Code for Emacs.

Lisp Interaction mode

The mode for an interactive session with Emacs Lisp. It defines LFD to evaluate the sexp before point and insert its value in the buffer. See section Lisp Interaction Buffers.

Lisp mode

The mode for editing source files of programs that run in Lisps other than Emacs Lisp. This mode defines C-M-x to send the current defun to an inferior Lisp process. See section Running an External Lisp.

Inferior Lisp mode

The mode for an interactive session with an inferior Lisp process. This mode combines the special features of Lisp mode and Shell mode (see section Shell Mode).

Scheme mode

Like Lisp mode but for Scheme programs.

Inferior Scheme mode

The mode for an interactive session with an inferior Scheme process.

Most editing commands for working with Lisp programs are in fact available globally. See section Editing Programs.

Libraries of Lisp Code for Emacs

Lisp code for Emacs editing commands is stored in files whose names conventionally end in `.el'. This ending tells Emacs to edit them in Emacs-Lisp mode (see section Executing Lisp Expressions).

To execute a file of Emacs Lisp code, use M-x load-file. This command reads a file name using the minibuffer and then executes the contents of that file as Lisp code. It is not necessary to visit the file first; in any case, this command reads the file as found on disk, not text in an Emacs buffer.

Once a file of Lisp code is installed in the Emacs Lisp library directories, users can load it using M-x load-library. Programs can load it by calling `load-library`, or with `load`, a more primitive function that is similar but accepts some additional arguments.

M-x load-library differs from M-x load-file in that it searches a sequence of directories and tries three file names in each directory. Suppose your argument is lib; the three names are `lib.elc', `lib.el',

and lastly just `lib'. If `lib.elc' exists, it is by convention the result of compiling `lib.el'; it is better to load the compiled file, since it will load and run faster.

If `load-library' finds that `lib.el' is newer than `lib.elc' file, it prints a warning, because it's likely that somebody made changes to the `.el' file and forgot to recompile it.

Because the argument to `load-library' is usually not in itself a valid file name, file name completion is not available. Indeed, when using this command, you usually do not know exactly what file name will be used.

The sequence of directories searched by M-x load-library is specified by the variable `load-path', a list of strings that are directory names. The default value of the list contains the directory where the Lisp code for Emacs itself is stored. If you have libraries of your own, put them in a single directory and add that directory to `load-path'. `nil' in this list stands for the current default directory, but it is probably not a good idea to put `nil' in the list. If you find yourself wishing that `nil' were in the list, most likely what you really want to do is use M-x load-file this once.

Often you do not have to give any command to load a library, because the commands defined in the library are set up to autoload that library. Trying to run any of those commands calls `load' to load the library; this replaces the autoload definitions with the real ones from the library.

Emacs Lisp code can be compiled into byte-code which loads faster, takes up less space when loaded, and executes faster. See section 'Byte Compilation' in the Emacs Lisp Reference Manual. By convention, the compiled code for a library goes in a separate file whose name consists of the library source file with `c' appended. Thus, the compiled code for `foo.el' goes in `foo.elc'. That's why `load-library' searches for `.elc' files first.

Evaluating Emacs-Lisp Expressions

Lisp programs intended to be run in Emacs should be edited in Emacs-Lisp mode; this happens automatically for file names ending in `.el'. By contrast, Lisp mode itself is used for editing Lisp programs intended for other Lisp systems. To switch to Emacs-Lisp mode explicitly, use the command M-x emacs-lisp-mode.

For testing of Lisp programs to run in Emacs, it is often useful to evaluate part of the program as it is found in the Emacs buffer. For example, after changing the text of a Lisp function definition, evaluating the definition installs the change for future calls to the function. Evaluation of Lisp expressions is also useful in any kind of editing, for invoking noninteractive functions (functions that are not commands).

M-:

> Read a single Lisp expression in the minibuffer, evaluate it, and print the value in the echo area (`eval-expression').

C-x C-e

> Evaluate the Lisp expression before point, and print the value in the echo area (`eval-last-sexp').

C-M-x

Evaluate the defun containing or after point, and print the value in the echo area (`eval-defu`

M-x eval-region

Evaluate all the Lisp expressions in the region.

M-x eval-current-buffer

Evaluate all the Lisp expressions in the buffer.

M-: (`eval-expression`) is the most basic command for evaluating a Lisp expression interactively. It reads the expression using the minibuffer, so you can execute any expression on a buffer regardless of what the buffer contains. When the expression is evaluated, the current buffer is once again the buffer that was current when M-: was typed.

M-: can easily confuse users who do not understand it. Therefore, `eval-expression` is normally a disabled command. Attempting to use this command asks for confirmation and gives you the option of enabling it; once you enable the command, confirmation will no longer be required for it. See section Disabling Commands.

In Emacs-Lisp mode, the key C-M-x is bound to the command `eval-defun`, which parses the defun containing or following point as a Lisp expression and evaluates it. The value is printed in the echo area. This command is convenient for installing in the Lisp environment changes that you have just made in the text of a function definition.

C-M-x treats `defvar` expressions specially. Normally, evaluating a `defvar` expression does nothing if the variable it defines already has a value. But C-M-x unconditionally resets the variable to the initial value specified in the `defvar` expression. This special feature is convenient for debugging Lisp programs.

The command C-x C-e (`eval-last-sexp`) evaluates the Lisp expression preceding point in the buffer, and displays the value in the echo area. It is available in all major modes, not just Emacs-Lisp mode. It does not treat `defvar` specially.

If C-M-x or C-x C-e is given a numeric argument, it inserts the value into the current buffer at point, rather than displaying it in the echo area. The argument's value does not matter.

The most general command for evaluating Lisp expressions from a buffer is `eval-region`. M-x eval-region parses the text of the region as one or more Lisp expressions, evaluating them one by one. M-x eval-current-buffer is similar but evaluates the entire buffer. This is a reasonable way to install the contents of a file of Lisp code that you are just ready to test. Later, as you find bugs and change individual functions, use C-M-x on each function that you change. This keeps the Lisp world in step with the source file.

Lisp Interaction Buffers

The buffer `*scratch*' which is selected when Emacs starts up is provided for evaluating Lisp expressions interactively inside Emacs.

The simplest way to use the `*scratch*' buffer is to insert Lisp expressions and type LFD after each expression. This command reads the Lisp expression before point, evaluates it, and inserts the value in

printed representation before point. The result is a complete typescript of the expressions you have evaluated and their values.

The `*scratch*' buffer's major mode is Lisp Interaction mode, which is the same as Emacs-Lisp mode except for the binding of LFD.

The rationale for this feature is that Emacs must have a buffer when it starts up, but that buffer is not useful for editing files since a new buffer is made for every file that you visit. The Lisp interpreter typescript is the most useful thing I can think of for the initial buffer to do. Type M-x lisp-interaction-mode to put the current buffer in Lisp Interaction mode.

An alternative way of evaluating Emacs Lisp expressions interactively is to use Inferior Emacs-Lisp mode, which provides an interface rather like Shell mode (see section Shell Mode) for evaluating Emacs Lisp expressions. Type M-x ielm to create an `*ielm*' buffer which uses this mode.

Running an External Lisp

Emacs has facilities for running programs in other Lisp systems. You can run a Lisp process as an inferior of Emacs, and pass expressions to it to be evaluated. You can also pass changed function definitions directly from the Emacs buffers in which you edit the Lisp programs to the inferior Lisp process.

To run an inferior Lisp process, type M-x run-lisp. This runs the program named `lisp`, the same program you would run by typing `lisp` as a shell command, with both input and output going through an Emacs buffer named `*lisp*'. That is to say, any "terminal output" from Lisp will go into the buffer, advancing point, and any "terminal input" for Lisp comes from text in the buffer. (You can change the name of the Lisp executable file by setting the variable `inferior-lisp-program`.)

To give input to Lisp, go to the end of the buffer and type the input, terminated by RET. The `*lisp*' buffer is in Inferior Lisp mode, which combines the special characteristics of Lisp mode with most of the features of Shell mode (see section Shell Mode). The definition of RET to send a line to a subprocess is one of the features of Shell mode.

For the source files of programs to run in external Lisps, use Lisp mode. This mode can be selected with M-x lisp-mode, and is used automatically for files whose names end in `.l', `.lsp', or `.lisp', as most Lisp systems usually expect.

When you edit a function in a Lisp program you are running, the easiest way to send the changed definition to the inferior Lisp process is the key C-M-x. In Lisp mode, this runs the function `lisp-eval-defun`, which finds the defun around or following point and sends it as input to the Lisp process. (Emacs can send input to any inferior process regardless of what buffer is current.)

Contrast the meanings of C-M-x in Lisp mode (for editing programs to be run in another Lisp system) and Emacs-Lisp mode (for editing Lisp programs to be run in Emacs): in both modes it has the effect of installing the function definition that point is in, but the way of doing so is different according to where the relevant Lisp environment is found. See section Executing Lisp Expressions.

Go to the previous, next section.

Go to the previous, next section.

Abbrevs

A defined abbrev is a word which expands, if you insert it, into some different text. Abbrevs are defined by the user to expand in specific ways. For example, you might define `foo' as an abbrev expanding to `find outer otter'. Then you would be able to insert `find outer otter ' into the buffer by typing f o o SPC.

A second kind of abbreviation facility is called dynamic abbrev expansion. You use dynamic abbrev expansion with an explicit command to expand the letters in the buffer before point by looking for other words in the buffer that start with those letters. See section Dynamic Abbrev Expansion.

Abbrev Concepts

An abbrev is a word which has been defined to expand into a specified expansion. When you insert a word-separator character following the abbrev, that expands the abbrev--replacing the abbrev with its expansion. For example, if `foo' is defined as an abbrev expanding to `find outer otter', then you can insert `find outer otter.' into the buffer by typing f o o ..

Abbrevs expand only when Abbrev mode (a minor mode) is enabled. Disabling Abbrev mode does not cause abbrev definitions to be forgotten, but they do not expand until Abbrev mode is enabled again. The command M-x abbrev-mode toggles Abbrev mode; with a numeric argument, it turns Abbrev mode on if the argument is positive, off otherwise. See section Minor Modes. `abbrev-mode` is also a variable; Abbrev mode is on when the variable is non-`nil`. The variable `abbrev-mode` automatically becomes local to the current buffer when it is set.

Abbrev definitions can be mode-specific---active only in one major mode. Abbrevs can also have global definitions that are active in all major modes. The same abbrev can have a global definition and various mode-specific definitions for different major modes. A mode specific definition for the current major mode overrides a global definition.

Abbrevs can be defined interactively during the editing session. Lists of abbrev definitions can also be saved in files and reloaded in later sessions. Some users keep extensive lists of abbrevs that they load in every session.

Defining Abbrevs

C-x a g

> Define an abbrev, using one or more words before point as its expansion (`add-global-abbrev`).

C-x a l

> Similar, but define an abbrev specific to the current major mode (`add-mode-abbrev`).

C-x a i g

Define a word in the buffer as an abbrev (`inverse-add-global-abbrev`).

C-x a i l

Define a word in the buffer as a mode-specific abbrev (`inverse-add-mode-abbrev`).

M-x kill-all-abbrevs

This command discards all abbrev definitions currently in effect, leaving a blank slate.

The usual way to define an abbrev is to enter the text you want the abbrev to expand to, position point after it, and type C-x a g (`add-global-abbrev`). This reads the abbrev itself using the minibuffer, and then defines it as an abbrev for one or more words before point. Use a numeric argument to say how many words before point should be taken as the expansion. For example, to define the abbrev `foo' as mentioned above, insert the text `find outer otter' and then type C-u 3 C-x a g f o o RET.

An argument of zero to C-x a g means to use the contents of the region as the expansion of the abbrev being defined.

The command C-x a l (`add-mode-abbrev`) is similar, but defines a mode-specific abbrev. Mode specific abbrevs are active only in a particular major mode. C-x a l defines an abbrev for the major mode in effect at the time C-x a l is typed. The arguments work the same as for C-x a g.

If the text already in the buffer is the abbrev, rather than its expansion, use command C-x a i g (`inverse-add-global-abbrev`) instead of C-x a g, or use C-x a i l (`inverse-add-mode-abbrev`) instead of C-x a l. These commands are called "inverse" because they invert the meaning of the two text strings they use (one from the buffer and one read with the minibuffer).

To change the definition of an abbrev, just define a new definition. When the abbrev has a prior definition, the abbrev definition commands ask for confirmation for replacing it.

To remove an abbrev definition, give a negative argument to the abbrev definition command: C-u - C-x a g or C-u - C-x a l. The former removes a global definition, while the latter removes a mode-specific definition.

M-x kill-all-abbrevs removes all the abbrev definitions there are, both global and local.

Controlling Abbrev Expansion

An abbrev expands whenever it is present in the buffer just before point and you type a self-inserting whitespace or punctuation character (SPC, comma, etc.). More precisely, any character that is not a word constituent expands an abbrev, and any word constituent character can be part of an abbrev. The most common way to use an abbrev is to insert it and then insert a punctuation character to expand it.

Abbrev expansion preserves case; thus, `foo' expands into `find outer otter'; `Foo' into `Find outer otter', and `FOO' into `FIND OUTER OTTER' or `Find Outer Otter' according to the variable `abbrev-all-caps` (a non-`nil` value chooses the first of the two expansions).

These commands are used to control abbrev expansion:

M-'

Separate a prefix from a following abbrev to be expanded (`abbrev-prefix-mark`).

C-x a e

Expand the abbrev before point (`expand-abbrev`). This is effective even when Abbrev mode is not enabled.

M-x expand-region-abbrevs

Expand some or all abbrevs found in the region.

You may wish to expand an abbrev with a prefix attached; for example, if `cnst' expands into `construction', you might want to use it to enter `reconstruction'. It does not work to type recnst, because that is not necessarily a defined abbrev. What you can do is use the command M-' (`abbrev-prefix-mark`) in between the prefix `re' and the abbrev `cnst'. First, insert `re'. Then type M-'; this inserts a hyphen in the buffer to indicate that it has done its work. Then insert the abbrev `cnst'; the buffer now contains `re-cnst'. Now insert a non-word character to expand the abbrev `cnst' into `construction'. This expansion step also deletes the hyphen that indicated M-' had been used. The result is the desired `reconstruction'.

If you actually want the text of the abbrev in the buffer, rather than its expansion, you can accomplish this by inserting the following punctuation with C-q. Thus, foo C-q , leaves `foo,' in the buffer.

If you expand an abbrev by mistake, you can undo the expansion and bring back the abbrev itself by typing C-_ to undo (see section Undoing Changes). This also undoes the insertion of the non-word character that expanded the abbrev. If the result you want is the terminating non-word character plus the unexpanded abbrev, you must reinsert the terminating character, quoting it with C-q.

M-x expand-region-abbrevs searches through the region for defined abbrevs, and for each one found offers to replace it with its expansion. This command is useful if you have typed in text using abbrevs but forgot to turn on Abbrev mode first. It may also be useful together with a special set of abbrev definitions for making several global replacements at once. This command is effective even if Abbrev mode is not enabled.

Expanding an abbrev runs the hook `pre-abbrev-expand-hook` (see section Hooks).

Examining and Editing Abbrevs

M-x list-abbrevs

Display a list of all abbrev definitions.

M-x edit-abbrevs

Edit a list of abbrevs; you can add, alter or remove definitions.

The output from M-x list-abbrevs looks like this:

```
(lisp-mode-abbrev-table)
"dk"            0    "define-key"
(global-abbrev-table)
"dfn"           0    "definition"
```

(Some blank lines of no semantic significance, and some other abbrev tables, have been omitted.)

A line containing a name in parentheses is the header for abbrevs in a particular abbrev table; `global-abbrev-table` contains all the global abbrevs, and the other abbrev tables that are named after major modes contain the mode-specific abbrevs.

Within each abbrev table, each nonblank line defines one abbrev. The word at the beginning of the line is the abbrev. The number that follows is the number of times the abbrev has been expanded. Emacs keeps track of this to help you see which abbrevs you actually use, so that you can eliminate those that you don't use often. The string at the end of the line is the expansion.

M-x edit-abbrevs allows you to add, change or kill abbrev definitions by editing a list of them in an Emacs buffer. The list has the same format described above. The buffer of abbrevs is called `*Abbrevs*', and is in Edit-Abbrevs mode. Type C-c C-c in this buffer to install the abbrev definitions as specified in the buffer--and delete any abbrev definitions not listed.

The command `edit-abbrevs` is actually the same as `list-abbrevs` except that it selects the buffer `*Abbrevs*' whereas `list-abbrevs` merely displays it in another window.

Saving Abbrevs

These commands allow you to keep abbrev definitions between editing sessions.

M-x write-abbrev-file RET file RET

: Write a file file describing all defined abbrevs.

M-x read-abbrev-file RET file RET

: Read the file file and define abbrevs as specified therein.

M-x quietly-read-abbrev-file RET file RET

: Similar but do not display a message about what is going on.

M-x define-abbrevs

: Define abbrevs from definitions in current buffer.

M-x insert-abbrevs

: Insert all abbrevs and their expansions into current buffer.

M-x write-abbrev-file reads a file name using the minibuffer and then writes a description of all current abbrev definitions into that file. This is used to save abbrev definitions for use in a later session. The text stored in the file is a series of Lisp expressions that, when executed, define the same abbrevs that you currently have.

M-x read-abbrev-file reads a file name using the minibuffer and then reads the file, defining abbrevs according to the contents of the file. M-x quietly-read-abbrev-file is the same except that it does not display a message in the echo area saying that it is doing its work; it is actually useful primarily in the `.emacs' file. If an empty argument is given to either of these functions, they use the file name specified in the variable `abbrev-file-name`, which is by default `"~/.abbrev_defs"`.

Emacs will offer to save abbrevs automatically if you have changed any of them, whenever it offers to save all files (for C-x s or C-x C-c). This feature can be inhibited by setting the variable `save-abbrevs` to `nil`.

The commands M-x insert-abbrevs and M-x define-abbrevs are similar to the previous commands but work on text in an Emacs buffer. M-x insert-abbrevs inserts text into the current buffer before point, describing all current abbrev definitions; M-x define-abbrevs parses the entire current buffer and defines abbrevs accordingly.

Dynamic Abbrev Expansion

The abbrev facility described above operates automatically as you insert text, but all abbrevs must be defined explicitly. By contrast, dynamic abbrevs allow the meanings of abbrevs to be determined automatically from the contents of the buffer, but dynamic abbrev expansion happens only when you request it explicitly.

M-/

> Expand the word in the buffer before point as a dynamic abbrev, by searching in the buffer for words starting with that abbreviation (`dabbrev-expand`).

C-M-/

> Complete the word before point as a dynamic abbrev (`dabbrev-completion`).

For example, if the buffer contains `does this follow ' and you type f o M-/, the effect is to insert `follow' because that is the last word in the buffer that starts with `fo'. A numeric argument to M-/ says to take the second, third, etc. distinct expansion found looking backward from point. Repeating M-/ searches for an alternative expansion by looking farther back. After considering the entire buffer before point, it searches the text after point. The variable `dabbrev-limit`, if non-`nil`, specifies how far in the buffer to search for an expansion.

After searching all of the current buffer, M-/ normally searches other buffers, unless you have set `dabbrev-check-all-buffers` to `nil`.

A negative argument to M-/, as in C-u - M-/, says to search first for expansions after point, and second for expansions before point. If you repeat the M-/ to look for another expansion, do not specify an argument. This tries all the expansions after point and then the expansions before point.

After you have expanded a dynamic abbrev, you can copy additional words that follow the expansion in its original context. Simply type SPC M-/ for each word you want to copy. The spacing and punctuation between words is copied along with the words.

The command C-M-/ (`dabbrev-completion`) performs completion of a dynamic abbreviation. Instead of trying the possible expansions one by one, it finds all of them, then inserts the text that they have in common. If they have nothing in common, C-M-/ displays a list of completions, from which you can select a choice in the usual manner. See section Completion.

Dynamic abbrev expansion is completely independent of Abbrev mode; the expansion of a word with M-/ is completely independent of whether it has a definition as an ordinary abbrev.

Customizing Dynamic Abbreviation

Normally, dynamic abbrev expansion ignores case when searching for expansions. That is, the expansion need not agree in case with the word you are expanding. If you set `dabbrev-case-fold-search` to `nil`, then the word and the expansion must match in case.

The value of `dabbrev-case-fold-search` may be any expression. Dynamic abbrev expansion evaluates that expression, and ignores case while searching if its value is not `nil`. The default value of `dabbrev-case-fold-search` is `case-fold-search`, so normally the value of `case-fold-search` controls the decision. The reason why dynamic abbrev expansion normally ignores case when searching for expansions is that normally the value of `case-fold-search` is `t`.

Normally, dynamic abbrev expansion preserves the case pattern *of the word you are expanding*, by converting the expansion to that case pattern. If you set `dabbrev-case-replace` to `nil`, the expansion is copied without conversion.

The variables `dabbrev-case-fold-search` and `dabbrev-case-replace` are handled in a special way. Their values are actually Lisp expressions which are evaluated each time a decision needs to be made. If the expression's value is non-`nil`, then case is ignored in searching, or converted on replacement, respectively. If the expression's value is `nil`, case is not ignored or not converted. The default values let the variables `case-fold-search` (see section Searching and Case) and `case-replace` (see section Replace Commands and Case) control what to do.

The variable `dabbrev-abbrev-char-regexp`, if non-`nil`, controls which characters are considered part of a word, for dynamic expansion purposes. The regular expression must match just one character, never two or more. The same regular expression also determines which characters are part of an expansion. The value `nil` has a special meaning: abbreviations are made of word characters, but expansions are made of word and symbol characters.

In shell scripts and makefiles, a variable name is sometimes prefixed with `$' and sometimes not. Major modes for this kind of text can customize dynamic abbreviation to handle optional prefixes by setting the variable `dabbrev-abbrev-skip-leading-regexp`. Its value should be a regular expression that matches the optional prefix that dynamic abbreviation should ignore.

Go to the previous, next section.

Go to the previous, next section.

Editing Pictures

To edit a picture made out of text characters (for example, a picture of the division of a register into fields, as a comment in a program), use the command M-x edit-picture to enter Picture mode.

In Picture mode, editing is based on the quarter-plane model of text, according to which the text characters lie studded on an area that stretches infinitely far to the right and downward. The concept of the end of a line does not exist in this model; the most you can say is where the last nonblank character on the line is found.

Of course, Emacs really always considers text as a sequence of characters, and lines really do have ends. But Picture mode replaces the most frequently-used commands with variants that simulate the quarter-plane model of text. They do this by inserting spaces or by converting tabs to spaces.

Most of the basic editing commands of Emacs are redefined by Picture mode to do essentially the same thing but in a quarter-plane way. In addition, Picture mode defines various keys starting with the C-c prefix to run special picture editing commands.

One of these keys, C-c C-c, is pretty important. Often a picture is part of a larger file that is usually edited in some other major mode. M-x edit-picture records the name of the previous major mode so you can use the C-c C-c command (`picture-mode-exit`) later to go back to that mode. C-c C-c also deletes spaces from the ends of lines, unless given a numeric argument.

The special commands of Picture mode all work in other modes (provided the `` `picture' `` library is loaded), but are not bound to keys except in Picture mode. The descriptions below talk of moving "one column" and so on, but all the picture mode commands handle numeric arguments as their normal equivalents do.

Turning on Picture mode runs the hook `picture-mode-hook` (see section Hooks).

Basic Editing in Picture Mode

Most keys do the same thing in Picture mode that they usually do, but do it in a quarter-plane style. For example, C-f is rebound to run `picture-forward-column`, a command which moves point one column to the right, inserting a space if necessary so that the actual end of the line makes no difference. C-b is rebound to run `picture-backward-column`, which always moves point left one column, converting a tab to multiple spaces if necessary. C-n and C-p are rebound to run `picture-move-down` and `picture-move-up`, which can either insert spaces or convert tabs as necessary to make sure that point stays in exactly the same column. C-e runs `picture-end-of-line`, which moves to after the last nonblank character on the line. There is no need to change C-a, as the choice of screen model does not affect beginnings of lines.

Insertion of text is adapted to the quarter-plane screen model through the use of Overwrite mode (see section Minor Modes). Self-inserting characters replace existing text, column by column, rather than

pushing existing text to the right. RET runs `picture-newline`, which just moves to the beginnin the following line so that new text will replace that line.

Picture mode provides erasure instead of deletion and killing of text. DEL (`picture-backward-clear-column`) replaces the preceding character with a space rather than removing it; this moves point backwards. C-d (`picture-clear-column`) replaces the next character or characters with spaces, but does not move point. (If you want to clear characters to spaces and move forward over them, use SPC.) C-k (`picture-clear-line`) really kills the contents of lines, but does not delete the newlines from the buffer.

To do actual insertion, you must use special commands. C-o (`picture-open-line`) creates a blank line after the current line; it never splits a line. C-M-o, `split-line`, makes sense in Picture mode, so it is not changed. LFD (`picture-duplicate-line`) inserts below the current line another line with the same contents.

To do actual deletion in Picture mode, use C-w, C-c C-d (which is defined as `delete-char`, as C-d is in other modes), or one of the picture rectangle commands (see section Picture Mode Rectangle Commands).

Controlling Motion after Insert

Since "self-inserting" characters in Picture mode overwrite and move point, there is no essential restriction on how point should be moved. Normally point moves right, but you can specify any of the eight orthogonal or diagonal directions for motion after a "self-inserting" character. This is useful for drawing lines in the buffer.

C-c <
: Move left after insertion (`picture-movement-left`).

C-c >
: Move right after insertion (`picture-movement-right`).

C-c ^
: Move up after insertion (`picture-movement-up`).

C-c .
: Move down after insertion (`picture-movement-down`).

C-c `
: Move up and left ("northwest") after insertion (`picture-movement-nw`).

C-c '
: Move up and right ("northeast") after insertion (`picture-movement-ne`).

C-c /
: Move down and left ("southwest") after insertion (`picture-movement-sw`).

C-c \
: Move down and right ("southeast") after insertion

(`picture-movement-se`).

Two motion commands move based on the current Picture insertion direction. The command C-c C-f (`picture-motion`) moves in the same direction as motion after "insertion" currently does, while C-c C-b (`picture-motion-reverse`) moves in the opposite direction.

Picture Mode Tabs

Two kinds of tab-like action are provided in Picture mode. Use M-TAB (`picture-tab-search`) for context-based tabbing. With no argument, it moves to a point underneath the next "interesting" character that follows whitespace in the previous nonblank line. "Next" here means "appearing at a horizontal position greater than the one point starts out at." With an argument, as in C-u M-TAB, this command moves to the next such interesting character in the current line. M-TAB does not change the text; it only moves point. "Interesting" characters are defined by the variable `picture-tab-chars`, which should define a set of characters. The syntax for this variable is like the syntax used inside of `[...]' in a regular expression--but without the `[' and the `]'. Its default value is `"!-~"`.

TAB itself runs `picture-tab`, which operates based on the current tab stop settings; it is the Picture mode equivalent of `tab-to-tab-stop`. Normally it just moves point, but with a numeric argument it clears the text that it moves over.

The context-based and tab-stop-based forms of tabbing are brought together by the command C-c TAB, `picture-set-tab-stops`. This command sets the tab stops to the positions which M-TAB would consider significant in the current line. The use of this command, together with TAB, can get the effect of context-based tabbing. But M-TAB is more convenient in the cases where it is sufficient.

It may be convenient to prevent use of actual tab characters in pictures. For example, this prevents C-x TAB from messing up the picture. You can do this by setting the variable `indent-tabs-mode` to `nil`. See section Tabs vs. Spaces.

Picture Mode Rectangle Commands

Picture mode defines commands for working on rectangular pieces of the text in ways that fit with the quarter-plane model. The standard rectangle commands may also be useful (see section Rectangles).

C-c C-k

Clear out the region-rectangle with spaces (`picture-clear-rectangle`). With argument, delete the text.

C-c C-w r

Similar but save rectangle contents in register r first (`picture-clear-rectangle-to-register`).

C-c C-y

Copy last killed rectangle into the buffer by overwriting, with upper left corner at point (`picture-yank-rectangle`). With argument, insert instead.

C-c C-x r

Similar, but use the rectangle in register r (`picture-yank-rectangle-from-register`).

The picture rectangle commands C-c C-k (`picture-clear-rectangle`) and C-c C-w (`picture-clear-rectangle-to-register`) differ from the standard rectangle commands in that they normally clear the rectangle instead of deleting it; this is analogous with the way C-d is changed in Picture mode.

However, deletion of rectangles can be useful in Picture mode, so these commands delete the rectangle if given a numeric argument. C-c C-k either with or without a numeric argument saves the rectangle for C-c C-y.

The Picture mode commands for yanking rectangles differ from the standard ones in overwriting instead of inserting. This is the same way that Picture mode insertion of other text differs from other modes. C-c C-y (`picture-yank-rectangle`) inserts (by overwriting) the rectangle that was most recently killed, while C-c C-x (`picture-yank-rectangle-from-register`) does likewise for the rectangle found in a specified register.

Go to the previous, next section.

Go to the previous, next section.

Sending Mail

To send a message in Emacs, you start by typing a command (C-x m) to select and initialize the `*mail*' buffer. Then you edit the text and headers of the message in this buffer, and type another command (C-c C-s or C-c C-c) to send the message.

C-x m
: Begin composing a message to send (`mail`).

C-x 4 m
: Likewise, but display the message in another window (`mail-other-window`).

C-x 5 m
: Likewise, but make a new frame (`mail-other-frame`).

C-c C-s
: In Mail mode, send the message (`mail-send`).

C-c C-c
: Send the message and bury the mail buffer (`mail-send-and-exit`).

The command C-x m (`mail`) selects a buffer named `*mail*' and initializes it with the skeleton of an outgoing message. C-x 4 m (`mail-other-window`) selects the `*mail*' buffer in a different window, leaving the previous current buffer visible. C-x 5 m (`mail-other-frame`) creates a new frame to select the `*mail*' buffer.

Because the mail composition buffer is an ordinary Emacs buffer, you can switch to other buffers while in the middle of composing mail, and switch back later (or never). If you use the C-x m command again when you have been composing another message but have not sent it, you are asked to confirm before the old message is erased. If you answer n, the `*mail*' buffer is left selected with its old contents, so you can finish the old message and send it. C-u C-x m is another way to do this. Sending the message marks the `*mail*' buffer "unmodified", which avoids the need for confirmation when C-x m is next used.

If you are composing a message in the `*mail*' buffer and want to send another message before finishing the first, rename the `*mail*' buffer using M-x rename-uniquely (see section Miscellaneous Buffer Operations). Then you can use C-x m or its variants described above to make a new `*mail*' buffer. Once you've done that, you can work with each mail buffer independently.

The Format of the Mail Buffer

In addition to the text or body, a message has header fields which say who sent it, when, to whom, why, and so on. Some header fields such as the date and sender are created automatically after the message is sent. Others, such as the recipient names, must be specified by you in order to send the message properly.

Mail mode provides a few commands to help you edit some header fields, and some are preinitialized in

the buffer automatically at times. You can insert and edit header fields using ordinary editing comma

The line in the buffer that says

```
--text follows this line--
```

is a special delimiter that separates the headers you have specified from the text. Whatever follows this line is the text of the message; the headers precede it. The delimiter line itself does not appear in the message actually sent. The text used for the delimiter line is controlled by the variable `mail-header-separator`.

Here is an example of what the headers and text in the mail buffer might look like.

```
To: gnu@prep.ai.mit.edu
CC: lungfish@spam.org, byob@spam.org
Subject: The Emacs Manual
--Text follows this line--
Please ignore this message.
```

Mail Header Fields

A header field in the mail buffer starts with a field name at the beginning of a line, terminated by a colon. Upper and lower case are equivalent in field names (and in mailing addresses also). After the colon and optional whitespace comes the contents of the field.

You can use any name you like for a header field, but normally people use only standard field names with accepted meanings. Here is a table of fields commonly used in outgoing messages.

`To'

This field contains the mailing addresses to which the message is addressed.

`Subject'

The contents of the `Subject' field should be a piece of text that says what the message is about. The reason `Subject' fields are useful is that most mail-reading programs can provide a summary of messages, listing the subject of each message but not its text.

`CC'

This field contains additional mailing addresses to send the message to, but whose readers should not regard the message as addressed to them.

`BCC'

This field contains additional mailing addresses to send the message to, which should not appear in the header of the message actually sent. Copies sent this way are called blind carbon copies.

To send a blind carbon copy of every outgoing message to yourself, set the variable `mail-self-blind` to `t`.

`FCC'

This field contains the name of one file and directs Emacs to append a copy of the message to that

file when you send the message. If the file is in Rmail format, Emacs writes the message to Rmail format; otherwise, Emacs writes the message in system mail file format.

To put a fixed file name as in `FCC' field each time you start editing an outgoing message, set the variable `mail-archive-file-name` to that file name. Unless you remove the `FCC' field before sending, the message will be written into that file when it is sent.

`From'

Use the `From' field to say who you are, when the account you are using to send the mail is not your own. The contents of the `From' field should be a valid mailing address, since replies will normally go there. If you don't specify the `From' field yourself, Emacs uses the value of `user-mail-address` as the default.

`Reply-to'

Use this field to direct replies to a different address. Most mail-reading programs (including Rmail) automatically send replies to the `Reply-to' address in preference to the `From' address. By adding a `Reply-to' field to your header, you can work around any problems your `From' address may cause for replies.

To put a fixed `Reply-to' address into every outgoing message, set the variable `mail-default-reply-to` to that address (as a string). Then `mail` initializes the message with a `Reply-to' field as specified. You can delete or alter that header field before you send the message, if you wish. When Emacs starts up, if the environment variable `REPLYTO` is set, `mail-default-reply-to` is initialized from that environment variable.

`In-reply-to'

This field contains a piece of text describing a message you are replying to. Some mail systems can use this information to correlate related pieces of mail. Normally this field is filled in by Rmail when you reply to a message in Rmail, and you never need to think about it (see section Reading Mail with Rmail).

The `To', `CC', `BCC' and `FCC' fields can appear any number of times, to specify many places to send the message. The `To', `CC', and `BCC' fields can have continuation lines. All the lines starting with whitespace, following the line on which the field starts, are considered part of the field. For example,

```
To: foo@here.net, this@there.net,
  me@gnu.cambridge.mass.usa.earth.spiral3281
```

When you send the message, if you didn't write a `From' field yourself, Emacs puts in one for you. The variable `mail-from-style` controls the format:

`nil`

Just the email address, as in `king@grassland.com'.

`parens`

Both email address and full name, as in `king@grassland.com (Elvis Parsley)'.

`angles`

Both email address and full name, as in `Elvis Parsley <king@grassland.com>'.

Mail Aliases

You can define mail aliases in a file named `~/.mailrc'. These are short mnemonic names which stand for mail addresses or groups of mail addresses. Like many other mail programs, Emacs expands aliases when they occur in the `To', `From', `CC', `BCC', and `Reply-to' fields, plus their `Resent-' variants.

To define an alias in `~/.mailrc', write a line in the following format:

```
alias shortaddress fulladdresses
```

Here fulladdresses stands for one or more mail addresses for shortaddress to expand into. Separate multiple addresses with spaces; if an address contains a space, quote the whole address with a pair of double-quotes.

For instance, to make `maingnu` stand for `gnu@prep.ai.mit.edu` plus a local address of your own, put in this line:

```
alias maingnu gnu@prep.ai.mit.edu local-gnu
```

Emacs also recognizes include commands in `.mailrc' files. They look like this:

```
source filename
```

The file `~/.mailrc' is used primarily by other mail-reading programs; it can contain various other commands. Emacs ignores everything in it except for alias definitions and include commands.

Another way to define a mail alias, within Emacs alone, is with the `define-mail-alias` command. It prompts for the alias and then the full address. You can use it to define aliases in your `.emacs' file, like this:

```
(define-mail-alias "maingnu" "gnu@prep.ai.mit.edu")
```

`define-mail-alias` records aliases by adding them to a variable named `mail-aliases`. If you are comfortable with manipulating Lisp lists, you can set `mail-aliases` directly. The initial value of `mail-aliases` is `t`, which means that Emacs should read `.mailrc' to get the proper value.

You can specify a different file name to use instead of `~/.mailrc' by setting the variable `mail-personal-alias-file`.

Normally, Emacs expands aliases when you send the message. If you like, you can have mail aliases expand as abbrevs, as soon as you type them in (see section Abbrevs). To enable this feature, execute the following:

```
(add-hook 'mail-setup-hook 'mail-abbrevs-setup)
```

This can go in your `.emacs' file. See section Hooks. If you use this feature, you must use

`define-mail-abbrev` instead of `define-mail-alias`; the latter does not work with this package. Note that the mail abbreviation package uses the variable `mail-abbrevs` instead of `mail-aliases`, and that all alias names are converted to lower case.

The mail abbreviation package also provides the C-c C-a (`mail-interactive-insert-alias`) command, which reads an alias name (with completion) and inserts its definition at point. This is useful when editing the message text itself or a header field such as `Subject' in which Emacs does not normally expand aliases.

Note that abbrevs expand only if you insert a word-separator character afterward. However, you can rebind C-n and M-> to cause expansion as well. Here's how to do that:

```
(add-hook 'mail-setup-hook
          '(lambda ()
             (substitute-key-definition
               'next-line 'mail-abbrev-next-line
               mail-mode-map global-map)
             (substitute-key-definition
               'end-of-buffer 'mail-abbrev-end-of-buffer
               mail-mode-map global-map)))
```

Mail Mode

The major mode used in the mail buffer is Mail mode, which is much like Text mode except that various special commands are provided on the C-c prefix. These commands all have to do specifically with editing or sending the message.

C-c C-s
: Send the message, and leave the mail buffer selected (`mail-send`).

C-c C-c
: Send the message, and select some other buffer (`mail-send-and-exit`).

M-TAB
: Complete a mailing address (`mail-complete`).

C-c C-f C-t
: Move to the `To' header field, creating one if there is none (`mail-to`).

C-c C-f C-s
: Move to the `Subject' header field, creating one if there is none (`mail-subject`).

C-c C-f C-c
: Move to the `CC' header field, creating one if there is none (`mail-cc`).

C-c C-f C-b
: Move to the `BCC' header field, creating one if there is none (`mail-bcc`).

C-c C-f C-f

Move to the `FCC' header field, creating one if there is none (`mail-fcc').

C-c C-t

Move to the beginning of the message body text (`mail-text').

C-c C-w

Insert the file `~/.signature' at the end of the message text (`mail-signature').

C-c C-y

Yank the selected message from Rmail (`mail-yank-original'). This command does nothing unless your command to start sending a message was issued with Rmail.

C-c C-q

Fill all paragraphs of yanked old messages, each individually (`mail-fill-yanked-message').

M-x ispell-message

Do spelling correction on the message text, but not on citations from other messages.

There are two ways to send the message. C-c C-s (`mail-send') sends the message and marks the mail buffer unmodified, but leaves that buffer selected so that you can modify the message (perhaps with new recipients) and send it again. C-c C-c (`mail-send-and-exit') sends and then deletes the window or switches to another buffer. It puts the mail buffer at the lowest priority for reselection by default, since you are finished with using it. This is the usual way to send the message.

While editing a header field that contains mailing addresses, such as `To:', `CC:' and `BCC:', you can complete a mailing address by typing M-TAB (`mail-complete'). For completion purposes, the valid mailing addresses are taken to be the local users' names plus your personal mail aliases. Additionally, if your site provides a mail directory or a specific host to use for any unrecognized user name, you can arrange to query that host for completion--see the variables `mail-directory-process' and `mail-directory-stream' in the source code.

If you type M-TAB in the body of the message, it invokes `ispell-complete-word', as in Text mode.

Mail mode provides special commands for editing the headers and text of the message before you send it. There are five commands defined to move point to particular header fields, all based on the prefix C-c C-f (`C-f' is for "field"). They are C-c C-f C-t (`mail-to') to move to the `To' field, C-c C-f C-s (`mail-subject') for the `Subject' field, C-c C-f C-c (`mail-cc') for the `CC' field, C-c C-f C-b (`mail-bcc') for the `BCC' field, and C-c C-f C-f (`mail-fcc') for the `FCC' field. If the field in question does not exist, these commands create one. We provide special motion commands for these particular fields because they are the fields users most often want to edit.

C-c C-t (`mail-text') moves point to just after the header separator line--that is, to the beginning of the message body text.

C-c C-w (`mail-signature') adds a standard piece text at the end of the message to say more about who you are. The text comes from the file `.signature' in your home directory. To insert your signature automatically, set the variable `mail-signature' to `t'; then starting a mail message automatically inserts the contents of your `.signature' file. If you want to omit your signature from

a particular message, delete it from the buffer before you send the message.

You can also set `mail-signature` to a string; then that string is inserted automatically as your signature when you start editing a message to send.

When mail sending is invoked from the Rmail mail reader using an Rmail command, C-c C-y can be used inside the mail buffer to insert the text of the message you are replying to. Normally it indents each line of that message four spaces and eliminates most header fields. A numeric argument specifies the number of spaces to indent. An argument of just C-u says not to indent at all and not to eliminate anything. C-c C-y always uses the current message from the Rmail buffer, so you can insert several old messages by selecting one in Rmail, switching to `*mail*' and yanking it, then switching back to Rmail to select another.

You can specify the text for C-c C-y to insert at the beginning of each line: set `mail-yank-prefix` to the desired string. (A value of `nil` means to use indentation; this is the default.) However, C-u C-c C-y never adds anything at the beginning of the inserted lines, regardless of the value of `mail-yank-prefix`.

After using C-c C-y, you can use the command C-c C-q (`mail-fill-yanked-message`) to fill the paragraphs of the yanked old message or messages. One use of C-c C-q fills all such paragraphs, each one individually. To fill a single paragraph of the quoted message, use M-q, after first setting the fill prefix appropriately to handle the indentation. See section Filling Text.

You can do spelling correction on the message text you have written with the command M-x ispell-message. If you have yanked an incoming message into the outgoing draft, this command skips what was yanked, but it checks the text that you yourself inserted. (It looks for indentation or `mail-yank-prefix` to distinguish the cited lines from your input.) See section Checking and Correcting Spelling.

Mail mode defines the character `%' as a word separator; this is helpful for using the word commands to edit mail addresses.

Mail mode is normally used in buffers set up automatically by the `mail` command and related commands. However, you can also switch to Mail mode in a file-visiting buffer. That is a useful thing to do if you have saved draft message text in a file. In a file-visiting buffer, C-c C-c does not clear the modified flag, because only saving the file should do that. As a result, you don't get a warning about trying to send the same message twice.

Turning on Mail mode (which C-x m does automatically) runs the normal hooks `text-mode-hook` and `mail-mode-hook`. Initializing a new outgoing message runs the normal hook `mail-setup-hook`; if you want to add special fields to your mail header or make other changes to the appearance of the mail buffer, use that hook. See section Hooks.

The main difference between these hooks is just when they are invoked. Whenever you type M-x mail, `mail-mode-hook` runs as soon as the `*mail*' buffer is created. Then the `mail-setup` function puts in the default contents of the buffer. After these default contents are inserted, `mail-setup-hook` runs.

Distracting the NSA

M-x spook adds a line of randomly chosen keywords to an outgoing mail message. The keywords are chosen from a list of words that suggest you are discussing something subversive.

The idea behind this feature is the suspicion that the NSA snoops on all electronic mail messages that contain keywords suggesting they might find them interesting. (The NSA says they don't, but that's what they *would* say.) The idea is that if lots of people add suspicious words to their messages, the NSA will get so busy with spurious input that they will have to give up reading it all.

Here's how to insert spook keywords automatically whenever you start entering an outgoing message:

```
(add-hook 'mail-setup-hook 'spook)
```

Whether or not this confuses the NSA, it at least amuses people.

Go to the previous, next section.

Go to the previous, next section.

Reading Mail with Rmail

Rmail is an Emacs subsystem for reading and disposing of mail that you receive. Rmail stores mail messages in files called Rmail files. Reading the message in an Rmail file is done in a special major mode, Rmail mode, which redefines most letters to run commands for managing mail.

Basic Concepts of Rmail

Using Rmail in the simplest fashion, you have one Rmail file `~/RMAIL' in which all of your mail is saved. It is called your primary Rmail file. The command M-x rmail reads your primary Rmail file, merges new mail in from your inboxes, displays the first message you haven't read yet, and lets you begin reading. The variable `rmail-file-name` specifies the name of the primary Rmail file.

Rmail uses narrowing to hide all but one message in the Rmail file. The message that is shown is called the current message. Rmail mode's special commands can do such things as delete the current message, copy it into another file, send a reply, or move to another message. You can also create multiple Rmail files and use Rmail to move messages between them.

Within the Rmail file, messages are normally arranged sequentially in order of receipt; you can specify other ways to sort them. Messages are assigned consecutive integers as their message numbers. The number of the current message is displayed in Rmail's mode line, followed by the total number of messages in the file. You can move to a message by specifying its message number with the j key (see section Moving Among Messages).

Following the usual conventions of Emacs, changes in an Rmail file become permanent only when the file is saved. You can save it with s (`rmail-save`), which also expunges deleted messages from the file first (see section Deleting Messages). To save the file without expunging, use C-x C-s. Rmail also saves the Rmail file after merging new mail from an inbox file (see section Rmail Files and Inboxes).

You can exit Rmail with q (`rmail-quit`); this expunges and saves the Rmail file and then switches to another buffer. But there is no need to `exit' formally. If you switch from Rmail to editing in other buffers, and never happen to switch back, you have exited. (The Rmail command b, `rmail-bury`, does this for you.) Just make sure to save the Rmail file eventually (like any other file you have changed). C-x s is a good enough way to do this (see section Saving Files).

Scrolling Within a Message

When Rmail displays a message that does not fit on the screen, you must scroll through it to read the rest. You could do this with C-v, M-v and M-<, but in Rmail scrolling is so frequent that it deserves to be easier to type.

SPC

Scroll forward (`scroll-up`).

DEL

Scroll backward (`scroll-down`).

.

Scroll to start of message (`rmail-beginning-of-message`).

Since the most common thing to do while reading a message is to scroll through it by screenfuls, Rmail makes SPC and DEL synonyms of C-v (`scroll-up`) and M-v (`scroll-down`)

The command . (`rmail-beginning-of-message`) scrolls back to the beginning of the selected message. This is not quite the same as M-<: for one thing, it does not set the mark; for another, it resets the buffer boundaries to the current message if you have changed them.

Moving Among Messages

The most basic thing to do with a message is to read it. The way to do this in Rmail is to make the message current. The usual practice is to move sequentially through the file, since this is the order of receipt of messages. When you enter Rmail, you are positioned at the first message that you have not yet made current (that is, the first one that has the `unseen' attribute; see section Labels). Move forward to see the other new messages; move backward to reexamine old messages.

n

Move to the next nondeleted message, skipping any intervening deleted messages (`rmail-next-undeleted-message`).

p

Move to the previous nondeleted message (`rmail-previous-undeleted-message`).

M-n

Move to the next message, including deleted messages (`rmail-next-message`).

M-p

Move to the previous message, including deleted messages (`rmail-previous-message`).

j

Move to the first message. With argument n, move to message number n (`rmail-show-message`).

>

Move to the last message (`rmail-last-message`).

<

Move to the first message (`rmail-first-message`).

M-s regexp RET

Move to the next message containing a match for regexp (`rmail-search`).

- M-s regexp RET

Move to the previous message containing a match for regexp.

n and p are the usual way of moving among messages in Rmail. They move through the messages sequentially, but skip over deleted messages, which is usually what you want to do. Their command definitions are named `rmail-next-undeleted-message` and `rmail-previous-undeleted-message`. If you do not want to skip deleted messages--for example, if you want to move to a message to undelete it--use the variants M-n and M-p (`rmail-next-message` and `rmail-previous-message`). A numeric argument to any of these commands serves as a repeat count.

In Rmail, you can specify a numeric argument by typing just the digits. You don't need to type C-u first.

The M-s (`rmail-search`) command is Rmail's version of search. The usual incremental search command C-s works in Rmail, but it searches only within the current message. The purpose of M-s is to search for another message. It reads a regular expression (see section Syntax of Regular Expressions) nonincrementally, then searches starting at the beginning of the following message for a match. It then selects that message. If regexp is empty, M-s reuses the regexp used the previous time.

To search backward in the file for another message, give M-s a negative argument. In Rmail you can do this with - M-s.

It is also possible to search for a message based on labels. See section Labels.

To move to a message specified by absolute message number, use j (`rmail-show-message`) with the message number as argument. With no argument, j selects the first message. < (`rmail-first-message`) also selects the first message. > (`rmail-last-message`) selects the last message.

Deleting Messages

When you no longer need to keep a message, you can delete it. This flags it as ignorable, and some Rmail commands pretend it is no longer present; but it still has its place in the Rmail file, and still has its message number.

Expunging the Rmail file actually removes the deleted messages. The remaining messages are renumbered consecutively. Expunging is the only action that changes the message number of any message, except for undigestifying (see section Digest Messages).

d

Delete the current message, and move to the next nondeleted message (`rmail-delete-forward`).

C-d

Delete the current message, and move to the previous nondeleted message (`rmail-delete-backward`).

u

Undelete the current message, or move back to a deleted message and undelete it (`rmail-undelete-previous-message`).

x

Expunge the Rmail file (`rmail-expunge`).

There are two Rmail commands for deleting messages. Both delete the current message and select another message. d (`rmail-delete-forward`) moves to the following message, skipping messages already deleted, while C-d (`rmail-delete-backward`) moves to the previous nondeleted message. If there is no nondeleted message to move to in the specified direction, the message that was just deleted remains current.

Whenever Rmail deletes a message, it invokes the function(s) listed in `rmail-delete-message-hook`. When the hook functions are invoked, the message has been marked deleted, but it is still the current message in the Rmail buffer.

To make all the deleted messages finally vanish from the Rmail file, type x (`rmail-expunge`). Until you do this, you can still undelete the deleted messages. The undeletion command, u (`rmail-undelete-previous-message`), is designed to cancel the effect of a d command in most cases. It undeletes the current message if the current message is deleted. Otherwise it moves backward to previous messages until a deleted message is found, and undeletes that message.

You can usually undo a d with a u because the u moves back to and undeletes the message that the d deleted. But this does not work when the d skips a few already-deleted messages that follow the message being deleted; then the u command undeletes the last of the messages that were skipped. There is no clean way to avoid this problem. However, by repeating the u command, you can eventually get back to the message that you intend to undelete. You can also select a particular deleted message with the M-p command, then type u to undelete it.

A deleted message has the `deleted' attribute, and as a result `deleted' appears in the mode line when the current message is deleted. In fact, deleting or undeleting a message is nothing more than adding or removing this attribute. See section Labels.

Rmail Files and Inboxes

The operating system places incoming mail for you in a file that we call your inbox. When you start up Rmail, it runs a C program called `movemail` to copy the new messages from your inbox into your primary Rmail file, which also contains other messages saved from previous Rmail sessions. It is in this file that you actually read the mail with Rmail. This operation is called getting new mail. You can get new mail at any time in Rmail by typing g.

The variable `rmail-primary-inbox-list` contains a list of the files which are inboxes for your primary Rmail file. If you don't set this variable explicitly, it is initialized from the `MAIL` environment variable, or, as a last resort, set to `nil`, which means to use the default inbox. The default inbox is `` `/var/mail/username' ``, `` `/usr/spool/mail/username' ``, or `` `/usr/mail/username' ``, depending on your operating system. You can specify the inbox file(s) for any Rmail file with the command `set-rmail-inbox-list`; see section Multiple Rmail Files.

Some sites use a method called POP for accessing users' inbox data instead of storing the data in inbox files. `movemail` can work with POP if you compile it with the macro `MAIL_USE_POP` defined, and

then install it setuid to `root`. It is safe to install `movemail` in this way. Note: `movemail` only wo with POP3, not with older versions of POP.

Assuming you have compiled and installed `movemail` appropriately, you can specify a POP inbox with a "file name" of the form `po:username'. `movemail` handles such a name by opening a connection to the POP server. The `MAILHOST` environment variable specifies the machine to look for the server on.

Accessing mail via POP may require a password. If the variable `rmail-pop-password` is non-`nil`, it specifies the password to use for POP. Alternatively, if `rmail-pop-password-required` is non-`nil`, then Rmail asks you for the password to use.

There are two reasons for having separate Rmail files and inboxes.

1. The inbox file format varies between operating systems and according to the other mail software in use. Only one part of Rmail needs to know about the alternatives, and it need only understand how to convert all of them to Rmail's own format.
2. It is very cumbersome to access an inbox file without danger of losing mail, because it is necessary to interlock with mail delivery. Moreover, different operating systems use different interlocking techniques. The strategy of moving mail out of the inbox once and for all into a separate Rmail file avoids the need for interlocking in all the rest of Rmail, since only Rmail operates on the Rmail file.

Rmail was written to use Babyl format as its internal format. Since then, we have recognized that the usual inbox format on Unix and GNU systems is adequate for the job, and we plan to change Rmail to use that as its internal format. However, the Rmail file will still be separate from the inbox file, even on systems where their format is the same.

When getting new mail, Rmail first copies the new mail from the inbox file to the Rmail file; then it saves the Rmail file; then it truncates the inbox file. This way, a system crash may cause duplication of mail between the inbox and the Rmail file, but cannot lose mail.

When `movemail` copies mail from an inbox in the system's mailer directory, it actually puts it in an intermediate file `` `~/.newmail-inboxname' ``. Once it finishes, Rmail reads that file, merges the new mail, saves the Rmail file, and only then deletes the intermediate file. If there is a crash at the wrong time, this file continues to exist and Rmail will use it again the next time it gets new mail from that inbox.

Multiple Rmail Files

Rmail operates by default on your primary Rmail file, which is named `` `~/RMAIL' `` and receives your incoming mail from your system inbox file. But you can also have other Rmail files and edit them with Rmail. These files can receive mail through their own inboxes, or you can move messages into them with explicit Rmail commands (see section Copying Messages Out to Files).

i file RET

Read file into Emacs and run Rmail on it (`rmail-input`).

M-x set-rmail-inbox-list RET files RET

Specify inbox file names for current Rmail file to get mail from.

g

Merge new mail from current Rmail file's inboxes (`rmail-get-new-mail`).

C-u g file RET

Merge new mail from inbox file file.

To run Rmail on a file other than your primary Rmail file, you may use the i (`rmail-input`) command in Rmail. This visits the file in Rmail mode. You can use M-x rmail-input even when not in Rmail.

The file you read with i should normally be a valid Rmail file. If it is not, Rmail tries to decompose it into a stream of messages in various known formats. If it succeeds, it converts the whole file to an Rmail file. If you specify a file name that doesn't exist, i initializes a new buffer for creating a new Rmail file.

You can also select an Rmail file from a menu. Choose first the menu bar Classify item, then from the Classify menu choose the Input Rmail File item; then choose the Rmail file you want. The variables `rmail-secondary-file-directory` and `rmail-secondary-file-regexp` specify which files to offer in the menu: the first variable says which directory to find them in; the second says which files in that directory to offer (all those that match the regular expression). These variables also apply to choosing a file for output (see section Copying Messages Out to Files).

Each Rmail file can contain a list of inbox file names; you can specify this list with M-x set-rmail-inbox-list RET files RET. The argument can contain any number of file names, separated by commas. It can also be empty, which specifies that this file should have no inboxes. Once a list of inboxes is specified, the Rmail file remembers it permanently until you specify a different list.

As a special exception, if your primary Rmail file does not specify any inbox files, it uses your standard system inbox.

The g command (`rmail-get-new-mail`) merges mail into the current Rmail file from its specified inboxes. If the Rmail file has no inboxes, g does nothing. The command M-x rmail also merges new mail into your primary Rmail file.

To merge mail from a file that is not the usual inbox, give the g key a numeric argument, as in C-u g. Then it reads a file name and merges mail from that file. The inbox file is not deleted or changed in any way when g with an argument is used. This is, therefore, a general way of merging one file of messages into another.

Copying Messages Out to Files

These commands copy messages from an Rmail file into another file.

o file RET

Append a copy of the current message to the file file, using Rmail file format by default (`rmail-output-to-rmail-file`).

C-o file RET

Append a copy of the current message to the file file, using system inbox file format by default

(`rmail-output`).

The commands o and C-o copy the current message into a specified file. This file may be an Rmail file or it may be in system inbox format; the output commands ascertain the file's format and write the copied message in that format.

The o and C-o commands differ in two ways: each has its own separate default file name, and each specifies a choice of format to use when the file does not already exist. The o command uses Rmail format when it creates a new file, while C-o uses system inbox format for a new file. The default file name for o is the file name used last with o, and the default file name for C-o is the file name used last with C-o.

If the output file is an Rmail file currently visited in an Emacs buffer, the output commands copy the message into that buffer. It is up to you to save the buffer eventually in its file.

You can also output a message to an Rmail file chosen with a menu. Choose first the menu bar Classify item, then from the Classify menu choose the Output Rmail Menu item; then choose the Rmail file you want. This outputs the current message to that file, like the o command. The variables `rmail-secondary-file-directory` and `rmail-secondary-file-regexp` specify which files to offer in the menu: the first variable says which directory to find them in; the second says which files in that directory to offer (all those that match the regular expression).

Copying a message gives the original copy of the message the `filed' attribute, so that `filed' appears in the mode line when such a message is current. If you like to keep just a single copy of every mail message, set the variable `rmail-delete-after-output` to `t`; then the o and C-o commands delete the original message after copying it. (You can undelete the original afterward if you wish.)

Copying messages into files in system inbox format uses the header fields that are displayed in Rmail at the time. Thus, if you use the t command to view the entire header and then copy the message, the entire header is copied. See section Display of Messages.

The variable `rmail-output-file-alist` lets you specify intelligent defaults for the output file, based on the contents of the current message. The value should be a list whose elements have this form:

```
(regexp . name-exp)
```

If there's a match for regexp in the current message, then the default file name for output is name-exp. If multiple elements match the message, the first matching element decides the default file name. The subexpression name-exp may be a string constant giving the file name to use, or more generally it may be any Lisp expression that returns a file name as a string. `rmail-output-file-alist` applies to both o and C-o.

Labels

Each message can have various labels assigned to it as a means of classification. Each label has a name; different names are different labels. Any given label is either present or absent on a particular message. A few label names have standard meanings and are given to messages automatically by Rmail when

appropriate; these special labels are called attributes. All other labels are assigned only by users.

a label RET

Assign the label label to the current message (`rmail-add-label`).

k label RET

Remove the label label from the current message (`rmail-kill-label`).

C-M-n labels RET

Move to the next message that has one of the labels labels (`rmail-next-labeled-message`).

C-M-p labels RET

Move to the previous message that has one of the labels labels (`rmail-previous-labeled-message`).

C-M-l labels RET

Make a summary of all messages containing any of the labels labels (`rmail-summary-by-labels`).

The a (`rmail-add-label`) and k (`rmail-kill-label`) commands allow you to assign or remove any label on the current message. If the label argument is empty, it means to assign or remove the same label most recently assigned or removed.

Once you have given messages labels to classify them as you wish, there are two ways to use the labels: in moving and in summaries.

The command C-M-n labels RET (`rmail-next-labeled-message`) moves to the next message that has one of the labels labels. The argument labels specifies one or more label names, separated by commas. C-M-p (`rmail-previous-labeled-message`) is similar, but moves backwards to previous messages. A numeric argument to either command serves as a repeat count.

The command C-M-l labels RET (`rmail-summary-by-labels`) displays a summary containing only the messages that have at least one of a specified set of labels. The argument labels is one or more label names, separated by commas. See section Summaries, for information on summaries.

If the labels argument to C-M-n, C-M-p or C-M-l is empty, it means to use the last set of labels specified for any of these commands.

Some labels such as `deleted' and `filed' have built-in meanings and are assigned to or removed from messages automatically at appropriate times; these labels are called attributes. Here is a list of Rmail attributes:

`unseen'

Means the message has never been current. Assigned to messages when they come from an inbox file, and removed when a message is made current. When you start Rmail, it initially shows the first message that has this attribute.

`deleted'

Means the message is deleted. Assigned by deletion commands and removed by undeletion commands (see section Deleting Messages).

`filed'

Means the message has been copied to some other file. Assigned by the file output commands (see section Multiple Rmail Files).

`answered'

Means you have mailed an answer to the message. Assigned by the r command (`rmail-reply`). See section Sending Replies.

`forwarded'

Means you have forwarded the message. Assigned by the f command (`rmail-forward`). See section Sending Replies.

`edited'

Means you have edited the text of the message within Rmail. See section Editing Within a Message.

`resent'

Means you have resent the message. Assigned by the command M-x rmail-resend. See section Sending Replies.

All other labels are assigned or removed only by the user, and have no standard meaning.

Sending Replies

Rmail has several commands that use Mail mode to send outgoing mail. See section Sending Mail, for information on using Mail mode. What are documented here are the special commands of Rmail for entering Mail mode. Note that the usual keys for sending mail---C-x m, C-x 4 m, and C-x 5 m---are available in Rmail mode and work just as they usually do.

m

Send a message (`rmail-mail`).

c

Continue editing already started outgoing message (`rmail-continue`).

r

Send a reply to the current Rmail message (`rmail-reply`).

f

Forward current message to other users (`rmail-forward`).

C-u f

Resend the current message to other users (`rmail-resend`).

M-m

Try sending a bounced message a second time (`rmail-retry-failure`).

The most common reason to send a message while in Rmail is to reply to the message you are reading. To do this, type r (`rmail-reply`). This displays the `*mail*' buffer in another window, much like C-x

4 m, but preinitializes the `Subject', `To', `CC' and `In-reply-to' header fields based on the message y are replying to. The `To' field starts out as the address of the person who sent the message you receiv and the `CC' field starts out with all the other recipients of that message.

You can exclude certain recipients from being placed automatically in the `CC', using the variable `rmail-dont-reply-to-names`. Its value should be a regular expression (as a string); any recipient that the regular expression matches, is excluded from the `CC' field. The default value matches your own name, and any name starting with `info-'. (Those names are excluded because there is a convention of using them for large mailing lists to broadcast announcements.)

To omit the `CC' field completely for a particular reply, enter the reply command with a numeric argument: C-u r or 1 r.

Once the `*mail*' buffer has been initialized, editing and sending the mail goes as usual (see section Sending Mail). You can edit the presupplied header fields if they are not right for you. You can also use the commands of Mail mode (see section Mail Mode), including C-c C-y which yanks in the message that you are replying to. You can switch to the Rmail buffer, select a different message there, switch back, and yank the new current message.

Sometimes a message does not reach its destination. Mailers usually send the failed message back to you, enclosed in a failure message. The Rmail command M-m (`rmail-retry-failure`) prepares to send the same message a second time: it sets up a `*mail*' buffer with the same text and header fields as before. If you type C-c C-c right away, you send the message again exactly the same as the first time. Alternatively, you can edit the text or headers and then send it. The variable `rmail-retry-ignored-headers`, in the same format as `rmail-ignored-headers` (see section Display of Messages), controls which headers are stripped from the failed message when retrying it; it defaults to nil.

Another frequent reason to send mail in Rmail is to forward the current message to other users. f (`rmail-forward`) makes this easy by preinitializing the `*mail*' buffer with the current message as the text, and a subject designating a forwarded message. All you have to do is fill in the recipients and send. When you forward a message, recipients get a message which is "from" you, and which has the original message in its contents.

Forwarding a message encloses it between two delimiter lines. It also modifies every line that starts with a dash, by inserting `- ' at the start of the line. When you receive a forwarded message, if it contains something besides ordinary text--for example, program source code--you might find it useful to undo that transformation. You can do this by selecting the forwarded message and typing M-x unforward-rmail-message. This command extracts the original forwarded message, deleting the inserted `- ' strings, and inserts it into the Rmail file as a separate message immediately following the current one.

Resending is an alternative similar to forwarding; the difference is that resending sends a message that is "from" the original sender, just as it reached you--with a few added header fields `Resent-from' and `Resent-to' to indicate that it came via you. To resend a message in Rmail, use C-u f. (f runs `rmail-forward`, which is programmed to invoke `rmail-resend` if you provide a numeric argument.)

The m (`rmail-mail`) command is used to start editing an outgoing message that is not a reply. It

leaves the header fields empty. Its only difference from C-x 4 m is that it makes the Rmail buffer accessible for C-c C-y, just as r does. Thus, m can be used to reply to or forward a message; it can do anything r or f can do.

The c (`rmail-continue`) command resumes editing the `*mail*' buffer, to finish editing an outgoing message you were already composing, or to alter a message you have sent.

If you set the variable `rmail-mail-new-frame` to a non-`nil` value, then all the Rmail commands to start sending a message create a new frame to edit it in. This frame is deleted when you send the message, or when you use the `Don't Send' item in the `Mail' menu.

Summaries

A summary is a buffer containing one line per message to give you an overview of the mail in an Rmail file. Each line shows the message number, the sender, the labels, and the subject. Almost all Rmail commands are valid in the summary buffer also; these apply to the message described by the current line of the summary. Moving point in the summary buffer selects messages as you move to their summary lines.

A summary buffer applies to a single Rmail file only; if you are editing multiple Rmail files, each one can have its own summary buffer. The summary buffer name is made by appending `-summary' to the Rmail buffer's name. Normally only one summary buffer is displayed at a time.

Making Summaries

Here are the commands to create a summary for the current Rmail file. Once the Rmail file has a summary buffer, changes in the Rmail file (such as deleting or expunging messages, and getting new mail) automatically update the summary.

h

C-M-h

> Summarize all messages (`rmail-summary`).

l labels RET

C-M-l labels RET

> Summarize message that have one or more of the specified labels (`rmail-summary-by-labels`).

C-M-r rcpts RET

> Summarize messages that have one or more of the specified recipients (`rmail-summary-by-recipients`).

C-M-t topic RET

> Summarize messages that have a match for the specified regexp topic in their subjects (`rmail-summary-by-topic`).

The h or C-M-h (`rmail-summary`) command fills the summary buffer for the current Rmail file with a summary of all the messages in the file. It then displays and selects the summary buffer in another

window.

C-M-l labels RET (`rmail-summary-by-labels`) makes a partial summary mentioning only the messages that have one or more of the labels labels. labels should contain label names separated by commas.

C-M-r rcpts RET (`rmail-summary-by-recipients`) makes a partial summary mentioning only the messages that have one or more of the recipients rcpts. rcpts should contain mailing addresses separated by commas.

C-M-t topic RET (`rmail-summary-by-topic`) makes a partial summary mentioning only the messages whose subjects have a match for the regular expression topic.

Note that there is only one summary buffer for any Rmail file; making one kind of summary discards any previously made summary.

The variable `rmail-summary-window-size` says how many lines to use for the summary window.

Editing in Summaries

You can use the Rmail summary buffer to do almost anything you can do in the Rmail buffer itself. In fact, once you have a summary buffer, there's no need to switch back to the Rmail buffer.

You can select and display various messages in the Rmail buffer, from the summary buffer, just by moving point in the summary buffer to different lines. It doesn't matter what Emacs command you use to move point; whichever line point is on at the end of the command, that message is selected in the Rmail buffer.

Almost all Rmail commands work in the summary buffer as well as in the Rmail buffer. Thus, d in the summary buffer deletes the current message, u undeletes, and x expunges. o and C-o output the current message to a file; r starts a reply to it. You can scroll the current message while remaining in the summary buffer using SPC and DEL.

The Rmail commands to move between messages also work in the summary buffer, but with a twist: they move through the set of messages included in the summary. They also ensure the Rmail buffer appears on the screen (unlike cursor motion commands, which update the contents of the Rmail buffer but don't display it in a window unless it already appears). Here is a list of these commands:

n
: Move to next line, skipping lines saying `deleted', and select its message.

p
: Move to previous line, skipping lines saying `deleted', and select its message.

M-n
: Move to next line and select its message.

M-p
: Move to previous line and select its message.

>

Move to the last line, and select its message.

<

Move to the first line, and select its message.

M-s pattern RET

Search through messages for pattern starting with the current message; select the message found, and move point in the summary buffer to that message's line.

Deletion, undeletion, and getting new mail, and even selection of a different message all update the summary buffer when you do them in the Rmail buffer. If the variable `rmail-redisplay-summary` is non-`nil`, these actions also bring the summary buffer back onto the screen.

When you are finished using the summary, type w (`rmail-summary-wipe`) to delete the summary buffer's window. You can also exit Rmail while in the summary: q (`rmail-summary-quit`) deletes the summary window, then exits from Rmail by saving the Rmail file and switching to another buffer.

Sorting the Rmail File

M-x rmail-sort-by-date

Sort messages of current Rmail file by date.

M-x rmail-sort-by-subject

Sort messages of current Rmail file by subject.

M-x rmail-sort-by-author

Sort messages of current Rmail file by author's name.

M-x rmail-sort-by-recipient

Sort messages of current Rmail file by recipient's names.

M-x rmail-sort-by-correspondent

Sort messages of current Rmail file by the name of the other correspondent.

M-x rmail-sort-by-lines

Sort messages of current Rmail file by size (number of lines).

M-x rmail-sort-by-keywords RET labels RET

Sort messages of current Rmail file by labels. The argument labels should be a comma-separated list of labels. The order of these labels specifies the order of messages; messages with the first label come first, messages with the second label come second, and so on. Messages which have none of these labels come last.

The Rmail sort commands perform a *stable sort*: if there is no reason to prefer either one of two messages, their order remains unchanged. You can use this to sort by more than one criterion. For example, if you use `rmail-sort-by-date` and then `rmail-sort-by-author`, messages from the same author appear in order by date.

With a numeric argument, all these commands reverse the order of comparison. This means they sort messages from newest to oldest, from biggest to smallest, or in reverse alphabetical order.

Display of Messages

Rmail reformats the header of each message before displaying it for the first time. Reformatting hides uninteresting header fields to reduce clutter. You can use the t command to show the entire header or to repeat the header reformatting operation.

t

Toggle display of complete header (`rmail-toggle-header`).

Reformatting the header involves deleting most header fields, on the grounds that they are not interesting. The variable `rmail-ignored-headers` holds a regular expression that specifies which header fields to hide in this way--if it matches the beginning of a header field, that whole field is hidden.

Rmail saves the complete original header before reformatting; to see it, use the t command (`rmail-toggle-header`). This discards the reformatted headers of the current message and displays it with the original header. Repeating t reformats the message again. Selecting the message again also reformats.

One consequence of this is that if you edit the reformatted header (using e; see section Editing Within a Message), subsequent use of t will discard your edits. On the other hand, if you use e after t, to edit the original (unreformatted) header, those changes are permanent.

When used with a window system that supports multiple fonts, Rmail highlights certain header fields that are especially interesting--by default, the `From' and `Subject' fields. The variable `rmail-highlighted-headers` holds a regular expression that specifies the header fields to highlight; if it matches the beginning of a header field, that whole field is highlighted.

If you specify unusual colors for your text foreground and background, the colors used for highlighting may not go well with them. If so, specify different colors for the `highlight` face. That is worth doing because the `highlight` face is used for other kinds of highlighting as well. See section Using Multiple Typefaces, for how to do this.

To turn off highlighting entirely in Rmail, set `rmail-highlighted-headers` to `nil`.

Editing Within a Message

Most of the usual Emacs commands are available in Rmail mode, though a few, such as C-M-n and C-M-h, are redefined by Rmail for other purposes. However, the Rmail buffer is normally read only, and most of the letters are redefined as Rmail commands. If you want to edit the text of a message, you must use the Rmail command e.

e

Edit the current message as ordinary text.

The e command (`rmail-edit-current-message`) switches from Rmail mode into Rmail Edit mode, another major mode which is nearly the same as Text mode. The mode line indicates this change.

In Rmail Edit mode, letters insert themselves as usual and the Rmail commands are not available. W you are finished editing the message and are ready to go back to Rmail, type C-c C-c, which switche back to Rmail mode. Alternatively, you can return to Rmail mode but cancel all the editing that you have done, by typing C-c C-].

Entering Rmail Edit mode runs the hook `text-mode-hook`; then it runs the hook `rmail-edit-mode-hook` (see section Hooks). It adds the attribute `edited' to the message.

Digest Messages

A digest message is a message which exists to contain and carry several other messages. Digests are used on some moderated mailing lists; all the messages that arrive for the list during a period of time such as one day are put inside a single digest which is then sent to the subscribers. Transmitting the single digest uses much less computer time than transmitting the individual messages even though the total size is the same, because the per-message overhead in network mail transmission is considerable.

When you receive a digest message, the most convenient way to read it is to undigestify it: to turn it back into many individual messages. Then you can read and delete the individual messages as it suits you.

To do this, select the digest message and type the command M-x undigestify-rmail-message. This extracts the submessages as separate Rmail messages, and inserts them following the digest. The digest message itself is flagged as deleted.

Converting an Rmail File to Inbox Format

The command M-x unrmail converts a file in Rmail format to inbox format (also known as the system mailbox format), so that you can use it with other mail-editing tools. You must specify two arguments, the name of the Rmail file and the name to use for the converted file. M-x unrmail does not alter the Rmail file itself.

Reading Rot13 Messages

Mailing list messages that might offend some readers are sometimes encoded in a simple code called rot13---so named because it rotates the alphabet by 13 letters. This code is not for secrecy, as it provides none; rather, it enables those who might be offended to avoid ever seeing the real text of the message.

To view a buffer using the rot13 code, use the command M-x rot13-other-window. This displays the current buffer in another window which applies the code when displaying the text.

Go to the previous, next section.

Go to the previous, next section.

Dired, the Directory Editor

Dired makes an Emacs buffer containing a listing of a directory, and optionally some of its subdirectories as well. You can use the normal Emacs commands to move around in this buffer, and special Dired commands to operate on the files listed.

Entering Dired

To invoke Dired, do C-x d or M-x dired. The command reads a directory name or wildcard file name pattern as a minibuffer argument to specify which files to list. Where `dired` differs from `list-directory` is in putting the buffer into Dired mode so that the special commands of Dired are available.

The variable `dired-listing-switches` specifies the options to give to `ls` for listing directory; this string *must* contain `-l'. If you use a numeric prefix argument with the `dired` command, you can specify the `ls` switches with the minibuffer after you finish entering the directory specification.

To display the Dired buffer in another window rather than in the selected window, use C-x 4 d (`dired-other-window`) instead of C-x d. C-x 5 d (`dired-other-frame`) uses a separate frame to display the Dired buffer.

Commands in the Dired Buffer

The Dired buffer is "read-only", and inserting text in it is not useful, so ordinary printing characters such as d and x are used for special Dired commands. Some Dired commands mark or flag the current file (that is, the file on the current line); other commands operate on the marked files or on the flagged files.

All the usual Emacs cursor motion commands are available in Dired buffers. Some special purpose cursor motion commands are also provided. The keys C-n and C-p are redefined to put the cursor at the beginning of the file name on the line, rather than at the beginning of the line.

For extra convenience, SPC and n in Dired are equivalent to C-n. p is equivalent to C-p. (Moving by lines is so common in Dired that it deserves to be easy to type.) DEL (move up and unflag) is often useful simply for moving up.

Deleting Files with Dired

The primary use of Dired is to flag files for deletion and then delete the files previously flagged.

d

Flag this file for deletion.

u

Remove deletion flag on this line.

DEL

Move point to previous line and remove the deletion flag on that line.

x

Delete the files that are flagged for deletion.

You can flag a file for deletion by moving to the line describing the file and typing d. The deletion flag is visible as a `D' at the beginning of the line. This command moves point to the next line, so that repeated d commands flag successive files. A numeric argument serves as a repeat count.

The files are flagged for deletion rather than deleted immediately to reduce the danger of deleting a file accidentally. Until you direct Dired to expunge the flagged files, you can remove deletion flags using the commands u and DEL. u works just like d, but removes flags rather than making flags. DEL moves upward, removing flags; it is like u with numeric argument automatically negated.

To delete the flagged files, type x (`dired-expunge`). This command first displays a list of all the file names flagged for deletion, and requests confirmation with yes. If you confirm, Dired deletes the flagged files, then deletes their lines from the text of the Dired buffer. The shortened Dired buffer remains selected.

If you answer no or quit with C-g when asked to confirm, you return immediately to Dired, with the deletion flags still present in the buffer, and no files actually deleted.

Flagging Many Files

#

Flag all auto-save files (files whose names start and end with `#') for deletion (see section Auto-Saving: Protection Against Disasters).

~

Flag all backup files (files whose names end with `~') for deletion (see section Backup Files).

. (Period)

Flag excess numeric backup files for deletion. The oldest and newest few backup files of any one file are exempt; the middle ones are flagged.

% d regexp RET

Flag for deletion all files whose names match the regular expression regexp (`dired-flag-files-regexp`).

The #, ~ and . commands flag many files for deletion, based on their file names. These commands are useful precisely because they do not actually delete any files; you can remove the deletion flags from any flagged files that you really wish to keep.

flags for deletion all files whose names look like auto-save files (see section Auto-Saving: Protection

Against Disasters)---that is, files whose names begin and end with `#'. ~ flags for deletion all files wh names say they are backup files (see section Backup Files)---that is, whose names end in `~'.

. (Period) flags just some of the backup files for deletion: all but the oldest few and newest few backups of any one file. Normally `dired-kept-versions` (**not** `kept-new-versions`; that applies only when saving) specifies the number of newest versions of each file to keep, and `kept-old-versions` specifies the number of oldest versions to keep.

Period with a positive numeric argument, as in C-u 3 ., specifies the number of newest versions to keep, overriding `dired-kept-versions`. A negative numeric argument overrides `kept-old-versions`, using minus the value of the argument to specify the number of oldest versions of each file to keep.

The % d command flags all files whose names match a specified regular expression (`dired-flag-files-regexp`). Only the non-directory part of the file name is used in matching. You can use `^' and `$' to anchor matches. You can exclude subdirectories by hiding them (see section Hiding Subdirectories).

Visiting Files in Dired

There are several Dired commands for visiting or examining the files listed in the Dired buffer. All of them apply to the current line's file; if that file is really a directory, these commands invoke Dired on that subdirectory (making a separate Dired buffer).

f

Visit the file described on the current line, like typing C-x C-f and supplying that file name (`dired-find-file`). See section Visiting Files.

RET

Equivalent to f.

o

Like f, but uses another window to display the file's buffer (`dired-find-file-other-window`). The Dired buffer remains visible in the first window. This is like using C-x 4 C-f to visit the file. See section Multiple Windows.

C-o

Visit the file described on the current line, and display the buffer in another window, but do not select that window (`dired-display-file`).

Mouse-2

Visit the file named by the line you click on (`dired-mouse-find-file-other-window`). This uses another window to display the file, like the o command.

v

View the file described on the current line, using M-x view-file (`dired-view-file`).

Viewing a file is like visiting it, but is slanted toward moving around in the file conveniently and

does not allow changing the file. See section Miscellaneous File Operations.

Dired Marks vs. Flags

Instead of flagging a file with `D', you can mark the file with some other character (usually `*'). Most Dired commands to operate on files, aside from "expunge" (x), look for files marked with `*'.

Here are some commands for marking with `*' (and also for unmarking). (See section Deleting Files with Dired, for commands to flag and unflag files.)

m

Mark the current file with `*' (`dired-mark`). With a numeric argument n, mark the next n files starting with the current file. (If n is negative, mark the previous -n files.)

*

Mark all executable files with `*' (`dired-mark-executables`). With a numeric argument, unmark all those files.

@

Mark all symbolic links with `*' (`dired-mark-symlinks`). With a numeric argument, unmark all those files.

/

Mark with `*' all files which are actually directories, except for `.' and `..' (`dired-mark-directories`). With a numeric argument, unmark all those files.

M-DEL markchar

Remove all marks that use the character markchar (`dired-unmark-all-files`). If you specify RET as markchar, this command removes all marks, no matter what the marker character is.

With a numeric argument, this command queries about each marked file, asking whether to remove its mark. You can answer y meaning yes, n meaning no, ! to remove the marks from the remaining files without asking about them.

c old new

Replace all marks that use the character old with marks that use the character new (`dired-change-marks`). This command is the primary way to create or use marks other than `*' or `D'. The arguments are single characters--do not use RET to terminate them.

You can use almost any character as a mark character by means of this command, to distinguish various classes of files. If old is a space (` '), then the command operates on all unmarked files; if new is a space, then the command unmarks the files it acts on.

To illustrate the power of this command, here is how to put `*' marks on all the files that are unmarked, while unmarking all those that have `*' marks:

```
c * t  c SPC *  c t SPC
```

% m regexp RET

Mark (with `*') all files whose names match the regular expression regexp (`dired-mark-files-regexp`). % m is like % d, except that it marks files with `*' instead of flagging with `D'. See section Flagging Many Files.

Only the non-directory part of the file name is used in matching. Use `^' and `$' to anchor matches. Exclude subdirectories by hiding them (see section Hiding Subdirectories).

Operating on Files

This section describes the basic Dired commands to operate on one file or several files. All of these commands are capital letters; all of them use the minibuffer, either to read an argument or to ask for confirmation, before they act. All use the following convention to decide which files to manipulate:

- If you give the command a numeric prefix argument n, it operates on the next n files, starting with the current file. (If n is negative, the command operates on the -n files preceding the current line.)
- Otherwise, if some files are marked with `*', the command operates on all those files.
- Otherwise, the command operates on the current file only.

Here are the file-manipulating commands that operate on files in this way. (Some other Dired commands, such as ! and the `%' commands, also use these conventions to decide which files to work on.)

C new RET

Copy the specified files (`dired-do-copy`). The argument new is the directory to copy into, or (if copying a single file) the new name.

If `dired-copy-preserve-time` is non-`nil`, then copying with this command sets the modification time of the new file to be the same as that of the old file.

R new RET

Rename the specified files (`dired-do-rename`). The argument new is the directory to rename into, or (if renaming a single file) the new name.

Dired automatically changes the visited file name of buffers associated with renamed files so that they refer to the new names.

H new RET

Make hard links to the specified files (`dired-do-hardlink`). The argument new is the directory to make the links in, or (if making just one link) the name to give the link.

S new RET

Make symbolic links to the specified files (`dired-do-symlink`). The argument new is the directory to make the links in, or (if making just one link) the name to give the link.

M modespec RET

Change the mode (also called "permission bits") of the specified files (`dired-do-chmod`). This uses the `chmod` program, so modespec can be any argument that `chmod` can handle.

G newgroup RET

Change the group of the specified files to newgroup (`dired-do-chgrp`).

O newowner RET

Change the owner of the specified files to newowner (`dired-do-chown`). (On most systems, only the superuser can do this.)

The variable `dired-chown-program` specifies the name of the program to use to do the work (different systems put `chown` in different places).

P command RET

Print the specified files (`dired-do-print`). You must specify the command to print them with, but the minibuffer starts out with a suitable guess made using the variables `lpr-command` and `lpr-switches` (the same variables that `lpr-file` uses; see section Hardcopy Output).

Z

Compress or uncompress the specified files (`dired-do-compress`). If the file appears to be a compressed file, it is uncompressed; otherwise, it is compressed.

L

Load the specified Emacs Lisp files (`dired-do-load`). See section Libraries of Lisp Code for Emacs.

B

Byte compile the specified Emacs Lisp files (`dired-do-byte-compile`). See section 'Byte Compilation' in The Emacs Lisp Reference Manual.

A regexp RET

Search all the specified files for the regular expression regexp (`dired-do-search`).

This command is a variant of `tags-search`. The search stops at the first match it finds; use M-, to resume the search and find the next match. See section Searching and Replacing with Tags Tables.

Q from RET to RET

Perform `query-replace-regexp` on each of the specified files, replacing matches for from (a regular expression) with the string to (`dired-do-query-replace`).

This command is a variant of `tags-query-replace`. If you exit the query replace loop, you can use M-, to resume the scan and replace more matches. See section Searching and Replacing with Tags Tables.

One special file-operation command is + (`dired-create-directory`). This command reads a directory name and creates the directory if it does not already exist.

Shell Commands in Dired

The dired command ! (`dired-do-shell-command`) reads a shell command string in the minibuffer and runs that shell command on all the specified files. There are two ways of applying a shell command to multiple files:

- If you use `*' in the shell command, then it runs just once, with the list of file names substituted for the `*'. The order of file names is the order of appearance in the Dired buffer.

  Thus, ! tar cf foo.tar * RET runs `tar` on the entire list of file names, putting them into one tar file `foo.tar'.

- If the command string doesn't contain `*', then it runs once *for each file*, with the file name added at the end.

  For example, ! uudecode RET runs `uudecode` on each file.

What if you want to run the shell command once for each file but with the file name inserted in the middle? Or if you want to use the file names in a more complicated fashion? Use a shell loop. For example, this shell command would run `uuencode` on each of the specified files, writing the output into a corresponding `.uu' file:

```
for file in *; uuencode $file $file >$file.uu; done
```

The working directory for the shell command is the top level directory of the Dired buffer.

The ! command does not attempt to update the Dired buffer to show new or modified files, because it doesn't really understand shell commands, and does not know what files the shell command changed. Use the g command to update the Dired buffer (see section Updating the Dired Buffer).

Transforming File Names in Dired

Here are commands that alter file names in a systematic way:

% u

Rename each of the selected files to an upper case name (`dired-upcase`). If the old file names are `Foo' and `bar', the new names are `FOO' and `BAR'.

% l

Rename each of the selected files to a lower case name (`dired-downcase`). If the old file names are `Foo' and `bar', the new names are `foo' and `bar'.

% R from RET to RET

% C from RET to RET

% H from RET to RET

% S from RET to RET

These four commands rename, copy, make hard links and make soft links, in each case computing the new name by regular expression substitution from the name of the old file.

The four regular expression substitution commands effectively perform a search-and-replace on the selected file names in the Dired buffer. They read two arguments: a regular expression from, and a substitution pattern to.

The commands match each "old" file name against the regular expression from, and then replace the

matching part with to. You can use `\&' and `\digit' in to to refer to all or part of what the pattern mat in the old file name, as in `query-replace-regexp` (see section Query Replace). If the regular expression matches more than once in a file name, only the first match is replaced.

For example, % R ^.*$ RET x-\& RET renames each selected file by prepending `x-' to its name. The inverse of this, removing `x-' from the front of each file name, is also possible: one method is % R ^x-\(.*\)$ RET \1 RET; another is % R ^x- RET RET. (Use `^' and `$' to anchor matches that should span the whole filename.)

Normally, the replacement process does not consider the files' directory names; it operates on the file name within the directory. If you specify a numeric argument of zero, then replacement affects the entire absolute file name including directory name.

Often you will want to apply the command to all files matching the same regexp that you use in the command. To do this, mark those files with % m regexp RET, then use the same regular expression in the command to operate on the files. To make this easier, the % commands to operate on files use the last regular expression specified in any % command as a default.

File Comparison with Dired

Here are two Dired commands that compare specified files using `diff`.

=

Compare the current file (the file at point) with another file (the file at the mark) using the `diff` program (`dired-diff`). The file at the mark is the first argument of `diff`, and the file at point is the second argument.

M-=

Compare the current file with its latest backup file (`dired-backup-diff`). If the current file is itself a backup, compare it with the file it is a backup of; this way, you can compare a file with any backup version of your choice.

The backup file is the first file given to `diff`.

Subdirectories in Dired

A Dired buffer displays just one directory in the normal case; but you can optionally include its subdirectories as well.

The simplest way to include multiple directories in one Dired buffer is to specify the options `-lR' for running `ls`. (If you give a numeric argument when you run Dired, then you can specify these options in the minibuffer.) That produces a recursive directory listing showing all subdirectories at all levels.

But usually all the subdirectories are too many; usually you will prefer to include specific subdirectories only. You can do this with the i command:

i

Insert the contents of a subdirectory later in the buffer.

Use the i (`dired-maybe-insert-subdir`) command on a line that describes a file which is a directory. It inserts the contents of that directory into the same Dired buffer, and moves there. Inserted subdirectory contents follow the top-level directory of the Dired buffer, just as they do in `ls -lR' output.

If the subdirectory's contents are already present in the buffer, the i command just moves to it.

In either case, i sets the Emacs mark before moving, so C-u C-SPC takes you back to the old position in the buffer (the line describing that subdirectory).

Use the l command (`dired-do-redisplay`) to update the subdirectory's contents. Use k to delete the subdirectory. See section Updating the Dired Buffer.

Moving Over Subdirectories

When a Dired buffer lists subdirectories, you can use the page motion commands C-x [and C-x] to move by entire directories.

The following commands move across, up and down in the tree of directories within one Dired buffer. They move to directory header lines, which are the lines that give a directory's name, at the beginning of the directory's contents.

C-M-n
: Go to next subdirectory header line, regardless of level (`dired-next-subdir`).

C-M-p
: Go to previous subdirectory header line, regardless of level (`dired-prev-subdir`).

C-M-u
: Go up to the parent directory's header line (`dired-tree-up`).

C-M-d
: Go down in the directory tree, to the first subdirectory's header line (`dired-tree-down`).

Hiding Subdirectories

Hiding a subdirectory means to make it invisible, except for its header line, via selective display (see section Selective Display).

$
: Hide or reveal the subdirectory that point is in, and move point to the next subdirectory (`dired-hide-subdir`). A numeric argument serves as a repeat count.

M-$
: Hide all subdirectories in this Dired buffer, leaving only their header lines (`dired-hide-all`). Or, if any subdirectory is currently hidden, make all subdirectories visible again. You can use this command to get an overview in very deep directory trees or to move quickly to subdirectories far away.

Ordinary Dired commands never consider files inside a hidden subdirectory. For example, the comm to operate on marked files ignore files in hidden directories even if they are marked. Thus you can us hiding to temporarily exclude subdirectories from operations without having to remove the markers.

The subdirectory hiding commands toggle; that is, they hide what was visible, and show what was hidden.

Updating the Dired Buffer

This section describes commands to update the Dired buffer to reflect outside (non-Dired) changes in the directories and files, and to delete part of the Dired buffer.

g
: Update the entire contents of the Dired buffer (`revert-buffer`).

l
: Update the specified files (`dired-do-redisplay`).

k
: Delete the specified *file lines*---not the files, just the lines (`dired-do-kill-lines`).

s
: Toggle between sorting by file name and sorting by date/time (`dired-sort-toggle-or-edit`).

C-u s switches RET
: Refresh the Dired buffer using switches as `dired-listing-switches`.

Type g (`revert-buffer`) to update the contents of the Dired buffer, based on changes in the files and directories listed. This preserves all marks except for those on files that have vanished. Hidden subdirectories are updated but remain hidden.

To update only some of the files, type l (`dired-do-redisplay`). This command applies to the next n files, or to the marked files if any, or to the current file. Updating them means reading their current status from the file system and changing the buffer to reflect it properly.

If you use l on a subdirectory header line, it updates the contents of the corresponding subdirectory.

To delete the specified *file lines*---not the files, just the lines--type k (`dired-do-kill-lines`). This command applies to the next n files, or to the marked files if any, or to the current file.

If you kill the line for a file that is a directory, the directory's contents are also deleted from the buffer. Typing C-u k on the header line for a subdirectory is another way to delete a subdirectory from the Dired buffer.

The g command brings back any individual lines that you have killed in this way, but not subdirectories--you must use i to reinsert each subdirectory.

The files in a Dired buffers are normally in listed alphabetical order by file names. Alternatively Dired can sort them by date/time. The Dired command s (`dired-sort-toggle-or-edit`) switches

between these two sorting modes. The mode line in a Dired buffer indicates which way it is currently sorted--by name, or by date.

C-u s switches RET lets you specify a new value for `dired-listing-switches`.

Dired and `find`

You can select a set of files for display in a Dired buffer more flexibly by using the `find` utility to choose the files.

To search for files with names matching a wildcard pattern use M-x find-name-dired. It reads arguments directory and pattern, and chooses all the files in directory or its subdirectories whose individual names match pattern.

The files thus chosen are displayed in a Dired buffer in which the ordinary Dired commands are available.

If you want to test the contents of files, rather than their names, use M-x find-grep-dired. This command reads two minibuffer arguments, directory and regexp; it chooses all the files in directory or its subdirectories that contain a match for regexp. It works by running the programs `find` and `grep`.

The most general command in this series is M-x find-dired, which lets you specify any condition that `find` can test. It takes two minibuffer arguments, directory and find-args; it runs `find` in directory, passing find-args to tell `find` what condition to test. To use this command, you need to know how to use `find`.

The format of listing produced by these commands is controlled by the variable `find-ls-option`, whose default value specifies using options `-ldi' for `ls`. If your listings are corrupted, you may need to change the value of this variable.

Go to the previous, next section.

Go to the previous, next section.

The Calendar and the Diary

Emacs provides the functions of a desk calendar, with a diary of planned or past events. To enter the calendar, type M-x calendar; this displays a three-month calendar centered on the current month, with point on the current date. With a numeric argument, as in C-u M-x calendar, it prompts you for the month and year to be the center of the three-month calendar. The calendar uses its own buffer, whose major mode is Calendar mode.

Mouse-2 in the calendar brings up a menu of operations on a particular date; C-Mouse-3 brings up a menu of commonly used calendar features that are independent of any particular date. To exit the calendar, type q. See section 'Calendar' in The Emacs Lisp Reference Manual, for customization information about the calendar and diary.

Movement in the Calendar

Calendar mode lets you move through the calendar in logical units of time such as days, weeks, months, and years. If you move outside the three months originally displayed, the calendar display "scrolls" automatically through time to make the selected date visible. Moving to a date lets you view its holidays or diary entries, convert it to other calendars; moving longer time periods is also useful simply to scroll the calendar.

Motion by Standard Lengths of Time

The commands for movement in the calendar buffer parallel the commands for movement in text. You can move forward and backward by days, weeks, months, and years.

C-f
: Move point one day forward (`calendar-forward-day`).

C-b
: Move point one day backward (`calendar-backward-day`).

C-n
: Move point one week forward (`calendar-forward-week`).

C-p
: Move point one week backward (`calendar-backward-week`).

M-}
: Move point one month forward (`calendar-forward-month`).

M-{
: Move point one month backward (`calendar-backward-month`).

C-x]

Move point one year forward (`calendar-forward-year`).

C-x [

Move point one year backward (`calendar-forward-year`).

The day and week commands are natural analogues of the usual Emacs commands for moving by characters and by lines. Just as C-n usually moves to the same column in the following line, in Calendar mode it moves to the same day in the following week. And C-p moves to the same day in the previous week.

The arrow keys are equivalent to C-f, C-b, C-n and C-p, just as they normally are in other modes.

The commands for motion by months and years work like those for weeks, but move a larger distance. The month commands M-} and M-{ move forward or backward by an entire month's time. The year commands C-x] and C-x [move forward or backward a whole year.

The easiest way to remember these commands is to consider months and years analogous to paragraphs and pages of text, respectively. But the commands themselves are not quite analogous. The ordinary Emacs paragraph commands move to the beginning or end of a paragraph, whereas these month and year commands move by an entire month or an entire year, which usually involves skipping across the end of a month or year.

All these commands accept a numeric argument as a repeat count. For convenience, the digit keys and the minus sign specify numeric arguments in Calendar mode even without the Meta modifier. For example, 100 C-f moves point 100 days forward from its present location.

Beginning or End of Week, Month or Year

A week (or month, or year) is not just a quantity of days; we think of weeks (months, years) as starting on particular dates. So Calendar mode provides commands to move to the beginning or end of a week, month or year:

C-a

Move point to start of week (`calendar-beginning-of-week`).

C-e

Move point to end of week (`calendar-end-of-week`).

M-a

Move point to start of month (`calendar-beginning-of-month`).

M-e

Move point to end of month (`calendar-end-of-month`).

M-<

Move point to start of year (`calendar-beginning-of-year`).

M->

Move point to end of year (`calendar-end-of-year`).

These commands also take numeric arguments as repeat counts, with the repeat count indicating how

many weeks, months, or years to move backward or forward.

By default, weeks begin on Sunday. To make them begin on Monday instead, set the variable `calendar-week-start-day` to 1.

Specified Dates

Calendar mode provides commands for moving to a particular date specified in various ways.

g d
: Move point to specified date (`calendar-goto-date`).

o
: Center calendar around specified month (`calendar-other-month`).

.
: Move point to today's date (`calendar-goto-today`).

g d (`calendar-goto-date`) prompts for a year, a month, and a day of the month, and then moves to that date. Because the calendar includes all dates from the beginning of the current era, you must type the year in its entirety; that is, type `1990', not `90'.

o (`calendar-other-month`) prompts for a month and year, then centers the three-month calendar around that month.

You can return to today's date with . (`calendar-goto-today`).

Scrolling in the Calendar

The calendar display scrolls automatically through time when you move out of the visible portion. You can also scroll it manually. Imagine that the calendar window contains a long strip of paper with the months on it. Scrolling it means moving the strip so that new months become visible in the window.

C-x <
: Scroll calendar one month forward (`scroll-calendar-left`).

C-x >
: Scroll calendar one month backward (`scroll-calendar-right`).

C-v

NEXT
: Scroll calendar three months forward (`scroll-calendar-left-three-months`).

M-v

PRIOR
: Scroll calendar three months backward (`scroll-calendar-right-three-months`).

The most basic calendar scroll commands scroll by one month at a time. This means that there are two months of overlap between the display before the command and the display after. C-x < scrolls the

calendar contents one month to the left; that is, it moves the display forward in time. C-x > scrolls the contents to the right, which moves backwards in time.

The commands C-v and M-v scroll the calendar by an entire "screenful"---three months--in analogy with the usual meaning of these commands. C-v makes later dates visible and M-v makes earlier dates visible. These commands take a numeric argument as a repeat count; in particular, since C-u multiplies the next command by four, typing C-u C-v scrolls the calendar forward by a year and typing C-u M-v scrolls the calendar backward by a year.

The function keys NEXT and PRIOR are equivalent to C-v and M-v, just as they are in other modes.

Counting Days

M-=
: Display the number of days in the current region (`calendar-count-days-region`).

To determine the number of days in the region, type M-= (`calendar-count-days-region`). The numbers of days printed is *inclusive*; that is, it includes the days specified by mark and point.

Miscellaneous Calendar Commands

p d
: Display day-in-year (`calendar-print-day-of-year`).

C-c C-l
: Regenerate the calendar window (`redraw-calendar`).

SPC
: Scroll the next window (`scroll-other-window`).

q
: Exit from calendar (`exit-calendar`).

To print the number of days elapsed since the start of the year, or the number of days remaining in the year, type the p d command (`calendar-print-day-of-year`). This displays both of those numbers in the echo area. The number of days elapsed includes the selected date. The number of days remaining does not include that date.

If the calendar window text gets corrupted, type C-c C-l (`redraw-calendar`) to redraw it. (This can only happen if you use non-Calendar-mode editing commands.)

In Calendar mode, you can use SPC (`scroll-other-window`) to scroll the other window. This is handy when you display a list of holidays or diary entries in another window.

To exit from the calendar, type q (`exit-calendar`). This buries all buffers related to the calendar, selecting other buffers. (If a frame contains a dedicated calendar window, exiting from the calendar iconifies that frame.)

TeX Calendar

The Calendar TeX commands produce a buffer of LaTeX code that prints as a calendar. Depending on the command you use, the printed calendar covers the day, week, month or year that point is in.

t m
: Generate a one-month calendar (`cal-tex-cursor-month`).

t M
: Generate a sideways-printing one-month calendar (`cal-tex-cursor-month-landscape`).

t d
: Generate a one-day calendar (`cal-tex-cursor-day`).

t w 1
: Generate a one-page calendar for one week (`cal-tex-cursor-week`).

t w 2
: Generate a two-page calendar for one week (`cal-tex-cursor-week2`).

t w 3
: Generate an ISO-style calendar for one week (`cal-tex-cursor-week-iso`).

t w 4
: Generate a calendar for one Monday-starting week (`cal-tex-cursor-week-monday`).

t f w
: Generate a Filofax-style two-weeks-at-a-glance calendar (`cal-tex-cursor-filofax-2week`).

t f W
: Generate a Filofax-style one-week-at-a-glance calendar (`cal-tex-cursor-filofax-week`).

t y
: Generate a calendar for one year (`cal-tex-cursor-year`).

t Y
: Generate a sideways-printing calendar for one year (`cal-tex-cursor-landscape-year`).

t f y
: Generate a Filofax-style calendar for one year (`cal-tex-cursor-filofax-year`).

Some of these commands print the calendar sideways (in "landscape mode"), so it can be wider than it is long. Some of them use Filofax paper size (3.75in x 6.75in). All of these commands accept a prefix argument which specifies how many days, weeks, months or years to print (starting always with the selected one).

If the variable `cal-tex-holidays` is non-`nil` (the default), then the printed calendars show the holidays in `calendar-holidays`. If the variable `cal-tex-diary` is non-`nil` (the default is `nil`), diary entries are included also (in weekly and monthly calendars only).

Holidays

The Emacs calendar knows about all major and many minor holidays, and can display them.

h
: Display holidays for the selected date (`calendar-cursor-holidays`).

Mouse-2 Holidays
: Display any holidays for the date you click on.

x
: Mark holidays in the calendar window (`mark-calendar-holidays`).

u
: Unmark calendar window (`calendar-unmark`).

a
: List all holidays for the displayed three months in another window (`list-calendar-holidays`).

M-x holidays
: List all holidays for three months around today's date in another window.

To see if any holidays fall on a given date, position point on that date in the calendar window and use the h command. Alternatively, click on that date with Mouse-2 and then choose Holidays from the menu that appears. Either way, this displays the holidays for that date, in the echo area if they fit there, otherwise in a separate window.

To view the distribution of holidays for all the dates shown in the calendar, use the x command. This displays the dates that are holidays in a different face (or places a `*' after these dates, if display with multiple faces is not available). The command applies both to the currently visible months and to other months that subsequently become visible by scrolling. To turn marking off and erase the current marks, type u, which also erases any diary marks (see section The Diary).

To get even more detailed information, use the a command, which displays a separate buffer containing a list of all holidays in the current three-month range. You can use SPC in the calendar window to scroll that list.

The command M-x holidays displays the list of holidays for the current month and the preceding and succeeding months; this works even if you don't have a calendar window. If you want the list of holidays centered around a different month, use C-u M-x holidays, which prompts for the month and year.

The holidays known to Emacs include American holidays and the major Christian, Jewish, and Islamic holidays; also the solstices and equinoxes.

The dates used by Emacs for holidays are based on *current practice*, not historical fact. Historically, for instance, the start of daylight savings time and even its existence have varied from year to year, but present American law mandates that daylight savings time begins on the first Sunday in April. In an American locale, Emacs always uses this definition, even though it is wrong for some prior years.

Times of Sunrise and Sunset

Special calendar commands can tell you, to within a minute or two, the times of sunrise and sunset for any date.

S
: Display times of sunrise and sunset for the selected date (`calendar-sunrise-sunset`).

Mouse-2 Sunrise/Sunset
: Display times of sunrise and sunset for the date you click on.

M-x sunrise-sunset
: Display times of sunrise and sunset for today's date.

C-u M-x sunrise-sunset
: Display times of sunrise and sunset for a specified date.

Within the calendar, to display the *local times* of sunrise and sunset in the echo area, move point to the date you want, and type S. Alternatively, click Mouse-2 on the date, then choose Sunrise/Sunset from the menu that appears. The command M-x sunrise-sunset is available outside the calendar to display this information for today's date or a specified date. To specify a date other than today, use C-u M-x sunrise-sunset, which prompts for the year, month, and day.

You can display the times of sunrise and sunset for any location and any date with C-u C-u M-x sunrise-sunset. This asks you for a longitude, latitude, number of minutes difference from Coordinated Universal Time, and date, and then tells you the times of sunrise and sunset for that location on that date.

Because the times of sunrise and sunset depend on the location on earth, you need to tell Emacs your latitude, longitude, and location name before using these commands. Here is an example of what to set:

```
(setq calendar-latitude 40.1)
(setq calendar-longitude -88.2)
(setq calendar-location-name "Urbana, IL")
```

Use one decimal place in the values of `calendar-latitude` and `calendar-longitude`.

Your time zone also affects the local time of sunrise and sunset. Emacs usually gets time zone information from the operating system, but if these values are not what you want (or if the operating system does not supply them), you must set them yourself. Here is an example:

```
(setq calendar-time-zone -360)
(setq calendar-standard-time-zone-name "CST")
(setq calendar-daylight-time-zone-name "CDT")
```

The value of `calendar-time-zone` is the number of minutes difference between your local standard time and Coordinated Universal Time (Greenwich time). The values of `calendar-standard-time-zone-name` and `calendar-daylight-time-zone-name` are the abbreviations used in your time zone. Emacs displays the times of sunrise and sunset *corrected for*

TOC

daylight savings time. See section Daylight Savings Time, for how daylight savings time is determin

As a user, you might find it convenient to set the calendar location variables for your usual physical location in your `.emacs' file. And when you install Emacs on a machine, you can create a `default.el' file which sets them properly for the typical location of most users of that machine. See section The Init File, `~/.emacs'.

Phases of the Moon

These calendar commands display the dates and times of the phases of the moon (new moon, first quarter, full moon, last quarter). This feature is useful for debugging problems that "depend on the phase of the moon."

M

Display the dates and times for all the quarters of the moon for the three-month period shown (`calendar-phases-of-moon`).

M-x phases-of-moon

Display dates and times of the quarters of the moon for three months around today's date.

Within the calendar, use the M command to display a separate buffer of the phases of the moon for the current three-month range. The dates and times listed are accurate to within a few minutes.

Outside the calendar, use the command M-x phases-of-moon to display the list of the phases of the moon for the current month and the preceding and succeeding months. For information about a different month, use C-u M-x phases-of-moon, which prompts for the month and year.

The dates and times given for the phases of the moon are given in local time (corrected for daylight savings, when appropriate); but if the variable `calendar-time-zone` is void, Coordinated Universal Time (the Greenwich time zone) is used. See section Daylight Savings Time.

Conversion To and From Other Calendars

The Emacs calendar displayed is *always* the Gregorian calendar, sometimes called the "new style" calendar, which is used in most of the world today. However, this calendar did not exist before the sixteenth century and was not widely used before the eighteenth century; it did not fully displace the Julian calendar and gain universal acceptance until the early twentieth century. The Emacs calendar can display any month since January, year 1 of the current era, but the calendar displayed is the Gregorian, even for a date at which the Gregorian calendar did not exist.

While Emacs cannot display other calendars, it can convert dates to and from several other calendars.

Supported Calendar Systems

The ISO commercial calendar is used largely in Europe.

The Julian calendar, named after Julius Caesar, was the one used in Europe throughout medieval times, and in many countries up until the nineteenth century.

Astronomers use a simple counting of days elapsed since noon, Monday, January 1, 4713 B.C. on the Julian calendar. The number of days elapsed is called the *Julian day number* or the *Astronomical day number*.

The Hebrew calendar is used by tradition in the Jewish religion. The Emacs calendar program uses the Hebrew calendar to determine the dates of Jewish holidays. Hebrew calendar dates begin and end at sunset.

The Islamic calendar is used in many predominantly Islamic countries. Emacs uses it to determine the dates of Islamic holidays. There is no universal agreement in the Islamic world about the calendar; Emacs uses a widely accepted version, but the precise dates of Islamic holidays often depend on proclamation by religious authorities, not on calculations. As a consequence, the actual dates of observance can vary slightly from the dates computed by Emacs. Islamic calendar dates begin and end at sunset.

The French Revolutionary calendar was created by the Jacobins after the 1789 revolution, to represent a more secular and nature-based view of the annual cycle, and to install a 10-day week in a rationalization measure similar to the metric system. The French government officially abandoned this calendar at the end of 1805.

The Maya of Central America used three separate, overlapping calendar systems, the *long count*, the *tzolkin*, and the *haab*. Emacs knows about all three of these calendars. Experts dispute the exact correlation between the Mayan calendar and our calendar; Emacs uses the Goodman-Martinez-Thompson correlation in its calculations.

The Copts use a calendar based on the ancient Egyptian solar calendar. Their calendar consists of twelve 30-day months followed by an extra five day period. Once every fourth year they add a leap day to this extra period to make it six days. The Ethiopic calendar is identical in structure, but has different year numbers and month names.

The Persians use a solar calendar based on a design of Omar Khayyam. Their calendar consists of twelve months of which the first six have 31 days, the next five have 30 days, and the last has 29 in ordinary years and 30 in leap years. Leap years occur in a complicated pattern every four or five years.

The Chinese calendar is a complicated system of lunar months arranged into solar years. The years go in cycles of sixty, each year containing either twelve months in an ordinary year or thirteen months in a leap year; each month has either 29 or 30 days. Years, ordinary months, and days are named by combining one of ten "celestial stems" with one of twelve "terrestrial branches" for a total of sixty names that are repeated in a cycle of sixty.

Converting To Other Calendars

The following commands describe the selected date (the date at point) in various other calendar systems.

Mouse-2 Other Calendars
: Display the date that you click on, expressed in various other calendars.

p c
: Display ISO commercial calendar equivalent for selected day (`calendar-print-iso-date`).

p j
: Display Julian date for selected day (`calendar-print-julian-date`).

p a
: Display astronomical (Julian) day number for selected day (`calendar-print-astro-day-number`).

p h
: Display Hebrew date for selected day (`calendar-print-hebrew-date`).

p i
: Display Islamic date for selected day (`calendar-print-islamic-date`).

p f
: Display French Revolutionary date for selected day (`calendar-print-french-date`).

p C
: Display Chinese date for selected day (`calendar-print-chinese-date`).

p k
: Display Coptic date for selected day (`calendar-print-coptic-date`).

p e
: Display Ethiopic date for selected day (`calendar-print-ethiopic-date`).

p p
: Display Persian date for selected day (`calendar-print-persian-date`).

p m
: Display Mayan date for selected day (`calendar-print-mayan-date`).

If you are using X, the easiest way to translate a date into other calendars is to click on it with Mouse-2, then choose Other Calendars from the menu that appears. This displays the equivalent forms of the date in all the calendars Emacs understands, in the form of a menu. (Choosing an alternative from this menu doesn't actually do anything--the menu is used only for display.)

Put point on the desired date of the Gregorian calendar, then type the appropriate keys. The p is a mnemonic for "print" since Emacs "prints" the equivalent date in the echo area.

Converting From Other Calendars

You can use the other supported calendars to specify a date to move to. This section describes the commands for doing this using calendars other than Mayan; for the Mayan calendar, see the following section.

g c
: Move to a date specified in the ISO commercial calendar (`calendar-goto-iso-date`).

g j
: Move to a date specified in the Julian calendar (`calendar-goto-julian-date`).

g a
: Move to a date specified in astronomical (Julian) day number (`calendar-goto-astro-day-number`).

g h
: Move to a date specified in the Hebrew calendar (`calendar-goto-hebrew-date`).

g i
: Move to a date specified in the Islamic calendar (`calendar-goto-islamic-date`).

g f
: Move to a date specified in the French Revolutionary calendar (`calendar-goto-french-date`).

g C
: Move to a date specified in the Chinese calendar (`calendar-goto-chinese-date`).

g p
: Move to a date specified in the Persian calendar (`calendar-goto-persian-date`).

g k
: Move to a date specified in the Coptic calendar (`calendar-goto-coptic-date`).

g e
: Move to a date specified in the Ethiopic calendar (`calendar-goto-ethiopic-date`).

These commands ask you for a date on the other calendar, move point to the Gregorian calendar date equivalent to that date, and display the other calendar's date in the echo area. Emacs uses strict completion (see section Completion) whenever it asks you to type a month name, so you don't have to worry about the spelling of Hebrew, Islamic, or French names.

One common question concerning the Hebrew calendar is the computation of the anniversary of a date of death, called a "yahrzeit." The Emacs calendar includes a facility for such calculations. If you are in the calendar, the command M-x list-yahrzeit-dates asks you for a range of years and then displays a list of the yahrzeit dates for those years for the date given by point. If you are not in the calendar, this command first asks you for the date of death and the range of years, and then displays the list of yahrzeit dates.

Converting from the Mayan Calendar

Here are the commands to select dates based on the Mayan calendar:

g m l

Move to a date specified by the long count calendar (`calendar-goto-mayan-long-count-date`).

g m n t

Move to the next occurrence of a place in the tzolkin calendar (`calendar-next-tzolkin-date`).

g m p t

Move to the previous occurrence of a place in the tzolkin calendar (`calendar-previous-tzolkin-date`).

g m n h

Move to the next occurrence of a place in the haab calendar (`calendar-next-haab-date`).

g m p h

Move to the previous occurrence of a place in the haab calendar (`calendar-previous-haab-date`).

g m n c

Move to the next occurrence of a place in the calendar round (`calendar-next-calendar-round-date`).

g m p c

Move to the previous occurrence of a place in the calendar round (`calendar-previous-calendar-round-date`).

To understand these commands, you need to understand the Mayan calendars. The long count is a counting of days with these units:

```
1 kin = 1 day   1 uinal = 20 kin   1 tun = 18 uinal
1 katun = 20 tun   1 baktun = 20 katun
```

Thus, the long count date 12.16.11.16.6 means 12 baktun, 16 katun, 11 tun, 16 uinal, and 6 kin. The Emacs calendar can handle Mayan long count dates as early as 7.17.18.13.1, but no earlier. When you use the g m l command, type the Mayan long count date with the baktun, katun, tun, uinal, and kin separated by periods.

The Mayan tzolkin calendar is a cycle of 260 days formed by a pair of independent cycles of 13 and 20 days. Since this cycle repeats endlessly, Emacs provides commands to move backward and forward to the previous or next point in the cycle. Type g m p t to go to the previous tzolkin date; Emacs asks you for a tzolkin date and moves point to the previous occurrence of that date. Similarly, type g m n t to go to the next occurrence of a tzolkin date.

The Mayan haab calendar is a cycle of 365 days arranged as 18 months of 20 days each, followed a 5-day monthless period. Like the tzolkin cycle, this cycle repeats endlessly, and there are commands to

move backward and forward to the previous or next point in the cycle. Type g m p h to go to the previous haab date; Emacs asks you for a haab date and moves point to the previous occurrence of that date. Similarly, type g m n h to go to the next occurrence of a haab date.

The Maya also used the combination of the tzolkin date and the haab date. This combination is a cycle of about 52 years called a *calendar round*. If you type g m p c, Emacs asks you for both a haab and a tzolkin date and then moves point to the previous occurrence of that combination. Use g m n c to move point to the next occurrence of a combination. These commands signal an error if the haab/tzolkin date combination you have typed is impossible.

Emacs uses strict completion (see section Strict Completion) whenever it asks you to type a Mayan name, so you don't have to worry about spelling.

The Diary

The Emacs diary keeps track of appointments or other events on a daily basis, in conjunction with the calendar. To use the diary feature, you must first create a diary file containing a list of events and their dates. Then Emacs can automatically pick out and display the events for today, for the immediate future, or for any specified date.

By default, Emacs uses `` `~/diary' `` as the diary file. This is the same file that the `calendar` utility uses. A sample `` `~/diary' `` file is:

```
12/22/1988  Twentieth wedding anniversary!!
&1/1.       Happy New Year!
10/22       Ruth's birthday.
* 21, *:    Payday
Tuesday--weekly meeting with grad students at 10am
         Supowit, Shen, Bitner, and Kapoor to attend.
1/13/89     Friday the thirteenth!!
&thu 4pm    squash game with Lloyd.
mar 16      Dad's birthday
April 15, 1989 Income tax due.
&* 15       time cards due.
```

This example uses extra spaces to align the event descriptions of most of the entries. Such formatting is purely a matter of taste.

Although you probably will start by creating a diary manually, Emacs provides a number of commands to let you view, add, and change diary entries.

Commands Displaying Diary Entries

Once you have created a `` `~/diary' `` file, you can use the calendar to view it. You can also view today's events outside of Calendar mode.

d

Display all diary entries for the selected date (`view-diary-entries`).

Mouse-2 Diary
: Display all diary entries for the date you click on.

s
: Display the entire diary file (`show-all-diary-entries`).

m
: Mark all visible dates that have diary entries (`mark-diary-entries`).

u
: Unmark the calendar window (`calendar-unmark`).

M-x print-diary-entries
: Print hard copy of the diary display as it appears.

M-x diary
: Display all diary entries for today's date.

Displaying the diary entries with d shows in a separate window the diary entries for the selected date in the calendar. The mode line of the new window shows the date of the diary entries and any holidays that fall on that date. If you specify a numeric argument with d, it shows all the diary entries for that many successive days. Thus, 2 d displays all the entries for the selected date and for the following day.

Another way to display the diary entries for a date is to click Mouse-2 on the date, and then choose Diary from the menu that appears.

To get a broader view of which days are mentioned in the diary, use the m command. This displays the dates that have diary entries fall in a different face (or places a `+' after these dates, if display with multiple faces is not available). The command applies both to the currently visible months and to other months that subsequently become visible by scrolling. To turn marking off and erase the current marks, type u, which also turns off holiday marks (see section Holidays).

To see the full diary file, rather than just some of the entries, use the s command.

Display of selected diary entries uses the selective display feature to hide entries that don't apply. This is the same feature that Outline mode uses to show part of an outline (see section Outline Mode).

The diary buffer as you see it is an illusion, so simply printing the buffer does not print what you see on your screen. There is a special command to print hard copy of the diary buffer *as it appears*; this command is M-x print-diary-entries. It sends the data directly to the printer. You can customize it like `lpr-region` (see section Hardcopy Output).

The command M-x diary displays the diary entries for the current date, independently of the calendar display, and optionally for the next few days as well; the variable `number-of-diary-entries` specifies how many days to include. See section 'Calendar/Diary Options' in The Emacs Lisp Reference Manual.

If you put `(diary)` in your `.emacs' file, this automatically displays a window with the day's diary entries, when you enter Emacs. The mode line of the displayed window shows the date and any holidays

TOC

that fall on that date.

The Diary File

Your diary file is a file that records events associated with particular dates. The name of the diary file is specified by the variable `diary-file`; `` `~/diary' `` is the default. The `calendar` utility program supports a subset of the format allowed by the Emacs diary facilities, so you can use that utility to view the diary file, with reasonable results aside from the entries it cannot understand.

Each entry in the diary file describes one event and consists of one or more lines. An entry always begins with a date specification at the left margin. The rest of the entry is simply text to describe the event. If the entry has more than one line, then the lines after the first must begin with whitespace to indicate they continue a previous entry. Lines that do not begin with valid dates and do not continue a preceding entry are ignored.

You can inhibit the marking of certain diary entries in the calendar window; to do this, insert an ampersand (`&') at the beginning of the entry, before the date. This has no effect on display of the entry in the diary window; it affects only marks on dates in the calendar window. Nonmarking entries are especially useful for generic entries that would otherwise mark many different dates.

If the first line of a diary entry consists only of the date or day name with no following blanks or punctuation, then the diary window display doesn't include that line; only the continuation lines appear. For example, this entry:

```
02/11/1989
      Bill B. visits Princeton today
      2pm Cognitive Studies Committee meeting
      2:30-5:30 Liz at Lawrenceville
      4:00pm Dentist appt
      7:30pm Dinner at George's
      8:00-10:00pm concert
```

appears in the diary window without the date line at the beginning. This style of entry looks neater when you display just a single day's entries, but can cause confusion if you ask for more than one day's entries.

You can edit the diary entries as they appear in the window, but it is important to remember that the buffer displayed contains the *entire* diary file, with portions of it concealed from view. This means, for instance, that the C-f (`forward-char`) command can put point at what appears to be the end of the line, but what is in reality the middle of some concealed line.

Be careful when editing the diary entries! Inserting additional lines or adding/deleting characters in the middle of a visible line cannot cause problems, but editing at the end of a line may not do what you expect. Deleting a line may delete other invisible entries that follow it. Before editing the diary, it is best to display the entire file with s (`show-all-diary-entries`).

Date Formats

Here are some sample diary entries, illustrating different ways of formatting a date. The examples all show dates in American order (month, day, year), but Calendar mode supports European order (day, month, year) as an option.

```
4/20/93  Switch-over to new tabulation system
apr. 25  Start tabulating annual results
4/30  Results for April are due
*/25  Monthly cycle finishes
Friday  Don't leave without backing up files
```

The first entry appears only once, on April 20, 1993. The second and third appear every year on the specified dates, and the fourth uses a wildcard (asterisk) for the month, so it appears on the 25th of every month. The final entry appears every week on Friday.

You can use just numbers to express a date, as in `month/day' or `month/day/year'. This must be followed by a nondigit. In the date itself, month and day are numbers of one or two digits. The optional year is also a number, and may be abbreviated to the last two digits; that is, you can use `11/12/1989' or `11/12/89'.

Dates can also have the form `monthname day' or `monthname day, year', where the month's name can be spelled in full or abbreviated to three characters (with or without a period). Case is not significant.

A date may be generic; that is, partially unspecified. Then the entry applies to all dates that match the specification. If the date does not contain a year, it is generic and applies to any year. Alternatively, month, day, or year can be a `*'; this matches any month, day, or year, respectively. Thus, a diary entry `3/*/*' matches any day in March of any year; so does `march *'.

If you prefer the European style of writing dates--in which the day comes before the month--type M-x european-calendar while in the calendar, or set the variable `european-calendar-style` to `t` *before* using any calendar or diary command. This mode interprets all dates in the diary in the European manner, and also uses European style for displaying diary dates. (Note that there is no comma after the monthname in the European style.) To go back to the (default) American style of writing dates, type M-x american-calendar.

You can use the name of a day of the week as a generic date which applies to any date falling on that day of the week. You can abbreviate the day of the week to three letters (with or without a period) or spell it in full; case is not significant.

Commands to Add to the Diary

While in the calendar, there are several commands to create diary entries:

i d

Add a diary entry for the selected date (`insert-diary-entry`).

i w

Add a diary entry for the selected day of the week (`insert-weekly-diary-entry`).

i m

Add a diary entry for the selected day of the month (`insert-monthly-diary-entry`).

i y

Add a diary entry for the selected day of the year (`insert-yearly-diary-entry`).

You can make a diary entry for a specific date by selecting that date in the calendar window and typing the i d command. This command displays the end of your diary file in another window and inserts the date; you can then type the rest of the diary entry.

If you want to make a diary entry that applies to a specific day of the week, select that day of the week (any occurrence will do) and type i w. This inserts the day-of-week as a generic date; you can then type the rest of the diary entry. You can make a monthly diary entry in the same fashion. Select the day of the month, use the i m command, and type rest of the entry. Similarly, you can insert a yearly diary entry with the i y command.

All of the above commands make marking diary entries by default. To make a nonmarking diary entry, give a numeric argument to the command. For example, C-u i w makes a nonmarking weekly diary entry.

When you modify the diary file, be sure to save the file before exiting Emacs.

Special Diary Entries

In addition to entries based on calendar dates, the diary file can contain sexp entries for regular events such as anniversaries. These entries are based on Lisp expressions (sexps) that Emacs evaluates as it scans the diary file. Instead of a date, a sexp entry contains `%%' followed by a Lisp expression which must begin and end with parentheses. The Lisp expression determines which dates the entry applies to.

Calendar mode provides commands to insert certain commonly used sexp entries:

i a

Add an anniversary diary entry for the selected date (`insert-anniversary-diary-entry`).

i b

Add a block diary entry for the current region (`insert-block-diary-entry`).

i c

Add a cyclic diary entry starting at the date (`insert-cyclic-diary-entry`).

If you want to make a diary entry that applies to the anniversary of a specific date, move point to that date and use the i a command. This displays the end of your diary file in another window and inserts the anniversary description; you can then type the rest of the diary entry. The entry looks like this:

```
%%(diary-anniversary 10 31 1948) Arthur's birthday
```

This entry applies to October 31 in any year after 1948; `10 31 1948' specifies the date. (If you are using the European calendar style, the month and day are interchanged.) The reason this expression requires a beginning year is that advanced diary functions can use it to calculate the number of elapsed years.

A block diary entry applies to a specified range of consecutive dates. Here is a block diary entry that applies to all dates from June 24, 1990 through July 10, 1990:

```
%%(diary-block 6 24 1990 7 10 1990) Vacation
```

The `6 24 1990' indicates the starting date and the `7 10 1990' indicates the stopping date. (Again, if you are using the European calendar style, the month and day are interchanged.)

To insert a block entry, place point and the mark on the two dates that begin and end the range, and type i b. This command displays the end of your diary file in another window and inserts the block description; you can then type the diary entry.

Cyclic diary entries repeat after a fixed interval of days. To create one, select the starting date and use the i c command. The command prompts for the length of interval, then inserts the entry, which looks like this:

```
%%(diary-cyclic 50 3 1 1990) Renew medication
```

This entry applies to March 1, 1990 and every 50th day following; `3 1 1990' specifies the starting date. (If you are using the European calendar style, the month and day are interchanged.)

All three of these commands make marking diary entries. To insert a nonmarking entry, give a numeric argument to the command. For example, C-u i a makes a nonmarking anniversary diary entry.

Marking sexp diary entries in the calendar is *extremely* time-consuming, since every date visible in the calendar window must be individually checked. So it's a good idea to make sexp diary entries nonmarking (with `&') when possible.

Another sophisticated kind of sexp entry, a floating diary entry, specifies a regularly-occurring event by offsets specified in days, weeks, and months. It is comparable to a crontab entry interpreted by the `cron` utility. Here is a nonmarking, floating diary entry that applies to the last Thursday in November:

```
&%%(diary-float 11 4 -1) American Thanksgiving
```

The 11 specifies November (the eleventh month), the 4 specifies Thursday (the fourth day of the week, where Sunday is numbered zero), and the -1 specifies "last" (1 would mean "first", 2 would mean "second", -2 would mean "second-to-last", and so on). The month can be a single month or a list of months. Thus you could change the 11 above to `'(1 2 3)' and have the entry apply to the last Thursday of January, February, and March. If the month is `t`, the entry applies to all months of the year.

Most generally, sexp diary entries can perform arbitrary computations to determine when they apply. See section 'Sexp Diary Entries' in The Emacs Lisp Reference Manual.

Appointments

If you have a diary entry for an appointment, and that diary entry begins with a recognizable time of day, Emacs can warn you, several minutes beforehand, that that appointment is pending. Emacs alerts you to

the appointment by displaying a message in the mode line.

To enable appointment notification, you must enable the time display feature of Emacs, M-x display-time (see section The Mode Line). You must also add the function `appt-make-list` to the `diary-hook`, like this:

```
(add-hook 'diary-hook 'appt-make-list)
```

With these preparations done, when you display the diary (either with the d command in the calendar window or with the M-x diary command), it sets up an appointment list of all the diary entries found with recognizable times of day, and reminds you just before each of them.

For example, suppose the diary file contains these lines:

```
Monday
  9:30am Coffee break
 12:00pm Lunch
```

Then on Mondays, after you have displayed the diary, you will be reminded at 9:20am about your coffee break and at 11:50am about lunch.

You can write times in am/pm style (with `12:00am' standing for midnight and `12:00pm' standing for noon), or 24-hour European/military style. You need not be consistent; your diary file can have a mixture of the two styles.

Emacs updates the appointments list automatically just after midnight. This also displays the next day's diary entries in the diary buffer, unless you set `appt-display-diary` to `nil`.

You can also use the appointment notification facility like an alarm clock. The command M-x appt-add adds entries to the appointment list without affecting your diary file. You delete entries from the appointment list with M-x appt-delete.

You can turn off the appointment notification feature at any time by setting `appt-issue-message` to `nil`.

Daylight Savings Time

Emacs understands the difference between standard time and daylight savings time--the times given for sunrise, sunset, solstices, equinoxes, and the phases of the moon take that into account. The rules for daylight savings time vary from place to place and have also varied historically from year to year. To do the job properly, Emacs needs to know which rules to use.

Some operating systems keep track of the rules that apply to the place where you are; on these systems, Emacs gets the information it needs from the system automatically. If some or all of this information is missing, Emacs fills in the gaps with the rules currently used in Cambridge, Massachusetts. If the resulting rules are not what you want, you can tell Emacs the rules to use by setting certain variables.

These values should be Lisp expressions that refer to the variable `year`, and evaluate to the Gregorian

date on which daylight savings time starts or (respectively) ends, in the form of a list `(month day year)`. The values should be `nil` if your area does not use daylight savings time.

Emacs uses these expressions to determine the starting date of daylight savings time for the holiday list and for correcting times of day in the solar and lunar calculations.

The values for Cambridge, Massachusetts are as follows:

```
(calendar-nth-named-day 1 0 4 year)
(calendar-nth-named-day -1 0 10 year)
```

That is, the first 0th day (Sunday) of the fourth month (April) in the year specified by `year`, and the last Sunday of the tenth month (October) of that year. If daylight savings time were changed to start on October 1, you would set `calendar-daylight-savings-starts` to this:

```
(list 10 1 year)
```

If there is no daylight savings time at your location, or if you want all times in standard time, set `calendar-daylight-savings-starts` and `calendar-daylight-savings-ends` to `nil`.

The variable `calendar-daylight-time-offset` specifies the difference between daylight savings time and standard time, measured in minutes. The value for Cambridge, Massachusetts is 60.

The two variables `calendar-daylight-savings-starts-time` and `calendar-daylight-savings-ends-time` specify the number of minutes after midnight local time when the transition to and from daylight savings time should occur. For Cambridge, Massachusetts both variables' values are 120.

Go to the previous, next section.

Go to the previous, next section.

Miscellaneous Commands

This chapter contains several brief topics that do not fit anywhere else: reading netnews, running shell commands and shell subprocesses, using a single shared Emacs for utilities that expect to run an editor as a subprocess, printing hardcopy, sorting text, narrowing display to part of the buffer, editing double-column files and binary files, saving an Emacs session for later resumption, emulating other editors, various diversions and amusements.

Gnus

Gnus is an Emacs package primarily designed for reading and posting Usenet news. It can also be used to read and respond to messages from a number of other sources--mail, remote directories, digests, and so on.

Here we introduce Gnus and describe several basic features. For full details on Gnus, type M-x info and then select the Gnus manual.

To start Gnus, type M-x gnus RET.

Gnus Buffers

As opposed to most normal Emacs packages, Gnus uses a number of different buffers to display information and to receive commands. The three buffers users spend most of their time in are the group buffer, the summary buffer and the article buffer.

The group buffer contains a list of groups. This is the first buffer Gnus displays when it starts up. It normally displays only the groups to which you subscribe and that contain unread articles. Use this buffer to select a specific group.

The summary buffer lists one line for each article in a single group. By default, the author, the subject and the line number are displayed for each article, but this is customizable, like most aspects of Gnus display. The summary buffer is created when you select a group in the group buffer, and is killed when you exit the group. Use this buffer to select an article.

The article buffer displays the article. In normal Gnus usage, you don't select this buffer--all useful article-oriented commands work in the summary buffer. But you can select the article buffer, and execute all Gnus commands from that buffer, if you want to.

When Gnus Starts Up

At startup, Gnus reads your `.newsrc' news initialization file and attempts to communicate with the local news server, which is a repository of news articles. The news server need not be the same computer you are logged in on.

If you start Gnus and connect to the server, but do not see any newsgroups listed in the group buffer, type L or A k to get a listing of all the groups. Then type u to toggle subscription to groups.

The first time you start Gnus, Gnus subscribes you to a few selected groups. All other groups start out as killed groups for you; you can list them with A k. All new groups that subsequently come to exist at the news server become zombie groups for you; type A z to list them. You can subscribe to a group shown in these lists using the u command.

When you quit Gnus with q, it automatically records in your `.newsrc' and `.newsrc.eld' initialization files the subscribed or unsubscribed status of all groups. You should normally not edit these files manually, but you may if you know how.

Summary of Gnus Commands

Reading news is a two step process:

1. Choose a group in the group buffer.
2. Select articles from the summary buffer. Each article selected is displayed in the article buffer in a large window, below the summary buffer in its small window.

Each Gnus buffer has its own special commands; however, the meanings of any given key in the various Gnus buffers are usually analogous, even if not identical. Here are commands for the group and summary buffers:

q

In the group buffer, update your `.newsrc' initialization file and quit Gnus.

In the summary buffer, exit the current group and return to the group buffer. Thus, typing q twice quits Gnus.

L

In the group buffer, list all the groups available on your news server (except those you have killed). This may be a long list!

l

In the group buffer, list only the groups to which you subscribe and which contain unread articles.

u

In the group buffer, unsubscribe from (or subscribe to) the group listed in the line that point is on. When you quit Gnus by typing q, Gnus lists in your `.newsrc' file which groups you have subscribed to. The next time you start Gnus, you won't see this group, because Gnus normally displays only subscribed-to groups.

C-k

In the group buffer, "kill" the current line's group--don't even list it in `.newsrc' from now on. This affects future Gnus sessions as well as the present session.

When you quit Gnus by typing q, Gnus writes information in the file `.newsrc' describing all newsgroups except those you have "killed."

SPC

In the group buffer, select the group on the line under the cursor and display the first unread ar in that group.

In the summary buffer,

Select the article on the line under the cursor if none is selected.

Scroll the text of the selected article (if there is one).

Select the next unread article if at the end of the current article.

Thus, you can move through all the articles by repeatedly typing SPC.

DEL

In the group buffer, move point to the previous group containing unread articles.

In the summary buffer, scroll the text of the article backwards.

n

Move point to the next unread group, or select the next unread article.

p

Move point to the previous unread group, or select the previous unread article.

C-n

C-p

Move point to the next or previous item, even if it is marked as read. This does not select the article or group on that line.

s

In the summary buffer, do an incremental search of the current text in the article buffer, just as if you switched to the article buffer and typed C-s.

M-s regexp RET

In the summary buffer, search forward for articles containing a match for regexp.

Running Shell Commands from Emacs

Emacs has commands for passing single command lines to inferior shell processes; it can also run a shell interactively with input and output to an Emacs buffer named `*shell*'.

M-! cmd RET

Run the shell command line cmd and display the output (`shell-command`).

M-| cmd RET

Run the shell command line cmd with region contents as input; optionally replace the region with the output (`shell-command-on-region`).

M-x shell

Run a subshell with input and output through an Emacs buffer. You can then give commands interactively.

Single Shell Commands

M-! (`shell-command`) reads a line of text using the minibuffer and executes it as a shell command in a subshell made just for that command. Standard input for the command comes from the null device. If the shell command produces any output, the output goes into an Emacs buffer named `*Shell Command Output*', which is displayed in another window but not selected. A numeric argument, as in M-1 M-!, directs this command to insert any output into the current buffer. In that case, point is left before the output and the mark is set after the output.

If the shell command line ends in `&', it runs asynchronously.

M-| (`shell-command-on-region`) is like M-! but passes the contents of the region as the standard input to the shell command, instead of no input. If a numeric argument is used, meaning insert the output in the current buffer, then the old region is deleted first and the output replaces it as the contents of the region.

Both M-! and M-| use `shell-file-name` to specify the shell to use. This variable is initialized based on your `SHELL` environment variable when Emacs is started. If the file name does not specify a directory, the directories in the list `exec-path` are searched; this list is initialized based on the environment variable `PATH` when Emacs is started. Your `` `.emacs' `` file can override either or both of these default initializations.

Both M-! and M-| wait for the shell command to complete. To stop waiting, type C-g to quit; that terminates the shell command with the signal `SIGINT`---the same signal that C-c normally generates in the shell. Emacs waits until the command actually terminates. If the shell command doesn't stop (because it ignores the `SIGINT` signal), type C-g again; this sends the command a `SIGKILL` signal which is impossible to ignore.

Interactive Inferior Shell

To run a subshell interactively, putting its typescript in an Emacs buffer, use M-x shell. This creates (or reuses) a buffer named `*shell*' and runs a subshell with input coming from and output going to that buffer. That is to say, any "terminal output" from the subshell goes into the buffer, advancing point, and any "terminal input" for the subshell comes from text in the buffer. To give input to the subshell, go to the end of the buffer and type the input, terminated by RET.

Emacs does not wait for the subshell to do anything. You can switch windows or buffers and edit them while the shell is waiting, or while it is running a command. Output from the subshell waits until Emacs has time to process it; this happens whenever Emacs is waiting for keyboard input or for time to elapse.

To make multiple subshells, rename the buffer `*shell*' to something different using M-x rename-uniquely. Then type M-x shell again to create a new buffer `*shell*' with its own subshell. If you rename this buffer as well, you can create a third one, and so on. All the subshells run independently and in parallel.

The file name used to load the subshell is the value of the variable `explicit-shell-file-name`, if that is non-`nil`. Otherwise, the environment variable `ESHELL` is used, or the environment variable `SHELL` if there is no `ESHELL`. If the file name specified is relative, the directories in the list

TOC

`exec-path` are searched; this list is initialized based on the environment variable `PATH` when Ema started. Your `.emacs' file can override either or both of these default initializations.

As soon as the subshell is started, it is sent as input the contents of the file `~/.emacs_shellname', if that file exists, where shellname is the name of the file that the shell was loaded from. For example, if you use bash, the file sent to it is `~/.emacs_bash'.

`cd`, `pushd` and `popd` commands given to the inferior shell are watched by Emacs so it can keep the `*shell*' buffer's default directory the same as the shell's working directory. These commands are recognized syntactically by examining lines of input that are sent. If you use aliases for these commands, you can tell Emacs to recognize them also. For example, if the value of the variable `shell-pushd-regexp` matches the beginning of a shell command line, that line is regarded as a `pushd` command. Change this variable when you add aliases for `pushd'. Likewise, `shell-popd-regexp` and `shell-cd-regexp` are used to recognize commands with the meaning of `popd' and `cd'. These commands are recognized only at the beginning of a shell command line.

If Emacs gets an error while trying to handle what it believes is a `cd', `pushd' or `popd' command, it runs the hook `shell-set-directory-error-hook` (see section Hooks).

If Emacs does not properly track changes in the current directory of the subshell, use the command M-x dirs to ask the shell what its current directory is. This command works for shells that support the most common command syntax; it may not work for unusual shells.

Shell Mode

Shell buffers use Shell mode, which defines several special keys attached to the C-c prefix. They are chosen to resemble the usual editing and job control characters present in shells that are not under Emacs, except that you must type C-c first. Here is a complete list of the special key bindings of Shell mode:

RET

At end of buffer send line as input; otherwise, copy current line to end of buffer and send it (`comint-send-input`). When a line is copied, any text at the beginning of the line that matches the variable `shell-prompt-pattern` is left out; this variable's value should be a regexp string that matches the prompts that your shell uses.

TAB

Complete the command name or file name before point in the shell buffer (`comint-dynamic-complete`). TAB also completes history references (see section Shell History References) and environment variable names.

The variable `shell-completion-fignore` specifies a list of file name extensions to ignore in Shell mode completion. The default setting ignores file names ending in `~', `#' or `%'. Other related Comint modes use the variable `comint-completion-fignore` instead.

M-?

Display temporarily a list of the possible completions of the file name before point in the shell buffer (`comint-dynamic-list-filename-completions`).

C-d

Either delete a character or send EOF (`comint-delchar-or-maybe-eof`). Typed at the end of the shell buffer, C-d sends EOF to the subshell. Typed at any other position in the buffer, C-d deletes a character as usual.

C-c C-a

Move to the beginning of the line, but after the prompt if any (`comint-bol`).

C-c C-u

Kill all text pending at end of buffer to be sent as input (`comint-kill-input`).

C-c C-w

Kill a word before point (`backward-kill-word`).

C-c C-c

Interrupt the shell or its current subjob if any (`comint-interrupt-subjob`).

C-c C-z

Stop the shell or its current subjob if any (`comint-stop-subjob`).

C-c C-\

Send quit signal to the shell or its current subjob if any (`comint-quit-subjob`).

C-c C-o

Kill the last batch of output from a shell command (`comint-kill-output`). This is useful if a shell command spews out lots of output that just gets in the way.

C-c C-r

C-M-l

Scroll to display the beginning of the last batch of output at the top of the window; also move the cursor there (`comint-show-output`).

C-c C-e

Scroll to put the end of the buffer at the bottom of the window (`comint-show-maximum-output`).

C-c C-f

Move forward across one shell command, but not beyond the current line (`shell-forward-command`). The variable `shell-command-regexp` specifies how to recognize the end of a command.

C-c C-b

Move backward across one shell command, but not beyond the current line (`shell-backward-command`).

C-c C-l

Display the buffer's history of shell commands in another window (`comint-dynamic-list-input-ring`).

M-x dirs

Ask the shell what its current directory is, so that Emacs can agree with the shell.

M-x send-invisible RET text RET

Send text as input to the shell, after reading it without echoing. This is useful when a shell command runs a program that asks for a password.

Alternatively, you can arrange for Emacs to notice password prompts and turn off echoing for them, as follows:

```
(add-hook 'comint-output-filter-functions
          'comint-watch-for-password-prompt)
```

M-x comint-continue-subjob

Continue the shell process. This is useful if you accidentally suspend the shell process.(3)

M-x comint-strip-ctrl-m

Discard all control-m characters from the current group of shell output. The most convenient way to use this command is to make it run automatically when you get output from the subshell. To do that, evaluate this Lisp expression:

```
(add-hook 'comint-output-filter-functions
          'comint-strip-ctrl-m)
```

M-x comint-truncate-buffer

This command truncates the shell buffer to a certain maximum number of lines, specified by the variable `comint-buffer-maximum-size`. Here's how to do this automatically each time you get output from the subshell:

```
(add-hook 'comint-output-filter-functions
          'comint-truncate-buffer)
```

Shell mode also customizes the paragraph commands so that only shell prompts start new paragraphs. Thus, a paragraph consists of an input command plus the output that follows it in the buffer.

Shell mode is a derivative of Comint mode, a general purpose mode for communicating with interactive subprocesses. Most of the features of Shell mode actually come from Comint mode, as you can see from the command names listed above. The special features of Shell mode in particular include the choice of regular expression for detecting prompts, the directory tracking feature, and a few user commands.

Other Emacs features that use variants of Comint mode include GUD (see section Running Debuggers Under Emacs) and M-x run-lisp (see section Running an External Lisp).

You can use M-x comint-run to execute any program of your choice in a subprocess using unmodified Comint mode--without the specializations of Shell mode.

Shell Command History

Shell buffers support three ways of repeating earlier commands. You can use the same keys used in the minibuffer; these work much as they do in the minibuffer, inserting text from prior commands while point remains always at the end of the buffer. You can move through the buffer to previous inputs in their original place, then resubmit them or copy them to the end. Or you can use a `!'-style history reference.

Shell History Ring

M-p
: Fetch the next earlier old shell command.

M-n
: Fetch the next later old shell command.

M-r regexp RET

M-s regexp RET
: Search backwards or forwards for old shell commands that match regexp.

Shell buffers provide a history of previously entered shell commands. To reuse shell commands from the history, use the editing commands M-p, M-n, M-r and M-s. These work just like the minibuffer history commands except that they operate on the text at the end of the shell buffer, where you would normally insert text to send to the shell.

M-p fetches an earlier shell command to the end of the shell buffer. Successive use of M-p fetches successively earlier shell commands, each replacing any text that was already present as potential shell input. M-n does likewise except that it finds successively more recent shell commands from the buffer.

The history search commands M-r and M-s read a regular expression and search through the history for a matching command. Aside from the choice of which command to fetch, they work just like M-p and M-r. If you enter an empty regexp, these commands reuse the same regexp used last time.

When you find the previous input you want, you can resubmit it by typing RET, or you can edit it first and then resubmit it if you wish.

These commands get the text of previous shell commands from a special history list, not from the shell buffer itself. Thus, editing the shell buffer, or even killing large parts of it, does not affect the history that these commands access.

Some shells store their command histories in files so that you can refer to previous commands from previous shell sessions. Emacs reads the command history file for your chosen shell, to initialize its own command history. The file name is `~/.bash_history' for bash, `~/.sh_history' for ksh, and `~/.history' for other shells.

Shell History Copying

C-c C-p
: Move point to the previous prompt (`comint-previous-prompt`).

C-c C-n
: Move point to the following prompt (`comint-next-prompt`).

C-c RET
: Copy the input command which point is in, inserting the copy at the end of the buffer (`comint-copy-old-input`). This is useful if you move point back to a previous command.

After you copy the command, you can submit the copy as input with RET. If you wish, you can edit the copy before resubmitting it.

Moving to a previous input and then copying it with C-c RET produces the same results--the same buffer contents--that you would get by using M-p enough times to fetch that previous input from the history list. However, C-c RET copies the text from the buffer, which can be different from what is in the history list if you edit the input text in the buffer after it has been sent.

Shell History References

Various shells including csh and bash support history references that begin with `!' and `^'. Shell mode can understand these constructs and perform the history substitution for you. If you insert a history reference and type TAB, this searches the input history for a matching command, performs substitution if necessary, and places the result in the buffer in place of the history reference. For example, you can fetch the most recent command beginning with `mv' with ! m v TAB. You can edit the command if you wish, and then resubmit the command to the shell by typing RET.

History references take effect only following a shell prompt. The variable `shell-prompt-pattern` specifies how to recognize a shell prompt. Comint modes in general use the variable `comint-prompt-regexp` to specify how to find a prompt; Shell mode uses `shell-prompt-pattern` to set up the local value of `comint-prompt-regexp`.

Shell mode can optionally expand history references in the buffer when you send them to the shell. To request this, set the variable `comint-input-autoexpand` to `input`.

You can make SPC perform history expansion by binding SPC to the command `comint-magic-space`.

Shell Mode Options

If the variable `comint-scroll-to-bottom-on-input` is non-`nil`, insertion and yank commands scroll the selected window to the bottom before inserting.

If `comint-scroll-show-maximum-output` is non-`nil`, then scrolling due to arrival of output tries to place the last line of text at the bottom line of the window, so as to show as much useful text as possible. (This mimics the scrolling behavior of many terminals.) The default is `nil`.

By setting `comint-scroll-to-bottom-on-output`, you can opt for having point jump to the end of the buffer whenever output arrives--no matter where in the buffer point was before. If the value is `this`, point jumps in the selected window. If the value is `all`, point jumps in each window that shows the comint buffer. If the value is `other`, point jumps in all nonselected windows that show the current buffer. The default value is `nil`, which means point does not jump to the end.

The variable `comint-input-ignoredups` controls whether successive identical inputs are stored in the input history. A non-`nil` value means to omit an input that is the same as the previous input. The default is `nil`, which means to store each input even if it is equal to the previous input.

Three variables customize file name completion. The variable `comint-completion-addsuffix`

controls whether completion inserts a space or a slash to indicate a fully completed file or directory r (non-`nil` means do insert a space or slash). `comint-completion-recexact`, if non-`nil`, dire TAB to choose the shortest possible completion if the usual Emacs completion algorithm cannot add even a single character. `comint-completion-autolist`, if non-`nil`, says to list all the possible completions whenever completion is not exact.

The command `comint-dynamic-complete-variable` does variable name completion using the environment variables as set within Emacs. The variables controlling file name completion apply to variable name completion too. This command is normally available through the menu bar.

Command completion normally considers only executable files. If you set `shell-command-execonly` to `nil`, it considers nonexecutable files as well.

You can configure the behavior of `pushd'. Variables control whether `pushd' behaves like `cd' if no argument is given (`shell-pushd-tohome`), pop rather than rotate with a numeric argument (`shell-pushd-dextract`), and only add directories to the directory stack if they are not already on it (`shell-pushd-dunique`). The values you choose should match the underlying shell, of course.

Remote Host Shell

Emacs provides two commands for logging in to another computer and communicating with it through an Emacs buffer.

M-x telnet RET hostname RET

Set up a Telnet connection to the computer named hostname.

M-x rlogin RET hostname RET

Set up an Rlogin connection to the computer named hostname.

Use M-x telnet to set up a Telnet connection to another computer. (Telnet is the standard Internet protocol for remote login.) It reads the host name of the other computer as an argument with the minibuffer. Once the connection is established, talking to the other computer works like talking to a subshell: you can edit input with the usual Emacs commands, and send it a line at a time by typing RET. The output is inserted in the Telnet buffer interspersed with the input.

Use M-x rlogin to set up an Rlogin connection. Rlogin is another remote login communication protocol, essentially much like the Telnet protocol but incompatible with it, and supported only by certain systems. Rlogin's advantages are that you can arrange not to have to give your user name and password when communicating between two machines you frequently use, and that you can make an 8-bit-clean connection. (To do that in Emacs, set `rlogin-explicit-args` to `("-8")` before you run Rlogin.)

M-x rlogin sets up the default file directory of the Emacs buffer to access the remote host via FTP (see section File Names), and it tracks the shell commands that change the current directory just like Shell mode.

There are two ways of doing directory tracking in an Rlogin buffer--either with remote directory names `` `/host:dir/' `` or with local names (that works if the "remote" machine shares file systems with your machine of origin). You can use the command `rlogin-directory-tracking-mode` to switch modes. No argument means use remote directory names, a positive argument means use local names, and

a negative argument means turn off directory tracking.

Using Emacs as a Server

Various programs such as `mail` can invoke your choice of editor to edit a particular piece of text, such as a message that you are sending. By convention, most of these programs use the environment variable `EDITOR` to specify which editor to run. If you set `EDITOR` to `emacs', they invoke Emacs--but in an inconvenient fashion, by starting a new, separate Emacs process. This is inconvenient because it takes time and because the new Emacs process doesn't share the buffers in the existing Emacs process.

You can arrange to use your existing Emacs process as the editor for programs like `mail` by using the Emacs client and Emacs server programs. Here is how.

First, the preparation. Within Emacs, call the function `server-start`. (Your `` `.emacs' `` file can do this automatically if you add the expression `(server-start)` to it.) Then, outside Emacs, set the `EDITOR` environment variable to `emacsclient'. (Note that some programs use a different environment variable; for example, to make TeX use `emacsclient', you should set the `TEXEDIT` environment variable to `emacsclient +%d %s'.)

Then, whenever any program invokes your specified `EDITOR` program, the effect is to send a message to your principal Emacs telling it to visit a file. (That's what the program `emacsclient` does.) Emacs displays the buffer immediately and you can immediately begin editing it.

When you've finished editing that buffer, type C-x # (`server-edit`). This saves the file and sends a message back to the `emacsclient` program telling it to exit. The programs that use `EDITOR` wait for the "editor" (actually, `emacsclient`) to exit. C-x # also checks for other pending external requests to edit various files, and selects the next such file.

You can switch to a server buffer manually if you wish; you don't have to arrive at it with C-x #. But C-x # is the only way to say that you are "finished" with one.

If you set the variable `server-window` to a window or a frame, C-x # displays the server buffer in that window or in that frame.

While `mail` or another application is waiting for `emacsclient` to finish, `emacsclient` does not read terminal input. So the terminal that `mail` was using is effectively blocked for the duration. In order to edit with your principal Emacs, you need to be able to use it without using that terminal. There are two ways to do this:

- Using a window system, run `mail` and the principal Emacs in two separate windows. While `mail` is waiting for `emacsclient`, the window where it was running is blocked, but you can use Emacs by switching windows.
- Use Shell mode in Emacs to run the other program such as `mail`; then, `emacsclient` blocks only the subshell under Emacs, and you can still use Emacs to edit the file.

Some programs write temporary files for you to edit. After you edit the temporary file, the program reads it back and deletes it. If the Emacs server is later asked to edit the same file name, it should assume this has nothing to do with the previous occasion for that file name. The server accomplishes this by killing

the temporary file's buffer when you finish with the file. Use the variable `server-temp-file-regexp` to specify which files are temporary in this sense; its value should be a regular expression that matches file names that are temporary.

Hardcopy Output

The Emacs commands for making hardcopy let you print either an entire buffer or just part of one, either with or without page headers. See also the hardcopy commands of Dired (see section Miscellaneous File Operations) and the diary (see section Commands Displaying Diary Entries).

M-x print-buffer

Print hardcopy of current buffer with page headings containing the file name and page number.

M-x lpr-buffer

Print hardcopy of current buffer without page headings.

M-x print-region

Like `print-buffer` but print only the current region.

M-x lpr-region

Like `lpr-buffer` but print only the current region.

The hardcopy commands (aside from the Postscript commands) pass extra switches to the `lpr` program based on the value of the variable `lpr-switches`. Its value should be a list of strings, each string an option starting with `-'. For example, to use a printer named `nearme', set `lpr-switches` like this:

```
(setq lpr-switches '("-Pnearme"))
```

The variable `lpr-command` specifies the name of the printer program to run; the default value depends on your operating system type. On most systems, the default is `"lpr"`. The variable `lpr-headers-switches` similarly specifies the extra switches to use to make page headers. The variable `lpr-add-switches` controls whether to supply `-T' and `-J' options (suitable for `lpr`) to the printer program: `nil` means don't add them. `lpr-add-switches` should be `nil` if your printer program is not compatible with `lpr`.

Postscript Hardcopy

These commands convert buffer contents to Postscript, either printing it or leaving it in another Emacs buffer.

M-x ps-print-buffer

Print hardcopy of the current buffer in Postscript form.

M-x ps-print-region

Print hardcopy of the current region in Postscript form.

M-x ps-print-buffer-with-faces

Print hardcopy of the current buffer in Postscript form, showing the faces used in the text by

means of Postscript features.

M-x ps-print-region-with-faces

Print hardcopy of the current region in Postscript form, showing the faces used in the text.

M-x ps-spool-buffer

Generate Postscript for the current buffer text.

M-x ps-spool-region

Generate Postscript for the current region.

M-x ps-spool-buffer-with-faces

Generate Postscript for the current buffer, showing the faces used.

M-x ps-spool-region-with-faces

Generate Postscript for the current region, showing the faces used.

The Postscript commands, `ps-print-buffer` and `ps-print-region`, print buffer contents in Postscript form. One command prints the entire buffer; the other, just the region. The corresponding `-with-faces' commands, `ps-print-buffer-with-faces` and `ps-print-region-with-faces`, use Postscript features to show the faces (fonts and colors) in the text properties of the text being printed.

If you are using a color display, you can print a buffer of program code with color highlighting by turning on Font-Lock mode in that buffer, and using `ps-print-buffer-with-faces`.

All four of the commands above use the variables `ps-lpr-command` and `ps-lpr-switches` to specify how to print the output. `ps-lpr-command` specifies the command name to run, and `ps-lpr-switches` specifies command line options to use. If you don't set these variables yourself, they take their initial values from `lpr-command` and `lpr-switches`.

The variable `ps-print-header` controls whether these commands add header lines to each page--set it to `nil` to turn headers off. You can turn off color processing by setting `ps-print-color-p` to `nil`. Many other customization variables for these commands are defined and described in the Lisp file `ps-print.el'.

The commands whose names have `spool' instead of `print' generate the Postscript output in an Emacs buffer instead of sending it to the printer.

Sorting Text

Emacs provides several commands for sorting text in the buffer. All operate on the contents of the region (the text between point and the mark). They divide the text of the region into many sort records, identify a sort key for each record, and then reorder the records into the order determined by the sort keys. The records are ordered so that their keys are in alphabetical order, or, for numeric sorting, in numeric order. In alphabetic sorting, all upper case letters `A' through `Z' come before lower case `a', in accord with the ASCII character sequence.

The various sort commands differ in how they divide the text into sort records and in which part of each

record is used as the sort key. Most of the commands make each line a separate sort record, but some commands use paragraphs or pages as sort records. Most of the sort commands use each entire sort record as its own sort key, but some use only a portion of the record as the sort key.

M-x sort-lines

Divide the region into lines, and sort by comparing the entire text of a line. A numeric argument means sort into descending order.

M-x sort-paragraphs

Divide the region into paragraphs, and sort by comparing the entire text of a paragraph (except for leading blank lines). A numeric argument means sort into descending order.

M-x sort-pages

Divide the region into pages, and sort by comparing the entire text of a page (except for leading blank lines). A numeric argument means sort into descending order.

M-x sort-fields

Divide the region into lines, and sort by comparing the contents of one field in each line. Fields are defined as separated by whitespace, so the first run of consecutive non-whitespace characters in a line constitutes field 1, the second such run constitutes field 2, etc.

Specify which field to sort by with a numeric argument: 1 to sort by field 1, etc. A negative argument means count fields from the right instead of from the left; thus, minus 1 means sort by the last field. If several lines have identical contents in the field being sorted, they keep same relative order that they had in the original buffer.

A negative argument means count fields from the right (from the end of the line).

M-x sort-numeric-fields

Like M-x sort-fields except the specified field is converted to an integer for each line, and the numbers are compared. `10' comes before `2' when considered as text, but after it when considered as a number.

M-x sort-columns

Like M-x sort-fields except that the text within each line used for comparison comes from a fixed range of columns. See below for an explanation.

M-x reverse-region

Reverse the order of the lines in the region. This is useful for sorting into descending order by fields or columns, since those sort commands do not have a feature for doing that.

For example, if the buffer contains this:

```
On systems where clash detection (locking of files being edited) is
implemented, Emacs also checks the first time you modify a buffer
whether the file has changed on disk since it was last visited or
saved.  If it has, you are asked to confirm that you want to change
the buffer.
```

applying M-x sort-lines to the entire buffer produces this:


```
On systems where clash detection (locking of files being edited) is
implemented, Emacs also checks the first time you modify a buffer
saved.  If it has, you are asked to confirm that you want to change
the buffer.
whether the file has changed on disk since it was last visited or
```

where the upper case `O' sorts before all lower case letters. If you use C-u 2 M-x sort-fields instead, you get this:

```
implemented, Emacs also checks the first time you modify a buffer
saved.  If it has, you are asked to confirm that you want to change
the buffer.
On systems where clash detection (locking of files being edited) is
whether the file has changed on disk since it was last visited or
```

where the sort keys were `Emacs', `If', `buffer', `systems' and `the'.

M-x sort-columns requires more explanation. You specify the columns by putting point at one of the columns and the mark at the other column. Because this means you cannot put point or the mark at the beginning of the first line to sort, this command uses an unusual definition of `region': all of the line point is in is considered part of the region, and so is all of the line the mark is in, as well as all the lines in between.

For example, to sort a table by information found in columns 10 to 15, you could put the mark on column 10 in the first line of the table, and point on column 15 in the last line of the table, and then run `sort-columns`. Equivalently, you could run it with the mark on column 15 in the first line and point on column 10 in the last line.

This can be thought of as sorting the rectangle specified by point and the mark, except that the text on each line to the left or right of the rectangle moves along with the text inside the rectangle. See section Rectangles.

Many of the sort commands ignore case differences when comparing, if `sort-fold-case` is non-`nil`.

Narrowing

Narrowing means focusing in on some portion of the buffer, making the rest temporarily inaccessible. The portion which you can still get to is called the accessible portion. Canceling the narrowing, which makes the entire buffer once again accessible, is called widening. The amount of narrowing in effect in a buffer at any time is called the buffer's restriction.

Narrowing can make it easier to concentrate on a single subroutine or paragraph by eliminating clutter. It can also be used to restrict the range of operation of a replace command or repeating keyboard macro.

C-x n n

Narrow down to between point and mark (`narrow-to-region`).

C-x n w

Widen to make the entire buffer accessible again (`widen`).

C-x n p

Narrow down to the current page (`narrow-to-page`).

When you have narrowed down to a part of the buffer, that part appears to be all there is. You can't see the rest, you can't move into it (motion commands won't go outside the accessible part), you can't change it in any way. However, it is not gone, and if you save the file all the inaccessible text will be saved. The word `Narrow' appears in the mode line whenever narrowing is in effect.

The primary narrowing command is C-x n n (`narrow-to-region`). It sets the current buffer's restrictions so that the text in the current region remains accessible but all text before the region or after the region is inaccessible. Point and mark do not change.

Alternatively, use C-x n p (`narrow-to-page`) to narrow down to the current page. See section Pages, for the definition of a page.

The way to cancel narrowing is to widen with C-x n w (`widen`). This makes all text in the buffer accessible again.

You can get information on what part of the buffer you are narrowed down to using the C-x = command. See section Cursor Position Information.

Because narrowing can easily confuse users who do not understand it, `narrow-to-region` is normally a disabled command. Attempting to use this command asks for confirmation and gives you the option of enabling it; if you enable the command, confirmation will no longer be required for it. See section Disabling Commands.

Two-Column Editing

Two-column mode lets you conveniently edit two side-by-side columns of text. It uses two side-by-side windows, each showing its own buffer.

There are three ways to enter two-column mode:

f2 2 or C-x 6 2

Enter two-column mode with the current buffer on the left, and on the right, a buffer whose name is based on the current buffer's name (`2C-two-columns`). If the right-hand buffer doesn't already exist, it starts out empty; the current buffer's contents are not changed.

This command is appropriate when the current buffer is empty or contains just one column and you want to add another column.

f2 s or C-x 6 s

Split the current buffer, which contains two-column text, into two buffers, and display them side by side (`2C-split`). The current buffer becomes the left-hand buffer, but the text in the

right-hand column is moved into the right-hand buffer. The current column specifies the split point. Splitting starts with the current line and continues to the end of the buffer.

This command is appropriate when you have a buffer that already contains two-column text, and you wish to separate the columns temporarily.

f2 b buffer RET

C-x 6 b buffer RET

Enter two-column mode using the current buffer as the left-hand buffer, and using buffer buffer as the right-hand buffer (`2C-associate-buffer`).

f2 s or C-x 6 s looks for a column separator which is a string that appears on each line between the two columns. You can specify the width of the separator with a numeric argument to f2 s; that many characters, before point, constitute the separator string. By default, the width is 1, so the column separator is the character before point.

When a line has the separator at the proper place, f2 s puts the text after the separator into the right-hand buffer, and deletes the separator. Lines that don't have the column separator at the proper place remain unsplit; they stay in the left-hand buffer, and the right-hand buffer gets an empty line to correspond. (This is the way to write a line which "spans both columns while in two-column mode": write it in the left-hand buffer, and put an empty line in the right-hand buffer.)

The command C-x 6 RET or f2 RET (`2C-newline`) inserts a newline in each of the two buffers at corresponding positions. This is the easiest way to add a new line to the two-column text while editing it in split buffers.

When you have edited both buffers as you wish, merge them with f2 1 or C-x 6 1 (`2C-merge`). This copies the text from the right-hand buffer as a second column in the other buffer. To go back to two-column editing, use f2 s.

Use f2 d or C-x 6 d to disassociate the two buffers, leaving each as it stands (`2C-dissociate`). If the other buffer, the one not current when you type f2 d, is empty, f2 d kills it.

Editing Binary Files

There is a special major mode for editing binary files: Hexl mode. To use it, use M-x hexl-find-file instead of C-x C-f to visit the file. This command converts the file's contents to hexadecimal and lets you edit the translation. When you save the file, it is converted automatically back to binary.

You can also use M-x hexl-mode to translate an existing buffer into hex. This is useful if you visit a file normally and then discover it is a binary file.

Ordinary text characters overwrite in Hexl mode. This is to reduce the risk of accidentally spoiling the alignment of data in the file. There are special commands for insertion. Here is a list of the commands of Hexl mode:

C-M-d

Insert a byte with a code typed in decimal.

C-M-o
: Insert a byte with a code typed in octal.

C-M-x
: Insert a byte with a code typed in hex.

C-x [
: Move to the beginning of a 1k-byte "page".

C-x]
: Move to the end of a 1k-byte "page".

M-g
: Move to an address specified in hex.

M-j
: Move to an address specified in decimal.

C-c C-c
: Leave Hexl mode, going back to the major mode this buffer had before you invoked `hexl-mode`.

Saving Emacs Sessions

You can use the Desktop library to save the state of Emacs from one session to another. Saving the state means that Emacs starts up with the same set of buffers, major modes, buffer positions, and so on that the previous Emacs session had.

To use Desktop, you should first add these lines at the end of your `` `.emacs' `` file:

```
(load "desktop")
(desktop-load-default)
(desktop-read)
```

The first time you save the state of the Emacs session, you must do it manually, with the command M-x desktop-save. Once you have done that, exiting Emacs will save the state again--not only the present Emacs session, but also subsequent sessions. You can also save the state at any time, without exiting Emacs, by typing M-x desktop-save again.

In order for Emacs to recover the state from a previous session, you must start it with the same current directory as you used when you started the previous session.

The variable `desktop-files-not-to-save` controls which files are excluded from state saving. Its value is a regular expression that matches the files to exclude. By default, remote (ftp-accessed) files are excluded; this is because visiting them again in the subsequent session would be slow. If you want to include these files in state saving, set `desktop-files-not-to-save` to `"^$"`.

Recursive Editing Levels

A recursive edit is a situation in which you are using Emacs commands to perform arbitrary editing while in the middle of another Emacs command. For example, when you type C-r inside of a `query-replace`, you enter a recursive edit in which you can change the current buffer. On exiting from the recursive edit, you go back to the `query-replace`.

Exiting the recursive edit means returning to the unfinished command, which continues execution. The command to exit is C-M-c (`exit-recursive-edit`).

You can also abort the recursive edit. This is like exiting, but also quits the unfinished command immediately. Use the command C-] (`abort-recursive-edit`) to do this. See section Quitting and Aborting.

The mode line shows you when you are in a recursive edit by displaying square brackets around the parentheses that always surround the major and minor mode names. Every window's mode line shows this, in the same way, since being in a recursive edit is true of Emacs as a whole rather than any particular window or buffer.

It is possible to be in recursive edits within recursive edits. For example, after typing C-r in a `query-replace`, you may type a command that enters the debugger. This begins a recursive editing level for the debugger, within the recursive editing level for C-r. Mode lines display a pair of square brackets for each recursive editing level currently in progress.

Exiting the inner recursive edit (such as, with the debugger c command) resumes the command running in the next level up. When that command finishes, you can then use C-M-c to exit another recursive editing level, and so on. Exiting applies to the innermost level only. Aborting also gets out of only one level of recursive edit; it returns immediately to the command level of the previous recursive edit. If you wish, you can then abort the next recursive editing level.

Alternatively, the command M-x top-level aborts all levels of recursive edits, returning immediately to the top level command reader.

The text being edited inside the recursive edit need not be the same text that you were editing at top level. It depends on what the recursive edit is for. If the command that invokes the recursive edit selects a different buffer first, that is the buffer you will edit recursively. In any case, you can switch buffers within the recursive edit in the normal manner (as long as the buffer-switching keys have not been rebound). You could probably do all the rest of your editing inside the recursive edit, visiting files and all. But this could have surprising effects (such as stack overflow) from time to time. So remember to exit or abort the recursive edit when you no longer need it.

In general, we try to minimize the use of recursive editing levels in GNU Emacs. This is because they constrain you to "go back" in a particular order--from the innermost level toward the top level. When possible, we present different activities in separate buffers so that you can switch between them as you please. Some commands switch to a new major mode which provides a command to switch back. These approaches give you more flexibility to go back to unfinished tasks in the order you choose.

Emulation

GNU Emacs can be programmed to emulate (more or less) most other editors. Standard facilities can emulate these:

EDT (DEC VMS editor)

Turn on EDT emulation with M-x edt-emulation-on. M-x edt-emulation-off restores normal Emacs command bindings.

Most of the EDT emulation commands are keypad keys, and most standard Emacs key bindings are still available. The EDT emulation rebindings are done in the global keymap, so there is no problem switching buffers or major modes while in EDT emulation.

vi (Berkeley editor)

Viper is the newest emulator for vi. It implements several levels of emulation; level 1 is closest to vi itself, while level 5 departs somewhat from strict emulation to take advantage of the capabilities of Emacs. To invoke Viper, type M-x viper-mode; it will guide you the rest of the way and ask for the emulation level.

vi (another emulator)

M-x vi-mode enters a major mode that replaces the previously established major mode. All of the vi commands that, in real vi, enter "input" mode are programmed instead to return to the previous major mode. Thus, ordinary Emacs serves as vi's "input" mode.

Because vi emulation works through major modes, it does not work to switch buffers during emulation. Return to normal Emacs first.

If you plan to use vi emulation much, you probably want to bind a key to the `vi-mode` command.

vi (alternate emulator)

M-x vip-mode invokes another vi emulator, said to resemble real vi more thoroughly than M-x vi-mode. "Input" mode in this emulator is changed from ordinary Emacs so you can use ESC to go back to emulated vi command mode. To get from emulated vi command mode back to ordinary Emacs, type C-z.

This emulation does not work through major modes, and it is possible to switch buffers in various ways within the emulator. It is not so necessary to assign a key to the command `vip-mode` as it is with `vi-mode` because terminating insert mode does not use it.

For full information, see the long comment at the beginning of the source file, which is `` `lisp/vip.el' `` in the Emacs distribution.

Dissociated Press

M-x dissociated-press is a command for scrambling a file of text either word by word or character by character. Starting from a buffer of straight English, it produces extremely amusing output. The input comes from the current Emacs buffer. Dissociated Press writes its output in a buffer named `*Dissociation*', and redisplays that buffer after every couple of lines (approximately) so you can read

the output as it comes out.

Dissociated Press asks every so often whether to continue generating output. Answer n to stop it. You can also stop at any time by typing C-g. The dissociation output remains in the `*Dissociation*' buffer for you to copy elsewhere if you wish.

Dissociated Press operates by jumping at random from one point in the buffer to another. In order to produce plausible output rather than gibberish, it insists on a certain amount of overlap between the end of one run of consecutive words or characters and the start of the next. That is, if it has just printed out `president' and then decides to jump to a different point in the file, it might spot the `ent' in `pentagon' and continue from there, producing `presidentagon'.(4) Long sample texts produce the best results.

A positive argument to M-x dissociated-press tells it to operate character by character, and specifies the number of overlap characters. A negative argument tells it to operate word by word and specifies the number of overlap words. In this mode, whole words are treated as the elements to be permuted, rather than characters. No argument is equivalent to an argument of two. For your againformation, the output goes only into the buffer `*Dissociation*'. The buffer you start with is not changed.

Dissociated Press produces nearly the same results as a Markov chain based on a frequency table constructed from the sample text. It is, however, an independent, ignoriginal invention. Dissociated Press techniquitously copies several consecutive characters from the sample between random choices, whereas a Markov chain would choose randomly for each word or character. This makes for more plausible sounding results, and runs faster.

It is a mustatement that too much use of Dissociated Press can be a developediment to your real work. Sometimes to the point of outragedy. And keep dissociwords out of your documentation, if you want it to be well userenced and properbose. Have fun. Your buggestions are welcome.

Other Amusements

If you are a little bit bored, you can try M-x hanoi. If you are considerably bored, give it a numeric argument. If you are very very bored, try an argument of 9. Sit back and watch.

If you want a little more personal involvement, try M-x gomoku, which plays the game Go Moku with you.

M-x blackbox and M-x mpuz are two kinds of puzzles. `blackbox` challenges you to determine the location of objects inside a box by tomography. `mpuz` displays a multiplication puzzle with letters standing for digits in a code that you must guess--to guess a value, type a letter and then the digit you think it stands for.

M-x dunnet runs an adventure-style exploration game, which is a bigger sort of puzzle.

When you are frustrated, try the famous Eliza program. Just do M-x doctor. End each input by typing RET twice.

When you are feeling strange, type M-x yow.

Go to the previous, next section.

Go to the previous, next section.

Customization

This chapter talks about various topics relevant to adapting the behavior of Emacs in minor ways. See The Emacs Lisp Reference Manual for how to make more far-reaching changes.

All kinds of customization affect only the particular Emacs session that you do them in. They are completely lost when you kill the Emacs session, and have no effect on other Emacs sessions you may run at the same time or later. The only way an Emacs session can affect anything outside of it is by writing a file; in particular, the only way to make a customization `permanent' is to put something in your `.emacs' file or other appropriate file to do the customization in each session. See section The Init File, `~/.emacs'.

Minor Modes

Minor modes are optional features which you can turn on or off. For example, Auto Fill mode is a minor mode in which SPC breaks lines between words as you type. All the minor modes are independent of each other and of the selected major mode. Most minor modes say in the mode line when they are on; for example, `Fill' in the mode line means that Auto Fill mode is on.

Append `-mode` to the name of a minor mode to get the name of a command function that turns the mode on or off. Thus, the command to enable or disable Auto Fill mode is called M-x auto-fill-mode. These commands are usually invoked with M-x, but you can bind keys to them if you wish. With no argument, the function turns the mode on if it was off and off if it was on. This is known as toggling. A positive argument always turns the mode on, and an explicit zero argument or a negative argument always turns it off.

Enabling or disabling some minor modes applies only to the current buffer; each buffer is independent of the other buffers. Therefore, you can enable the mode in particular buffers and disable it in others. The per-buffer minor modes include Auto Fill mode, Auto Save mode, Font-Lock mode, ISO Accents mode, Outline minor mode, Overwrite mode, and Binary Overwrite mode.

Auto Fill mode allows you to enter filled text without breaking lines explicitly. Emacs inserts newlines as necessary to prevent lines from becoming too long. See section Filling Text.

Auto Save mode causes the contents of a buffer to be saved periodically to reduce the amount of work you can lose in case of a system crash. See section Auto-Saving: Protection Against Disasters.

Enriched mode enables editing and saving of formatted text. See section Editing Formatted Text.

Font-Lock mode automatically highlights certain textual units found in programs, such as comments, strings, and function names being defined. This requires a window system that can display multiple fonts. See section Using Multiple Typefaces.

ISO Accents mode makes the characters ``', `'', `'"', `^', `/' and `~' combine with the following letter, to produce an accented letter in the ISO Latin-1 character set. See section European Character Set Display.

Outline minor mode provides the same facilities as the major mode called Outline mode; but since it is a minor mode instead, you can combine it with any major mode. See section Outline Mode.

Overwrite mode causes ordinary printing characters to replace existing text instead of shoving it to the right. For example, if point is in front of the `B' in `FOOBAR', then in Overwrite mode typing a G changes it to `FOOGAR', instead of producing it `FOOGBAR' as usual. Binary Overwrite mode is a variant of Overwrite mode for editing binary files; it treats newlines and tabs like other characters, so that they overwrite other characters and can be overwritten by them.

The following minor modes normally apply to all buffers at once. Since each is enabled or disabled by the value of a variable, you *can* set them differently for particular buffers, by explicitly making the corresponding variables local in those buffers. See section Local Variables.

Abbrev mode allows you to define abbreviations that automatically expand as you type them. For example, `amd' might expand to `abbrev mode'. See section Abbrevs, for full information.

Icomplete mode displays an indication of available completions when you are in the minibuffer and completion is active. See section Completion Options.

Line Number mode enables continuous display in the mode line of the line number of point. See section The Mode Line.

Resize-Minibuffer mode makes the minibuffer expand as necessary to hold the text that you put in it. See section Editing in the Minibuffer.

Scroll Bar mode gives each window a scroll bar (see section Scroll Bars). Menu Bar mode gives each frame a menu bar (see section Menu Bars). Both of these modes are enabled by default when you use the X Window System.

In Transient Mark mode, every change in the buffer contents "deactivates" the mark, so that commands that operate on the region will get an error. This means you must either set the mark, or explicitly "reactivate" it, before each command that uses the region. The advantage of Transient Mark mode is that Emacs can display the region highlighted (currently only when using X). See section Setting the Mark.

For most minor modes, the command name is also the name of a variable which directly controls the mode. The mode is enabled whenever this variable's value is non-`nil`, and the minor mode command works by setting the variable. For example, the command `outline-minor-mode` works by setting the value of `outline-minor-mode` as a variable; it is this variable that directly turns Outline minor mode on and off. To check whether a given minor mode works this way, use C-h v to ask for documentation on the variable name.

These minor mode variables provide a good way for Lisp programs to turn minor modes on and off; they are also useful in a file's local variables list. But please think twice before setting minor modes with a local variables list, because most minor modes are matter of user preference--other users editing the same file might not want the same minor modes you prefer.

Variables

A variable is a Lisp symbol which has a value. The symbol's name is also called the name of the variable. A variable name can contain any characters that can appear in a file, but conventionally variable names consist of words separated by hyphens. A variable can have a documentation string which describes what kind of value it should have and how the value will be used.

Lisp allows any variable to have any kind of value, but most variables that Emacs uses require a value of a certain type. Often the value should always be a string, or should always be a number. Sometimes we say that a certain feature is turned on if a variable is "non-`nil`," meaning that if the variable's value is `nil`, the feature is off, but the feature is on for *any* other value. The conventional value to use to turn on the feature--since you have to pick one particular value when you set the variable--is `t`.

Emacs uses many Lisp variables for internal record keeping, as any Lisp program must, but the most interesting variables for you are the ones that exist for the sake of customization. Emacs does not (usually) change the values of these variables; instead, you set the values, and thereby alter and control the behavior of certain Emacs commands. These variables are called user options. Most user options are documented in this manual, and appear in the Variable Index (see section Variable Index).

One example of a variable which is a user option is `fill-column`, which specifies the position of the right margin (as a number of characters from the left margin) to be used by the fill commands (see section Filling Text).

Examining and Setting Variables

C-h v var RET

Display the value and documentation of variable var (`describe-variable`).

M-x set-variable RET var RET value RET

Change the value of variable var to value.

To examine the value of a single variable, use C-h v (`describe-variable`), which reads a variable name using the minibuffer, with completion. It displays both the value and the documentation of the variable. For example,

```
C-h v fill-column RET
```

displays something like this:

```
fill-column's value is 75

Documentation:
*Column beyond which automatic line-wrapping should happen.
Automatically becomes buffer-local when set in any fashion.
```

The star at the beginning of the documentation indicates that this variable is a user option. C-h v is not

restricted to user options; it allows any variable name.

The most convenient way to set a specific user option is with M-x set-variable. This reads the variable name with the minibuffer (with completion), and then reads a Lisp expression for the new value using the minibuffer a second time. For example,

```
M-x set-variable RET fill-column RET 75 RET
```

sets `fill-column` to 75.

M-x set-variable is limited to user option variables. You can set any variable with a Lisp expression using the function `setq`. Here's how to use it to set `fill-column`:

```
(setq fill-column 75)
```

Setting variables, like all means of customizing Emacs except where otherwise stated, affects only the current Emacs session.

Editing Variable Values

These two functions make it easy to display all the Emacs user option variables, and to change some of them if you wish.

M-x list-options

: Display a buffer listing names, values and documentation of all options.

M-x edit-options

: Change user option values by editing a list of user option variables.

M-x list-options displays a list of all Emacs option variables, in an Emacs buffer named `*List Options*'. Each user option is shown with its documentation and its current value. Here is what a portion of it might look like:

```
;; exec-path:
("." "/usr/local/bin" "/usr/ucb" "/bin" "/usr/bin" "/u2/emacs/etc")
*List of directories to search programs to run in subprocesses.
Each element is a string (directory name)
or nil (try the default directory).
;;
;; fill-column:
75
*Column beyond which automatic line-wrapping should happen.
Automatically becomes buffer-local when set in any fashion.
;;
```

M-x edit-options goes one step further and immediately selects the `*List Options*' buffer; this buffer uses the major mode Options mode, which provides commands that allow you to point at a user option variable and change its value:

s

Set the variable point is in or near to a new value read using the minibuffer.

x

Toggle the variable point is in or near: if the value was `nil`, it becomes `t`; otherwise it becomes `nil`.

1

Set the variable point is in or near to `t`.

0

Set the variable point is in or near to `nil`.

n

p

Move to the next or previous user option.

Any changes take effect immediately, and last until you exit from Emacs.

Hooks

A hook is a variable where you can store a function or functions to be called on a particular occasion by an existing program. Emacs provides a number of hooks for the sake of customization.

Most of the hooks in Emacs are normal hooks. These variables contain lists of functions to be called with no arguments. The reason most hooks are normal hooks is so that you can use them in a uniform way. Every variable in Emacs whose name ends in `-hook' is a normal hook.

Most major modes run hooks as the last step of initialization. This makes it easy for a user to customize the behavior of the mode, by overriding the local variable assignments already made by the mode. But hooks may also be used in other contexts. For example, the hook `suspend-hook` runs just before Emacs suspends itself (see section Exiting Emacs).

The recommended way to add a hook function to a normal hook is by calling `add-hook`. You can use any valid Lisp function as the hook function. For example, here's how to set up a hook to turn on Auto Fill mode when entering Text mode and other modes based on Text mode:

```
(add-hook 'text-mode-hook 'turn-on-auto-fill)
```

The next example shows how to use a hook to customize the indentation of C code. (People often have strong personal preferences for one format compared to another.) Here the hook function is an anonymous lambda expression.

```
(setq my-c-style
  '((c-comment-only-line-offset . 4)
    (c-cleanup-list . (scope-operator
                       empty-defun-braces
                       defun-close-semi))
```


```
    (c-offsets-alist . ((arglist-close . c-lineup-arglist)
                        (substatement-open . 0))))

(add-hook 'c-mode-common-hook
  (function (lambda ()
    (c-add-style "my-style" my-c-style t))))
```

It is best to design your hook functions so that the order in which they are executed does not matter. Any dependence on the order is "asking for trouble." However, the order is predictable: the most recently added hook functions are executed first.

Local Variables

M-x make-local-variable RET var RET

Make variable var have a local value in the current buffer.

M-x kill-local-variable RET var RET

Make variable var use its global value in the current buffer.

M-x make-variable-buffer-local RET var RET

Mark variable var so that setting it will make it local to the buffer that is current at that time.

Almost any variable can be made local to a specific Emacs buffer. This means that its value in that buffer is independent of its value in other buffers. A few variables are always local in every buffer. Every other Emacs variable has a global value which is in effect in all buffers that have not made the variable local.

M-x make-local-variable reads the name of a variable and makes it local to the current buffer. Further changes in this buffer will not affect others, and further changes in the global value will not affect this buffer.

M-x make-variable-buffer-local reads the name of a variable and changes the future behavior of the variable so that it will become local automatically when it is set. More precisely, once a variable has been marked in this way, the usual ways of setting the variable automatically do `make-local-variable` first. We call such variables per-buffer variables.

Major modes (see section Major Modes) always make variables local to the buffer before setting the variables. This is why changing major modes in one buffer has no effect on other buffers. Minor modes also work by setting variables--normally, each minor mode has one controlling variable which is non-`nil` when the mode is enabled (see section Minor Modes). For most minor modes, the controlling variable is per buffer.

Emacs contains a number of variables that are always per-buffer. These include `abbrev-mode`, `auto-fill-function`, `case-fold-search`, `comment-column`, `ctl-arrow`, `fill-column`, `fill-prefix`, `indent-tabs-mode`, `left-margin`, `mode-line-format`, `overwrite-mode`, `selective-display-ellipses`, `selective-display`, `tab-width`, and `truncate-lines`. Some other variables are always local in every buffer, but they are used for internal purposes.

A few variables cannot be local to a buffer because they are always local to each display instead (See section Multiple Displays). If you try to make one of these variables buffer-local, you'll get an error message.

M-x kill-local-variable reads the name of a variable and makes it cease to be local to the current buffer. The global value of the variable henceforth is in effect in this buffer. Setting the major mode kills all the local variables of the buffer except for a few variables specially marked as permanent locals.

To set the global value of a variable, regardless of whether the variable has a local value in the current buffer, you can use the Lisp construct `setq-default`. This construct is used just like `setq`, but it sets variables' global values instead of their local values (if any). When the current buffer does have a local value, the new global value may not be visible until you switch to another buffer. Here is an example:

```
(setq-default fill-column 75)
```

`setq-default` is the only way to set the global value of a variable that has been marked with `make-variable-buffer-local`.

Lisp programs can use `default-value` to look at a variable's default value. This function takes a symbol as argument and returns its default value. The argument is evaluated; usually you must quote it explicitly. For example, here's how to obtain the default value of `fill-column`:

```
(default-value 'fill-column)
```

Local Variables in Files

A file can specify local variable values for use when you edit the file with Emacs. Visiting the file checks for local variables specifications; it automatically makes these variables local to the buffer, and sets them to the values specified in the file.

There are two ways to specify local variable values: in the first line, or with a local variables list. Here's how to specify them in the first line:

```
-*- mode: modename; var: value; ... -*-
```

You can specify any number of variables/value pairs in this way, each pair with a colon and semicolon as shown above. `mode: modename;` specifies the major mode; this should come first in the line. The values are not evaluated; they are used literally. Here is an example that specifies Lisp mode and sets two variables with numeric values:

```
;; -*-mode: Lisp; fill-column: 75; comment-column: 50; -*-
```

A local variables list goes near the end of the file, in the last page. (It is often best to put it on a page by itself.) The local variables list starts with a line containing the string `Local Variables:', and ends with a line containing the string `End:'. In between come the variable names and values, one set per line, as `variable: value'. The values are not evaluated; they are used literally. If a file has both a local variables list and a `-*-' line, Emacs processes *everything* in the `-*-' line first, and *everything* in the local variables

list afterward.

Here is an example of a local variables list:

```
;;; Local Variables: ***
;;; mode:lisp ***
;;; comment-column:0 ***
;;; comment-start: ";;; "  ***
;;; comment-end:"***" ***
;;; End: ***
```

As you see, each line starts with the prefix `;;; ' and each line ends with the suffix ` ***'. Emacs recognizes these as the prefix and suffix based on the first line of the list, by finding them surrounding the magic string `Local Variables:'; then it automatically discards them from the other lines of the list.

The usual reason for using a prefix and/or suffix is to embed the local variables list in a comment, so it won't confuse other programs that the file is intended as input for. The example above is for a language where comment lines start with `;;; ' and end with `***'; the local values for `comment-start` and `comment-end` customize the rest of Emacs for this unusual syntax. Don't use a prefix (or a suffix) if you don't need one.

Two "variable names" have special meanings in a local variables list: a value for the variable `mode` really sets the major mode, and a value for the variable `eval` is simply evaluated as an expression and the value is ignored. `mode` and `eval` are not real variables; setting variables named `mode` and `eval` in any other context has no special meaning. If `mode` is used to set a major mode, it should be the first "variable" in the list.

You can use the `mode` "variable" to set minor modes as well as major modes; in fact, you can use it more than once, first to set the major mode and then to set minor modes which are specific to particular buffers. But most minor modes should not be specified in the file in any fashion, because they represent user preferences. For example, you should not try to specify Auto Fill mode with file local variables, because whether to use Auto Fill mode for editing a particular kind of text is a matter of personal taste, not an aspect of the format of the text.

The start of the local variables list must be no more than 3000 characters from the end of the file, and must be in the last page if the file is divided into pages. Otherwise, Emacs will not notice it is there. The purpose of this rule is so that a stray `Local Variables:' not in the last page does not confuse Emacs, and so that visiting a long file that is all one page and has no local variables list need not take the time to search the whole file.

You may be tempted to try to turn on Auto Fill mode with a local variable list. That is a mistake. The choice of Auto Fill mode or not is a matter of individual taste, not a matter of the contents of particular files. If you want to use Auto Fill, set up major mode hooks with your `.emacs' file to turn it on (when appropriate) for you alone (see section The Init File, `~/.emacs'). Don't try to use a local variable list that would impose your taste on everyone.

The variable `enable-local-variables` controls whether to process local variables lists, and thus gives you a chance to override them. Its default value is `t`, which means do process local variables lists.

If you set the value to `nil`, Emacs simply ignores local variables lists. Any other value says to query about each local variables list, showing you the local variables list to consider.

The `eval` "variable", and certain actual variables, create a special risk; when you visit someone else's file, local variable specifications for these could affect your Emacs in arbitrary ways. Therefore, the option `enable-local-eval` controls whether Emacs processes `eval` variables, as well variables with names that end in `-hook', `-hooks', `-function' or `-functions', and certain other variables. The three possibilities for the option's value are `t`, `nil`, and anything else, just as for `enable-local-variables`. The default is `maybe`, which is neither `t` nor `nil`, so normally Emacs does ask for confirmation about file settings for these variables.

Use the command `normal-mode` to reset the local variables and major mode of a buffer according to the file name and contents, including the local variables list if any. See section How Major Modes are Chosen.

Keyboard Macros

A keyboard macro is a command defined by the user to stand for another sequence of keys. For example, if you discover that you are about to type C-n C-d forty times, you can speed your work by defining a keyboard macro to do C-n C-d and calling it with a repeat count of forty.

C-x (
: Start defining a keyboard macro (`start-kbd-macro`).

C-x)
: End the definition of a keyboard macro (`end-kbd-macro`).

C-x e
: Execute the most recent keyboard macro (`call-last-kbd-macro`).

C-u C-x (
: Re-execute last keyboard macro, then add more keys to its definition.

C-x q
: When this point is reached during macro execution, ask for confirmation (`kbd-macro-query`).

M-x name-last-kbd-macro
: Give a command name (for the duration of the session) to the most recently defined keyboard macro.

M-x insert-kbd-macro
: Insert in the buffer a keyboard macro's definition, as Lisp code.

C-x C-k
: Edit a previously defined keyboard macro (`edit-kbd-macro`).

M-x apply-macro-to-region-lines
: Run the last keyboard macro on each complete line in the region.

Keyboard macros differ from ordinary Emacs commands in that they are written in the Emacs command

language rather than in Lisp. This makes it easier for the novice to write them, and makes them more convenient as temporary hacks. However, the Emacs command language is not powerful enough as a programming language to be useful for writing anything intelligent or general. For such things, Lisp must be used.

You define a keyboard macro while executing the commands which are the definition. Put differently, as you define a keyboard macro, the definition is being executed for the first time. This way, you can see what the effects of your commands are, so that you don't have to figure them out in your head. When you are finished, the keyboard macro is defined and also has been, in effect, executed once. You can then do the whole thing over again by invoking the macro.

Basic Use

To start defining a keyboard macro, type the C-x (command (`start-kbd-macro`). From then on, your keys continue to be executed, but also become part of the definition of the macro. `Def' appears in the mode line to remind you of what is going on. When you are finished, the C-x ) command (`end-kbd-macro`) terminates the definition (without becoming part of it!). For example,

```
C-x ( M-f foo C-x )
```

defines a macro to move forward a word and then insert `foo'.

The macro thus defined can be invoked again with the C-x e command (`call-last-kbd-macro`), which may be given a repeat count as a numeric argument to execute the macro many times. C-x) can also be given a repeat count as an argument, in which case it repeats the macro that many times right after defining it, but defining the macro counts as the first repetition (since it is executed as you define it). Therefore, giving C-x) an argument of 4 executes the macro immediately 3 additional times. An argument of zero to C-x e or C-x) means repeat the macro indefinitely (until it gets an error or you type C-g).

If you wish to repeat an operation at regularly spaced places in the text, define a macro and include as part of the macro the commands to move to the next place you want to use it. For example, if you want to change each line, you should position point at the start of a line, and define a macro to change that line and leave point at the start of the next line. Then repeating the macro will operate on successive lines.

After you have terminated the definition of a keyboard macro, you can add to the end of its definition by typing C-u C-x (. This is equivalent to plain C-x (followed by retyping the whole definition so far. As a consequence it re-executes the macro as previously defined.

You can use function keys in a keyboard macro, just like keyboard keys. You can even use mouse events, but be careful about that: when the macro replays the mouse event, it uses the original mouse position of that event, the position that the mouse had while you were defining the macro. The effect of this may be hard to predict. (Using the current mouse position would be even less predictable.)

One thing that doesn't always work well in a keyboard macro is the command C-M-c (`exit-recursive-edit`). When this command exits a recursive edit that started within the macro, it works as you'd expect. But if it exits a recursive edit that started before you invoked the keyboard macro, it also necessarily exits the keyboard macro as part of the process.

You can edit a keyboard macro already defined by typing C-x C-k (`edit-kbd-macro`). Follow th with the keyboard input that you would use to invoke the macro---C-x e or M-x name or some other key sequence. This formats the macro definition in a buffer and enters a specialized major mode for editing it. Type C-h m once in that buffer to display details of how to edit the macro. When you are finished editing, type C-c C-c.

The command M-x apply-macro-to-region-lines repeats the last defined keyboard macro on each complete line within the current region. It does this line by line, by moving point to the beginning of the line and then executing the macro.

Naming and Saving Keyboard Macros

If you wish to save a keyboard macro for longer than until you define the next one, you must give it a name using M-x name-last-kbd-macro. This reads a name as an argument using the minibuffer and defines that name to execute the macro. The macro name is a Lisp symbol, and defining it in this way makes it a valid command name for calling with M-x or for binding a key to with `global-set-key` (see section Keymaps). If you specify a name that has a prior definition other than another keyboard macro, an error message is printed and nothing is changed.

Once a macro has a command name, you can save its definition in a file. Then it can be used in another editing session. First, visit the file you want to save the definition in. Then use this command:

```
M-x insert-kbd-macro RET macroname RET
```

This inserts some Lisp code that, when executed later, will define the same macro with the same definition it has now. (You need not understand Lisp code to do this, because `insert-kbd-macro` writes the Lisp code for you.) Then save the file. You can load the file later with `load-file` (see section Libraries of Lisp Code for Emacs). If the file you save in is your init file `` `~/.emacs' `` (see section The Init File, `` `~/.emacs' ``) then the macro will be defined each time you run Emacs.

If you give `insert-kbd-macro` a numeric argument, it makes additional Lisp code to record the keys (if any) that you have bound to the keyboard macro, so that the macro will be reassigned the same keys when you load the file.

Executing Macros with Variations

Using C-x q (`kbd-macro-query`), you can get an effect similar to that of `query-replace`, where the macro asks you each time around whether to make a change. While defining the macro, type C-x q at the point where you want the query to occur. During macro definition, the C-x q does nothing, but when you run the macro later, C-x q asks you interactively whether to continue.

The valid responses when C-x q asks are SPC (or y), DEL (or n), RET (or q), C-l and C-r. The answers are the same as in `query-replace`, though not all of the `query-replace` options are meaningful.

These responses include SPC to continue, and DEL to skip the remainder of this repetition of the macro and start right away with the next repetition. RET means to skip the remainder of this repetition and

cancel further repetitions. C-l redraws the screen and asks you again for a character to say what to do.

C-r enters a recursive editing level, in which you can perform editing which is not part of the macro. When you exit the recursive edit using C-M-c, you are asked again how to continue with the keyboard macro. If you type a SPC at this time, the rest of the macro definition is executed. It is up to you to leave point and the text in a state such that the rest of the macro will do what you want.

C-u C-x q, which is C-x q with a numeric argument, performs a completely different function. It enters a recursive edit reading input from the keyboard, both when you type it during the definition of the macro, and when it is executed from the macro. During definition, the editing you do inside the recursive edit does not become part of the macro. During macro execution, the recursive edit gives you a chance to do some particularized editing on each repetition. See section Recursive Editing Levels.

Customizing Key Bindings

This section describes key bindings which map keys to commands, and the keymaps which record key bindings. It also explains how to customize key bindings.

Recall that a command is a Lisp function whose definition provides for interactive use. Like every Lisp function, a command has a function name which usually consists of lower case letters and hyphens.

Keymaps

The bindings between key sequences and command functions are recorded in data structures called keymaps. Emacs has many of these, each used on particular occasions.

Recall that a key sequence (key, for short) is a sequence of input events that have a meaning as a unit. Input events include characters, function keys and mouse buttons--all the inputs that you can send to the computer with your terminal. A key sequence gets its meaning from its binding, which says what command it runs. The function of keymaps is to record these bindings.

The global keymap is the most important keymap because it is always in effect. The global keymap defines keys for Fundamental mode; most of these definitions are common to most or all major modes. Each major or minor mode can have its own keymap which overrides the global definitions of some keys.

For example, a self-inserting character such as g is self-inserting because the global keymap binds it to the command `self-insert-command`. The standard Emacs editing characters such as C-a also get their standard meanings from the global keymap. Commands to rebind keys, such as M-x global-set-key, actually work by storing the new binding in the proper place in the global map. See section Changing Key Bindings Interactively.

Meta characters work differently; Emacs translates each Meta character into a pair of characters starting with ESC. When you type the character M-a in a key sequence, Emacs replaces it with ESC a. A meta key comes in as a single input event, but becomes two events for purposes of key bindings. The reason for this is historical, and we might change it someday.

Most modern keyboards have function keys as well as character keys. Function keys send input events just as character keys do, and keymaps can have bindings for them.

On many terminals, typing a function key actually sends the computer a sequence of characters; the precise details of the sequence depends on which function key and on the model of terminal you are using. (Often the sequence starts with ESC [.) If Emacs understands your terminal type properly, it recognizes the character sequences forming function keys wherever they occur in a key sequence (not just at the beginning). Thus, for most purposes, you can pretend the function keys reach Emacs directly and ignore their encoding as character sequences.

Mouse buttons also produce input events. These events come with other data--the window and position where you pressed or released the button, and a time stamp. But only the choice of button matters for key bindings; the other data matters only if a command looks at it. (Commands designed for mouse invocation usually do look at the other data.)

A keymap records definitions for single events. Interpreting a key sequence of multiple events involves a chain of keymaps. The first keymap gives a definition for the first event; this definition is another keymap, which is used to look up the second event in the sequence, and so on.

Key sequences can mix function keys and characters. For example, C-x SELECT makes sense. If you make SELECT a prefix key, then SELECT C-n makes sense. You can even mix mouse events with keyboard events, but we recommend against it, because such sequences are inconvenient to type in.

As a user, you can redefine any key; but it might be best to stick to key sequences that consist of C-c followed by a letter. These keys are "reserved for users,", so they won't conflict with any properly designed Emacs extension. If you redefine some other key, your definition may be overridden by certain extensions or major modes which redefine the same key.

Prefix Keymaps

A prefix key such as C-x or ESC has its own keymap, which holds the definition for the event that immediately follows that prefix.

The definition of a prefix key is usually the keymap to use for looking up the following event. The definition can also be a Lisp symbol whose function definition is the following keymap; the effect is the same, but it provides a command name for the prefix key that can be used as a description of what the prefix key is for. Thus, the binding of C-x is the symbol `Ctl-X-Prefix`, whose function definition is the keymap for C-x commands. The definitions of C-c, C-x, C-h and ESC as prefix keys appear in the global map, so these prefix keys are always available.

Aside from ordinary prefix keys, there is a fictitious "prefix key" which represents the menu bar; see section 'Menu Bar' in The Emacs Lisp Reference Manual, for special information about menu bar key bindings. Mouse button events that invoke pop-up menus are also prefix keys; see section 'Menu Keymaps' in The Emacs Lisp Reference Manual, for more details.

Some prefix keymaps are stored in variables with names:

- `ctl-x-map` is the variable name for the map used for characters that follow C-x.
- `help-map` is for characters that follow C-h.

- `esc-map` is for characters that follow ESC. Thus, all Meta characters are actually defined by this map.
- `ctl-x-4-map` is for characters that follow C-x 4.
- `mode-specific-map` is for characters that follow C-c.

Local Keymaps

So far we have explained the ins and outs of the global map. Major modes customize Emacs by providing their own key bindings in local keymaps. For example, C mode overrides TAB to make it indent the current line for C code. Portions of text in the buffer can specify their own keymaps to substitute for the keymap of the buffer's major mode.

Minor modes can also have local keymaps. Whenever a minor mode is in effect, the definitions in its keymap override both the major mode's local keymap and the global keymap.

The local keymaps for Lisp mode and several other major modes always exist even when not in use. These are kept in variables named `lisp-mode-map` and so on. For major modes less often used, the local keymap is normally constructed only when the mode is used for the first time in a session. This is to save space. If you wish to change one of these keymaps, you must use the major mode's mode hook---see below.

All minor mode keymaps are created in advance. There is no way to defer their creation until the first time the minor mode is enabled.

A local keymap can locally redefine a key as a prefix key by defining it as a prefix keymap. If the key is also defined globally as a prefix, then its local and global definitions (both keymaps) effectively combine: both of them are used to look up the event that follows the prefix key. Thus, if the mode's local keymap defines C-c as another keymap, and that keymap defines C-z as a command, this provides a local meaning for C-c C-z. This does not affect other sequences that start with C-c; if those sequences don't have their own local bindings, their global bindings remain in effect.

Another way to think of this is that Emacs handles a multi-event key sequence by looking in several keymaps, one by one, for a binding of the whole key sequence. First it checks the minor mode keymaps for minor modes that are enabled, then it checks the major mode's keymap, and then it checks the global keymap. This is not precisely how key lookup works, but it's good enough for understanding ordinary circumstances.

To change the local bindings of a major mode, you must change the mode's local keymap. Normally you must wait until the first time the mode is used, because most major modes don't create their keymaps until then. If you want to specify something in your `` `~/.emacs' `` file to change a major mode's bindings, you must use the mode's mode hook to delay the change until the mode is first used.

For example, the command `texinfo-mode` to select Texinfo mode runs the hook `texinfo-mode-hook`. Here's how you can use the hook to add local bindings (not very useful, we admit) for C-c n and C-c p in Texinfo mode:

```
(add-hook 'texinfo-mode-hook
```


```
'(lambda ()
   (define-key texinfo-mode-map
               "\C-cp"
               'backward-paragraph)
   (define-key texinfo-mode-map
               "\C-cn"
               'forward-paragraph)
   ))
```

See section Hooks.

Minibuffer Keymaps

The minibuffer has its own set of local keymaps; they contain various completion and exit commands.

- `minibuffer-local-map` is used for ordinary input (no completion).
- `minibuffer-local-ns-map` is similar, except that SPC exits just like RET. This is used mainly for Mocklisp compatibility.
- `minibuffer-local-completion-map` is for permissive completion.
- `minibuffer-local-must-match-map` is for strict completion and for cautious completion.

Changing Key Bindings Interactively

The way to redefine an Emacs key is to change its entry in a keymap. You can change the global keymap, in which case the change is effective in all major modes (except those that have their own overriding local definitions for the same key). Or you can change the current buffer's local map, which affects all buffers using the same major mode.

M-x global-set-key RET key cmd RET
: Define key globally to run cmd.

M-x local-set-key RET key cmd RET
: Define key locally (in the major mode now in effect) to run cmd.

M-x global-unset-key RET key
: Make key undefined in the global map.

M-x local-unset-key RET key
: Make key undefined locally (in the major mode now in effect).

For example, suppose you like to execute commands in a subshell within an Emacs buffer, instead of suspending Emacs and executing commands in your login shell. Normally, C-z is bound to the function `suspend-emacs` (when not using the X Window System), but you can change C-z to invoke an interactive subshell within Emacs, by binding it to `shell` as follows:

```
M-x global-set-key RET C-z shell RET
```

`global-set-key` reads the command name after the key. After you press the key, a message like this

appears so that you can confirm that you are binding the key you want:

```
Set key C-z to command:
```

You can redefine function keys and mouse events in the same way; just type the function key or click the mouse when it's time to specify the key to rebind.

You can rebind a key that contains more than one event in the same way. Emacs keeps reading the key to rebind until it is a complete key (that is, not a prefix key). Thus, if you type C-f for key, that's the end; the minibuffer is entered immediately to read cmd. But if you type C-x, another character is read; if that is 4, another character is read, and so on. For example,

```
M-x global-set-key RET C-x 4 $ spell-other-window RET
```

redefines C-x 4 $ to run the (fictitious) command `spell-other-window`.

The two-character keys consisting of C-c followed by a letter are reserved for user customizations. Lisp programs are not supposed to define these keys, so the bindings you make for them will be available in all major modes and will never get in the way of anything.

You can remove the global definition of a key with `global-unset-key`. This makes the key undefined; if you type it, Emacs will just beep. Similarly, `local-unset-key` makes a key undefined in the current major mode keymap, which makes the global definition (or lack of one) come back into effect in that major mode.

If you have redefined (or undefined) a key and you subsequently wish to retract the change, undefining the key will not do the job--you need to redefine the key with its standard definition. To find the name of the standard definition of a key, go to a Fundamental mode buffer and use C-h c. The documentation of keys in this manual also lists their command names.

If you want to prevent yourself from invoking a command by mistake, it is better to disable the command than to undefine the key. A disabled command is less work to invoke when you really want to. See section Disabling Commands.

Rebinding Keys in Your Init File

If you have a set of key bindings that you like to use all the time, you can specify them in your `` `.emacs' `` file by using their Lisp syntax.

The simplest method for doing this works for ASCII characters and Meta-modified ASCII characters only. This method uses a string to represent the key sequence you want to rebind. For example, here's how to bind C-z to `shell`:

```
(global-set-key "\C-z" 'shell)
```

This example uses a string constant containing one character, C-z. The single-quote before the command name, `shell`, marks it as a constant symbol rather than a variable. If you omit the quote, Emacs would try to evaluate `shell` immediately as a variable. This probably causes an error; it certainly isn't what

you want.

Here is another example that binds a key sequence two characters long:

```
(global-set-key "\C-xl" 'make-symbolic-link)
```

When the key sequence includes function keys or mouse button events, or non-ASCII characters such as `C-=` or `H-a`, you must use the more general method of rebinding, which uses a vector to specify the key sequence.

The way to write a vector in Emacs Lisp is with square brackets around the vector elements. Use spaces to separate the elements. If an element is a symbol, simply write the symbol's name--no other delimiters or punctuation are needed. If a vector element is a character, write it as a Lisp character constant: `?' followed by the character as it would appear in a string.

Here are examples of using vectors to rebind C-= (a control character outside of ASCII), H-a (a Hyper character; ASCII doesn't have Hyper at all); f7 (a function key), and C-Mouse-1 (a keyboard-modified mouse button):

```
(global-set-key [?\C-=] 'make-symbolic-link)
(global-set-key [?\H-a] 'make-symbolic-link)
(global-set-key [f7] 'make-symbolic-link)
(global-set-key [C-mouse-1] 'make-symbolic-link)
```

You can use a vector for the simple cases too. Here's how to rewrite the first two examples, above, to use vectors:

```
(global-set-key [?\C-z] 'shell)

(global-set-key [?\C-x ?l] 'make-symbolic-link)
```

Rebinding Function Keys

Key sequences can contain function keys as well as ordinary characters. Just as Lisp characters (actually integers) represent keyboard characters, Lisp symbols represent function keys. If the function key has a word as its label, then that word is also the name of the corresponding Lisp symbol. Here are the conventional Lisp names for common function keys:

`left`, `up`, `right`, `down`

: Cursor arrow keys.

`begin`, `end`, `home`, `next`, `prior`

: Other cursor repositioning keys.

`select`, `print`, `execute`, `backtab`

`insert`, `undo`, `redo`, `clearline`

`insertline`, `deleteline`, `insertchar`, `deletechar`,

Miscellaneous function keys.

`f1`, `f2`, ... `f35`

Numbered function keys (across the top of the keyboard).

`kp-add`, `kp-subtract`, `kp-multiply`, `kp-divide`

`kp-backtab`, `kp-space`, `kp-tab`, `kp-enter`

`kp-separator`, `kp-decimal`, `kp-equal`

Keypad keys (to the right of the regular keyboard), with names or punctuation.

`kp-0`, `kp-1`, ... `kp-9`

Keypad keys with digits.

`kp-f1`, `kp-f2`, `kp-f3`, `kp-f4`

Keypad PF keys.

These names are conventional, but some systems (especially when using X windows) may use different names. To make certain what symbol is used for a given function key on your terminal, type C-h c followed by that key.

A key sequence which contains function key symbols (or anything but ASCII characters) must be a vector rather than a string. The vector syntax uses spaces between the elements, and square brackets around the whole vector. Thus, to bind function key `f1' to the command `rmail`, write the following:

```
(global-set-key [f1] 'rmail)
```

To bind the right-arrow key to the command `forward-char`, you can use this expression:

```
(global-set-key [right] 'forward-char)
```

This uses the Lisp syntax for a vector containing the symbol `right`. (This binding is present in Emacs by default.)

See section Rebinding Keys in Your Init File, for more information about using vectors for rebinding.

You can mix function keys and characters in a key sequence. This example binds C-x NEXT to the command `forward-page`.

```
(global-set-key [?\C-x next] 'forward-page)
```

where `?\C-x` is the Lisp character constant for the character C-x. The vector element `next` is a symbol and therefore does not take a question mark.

You can use the modifier keys CTRL, META, HYPER, SUPER, ALT and SHIFT with function keys. To represent these modifiers, add the strings `C-', `M-', `H-', `s-', `A-' and `S-' at the front of the symbol name. Thus, here is how to make Hyper-Meta-RIGHT move forward a word:

```
(global-set-key [H-M-right] 'forward-word)
```

Named ASCII Control Characters

TAB, RET, BS, LFD, ESC and DEL started out as names for certain ASCII control characters, used so often that they have special keys of their own. Later, users found it convenient to distinguish in Emacs between these keys and the "same" control characters typed with the CTRL key.

Emacs 19 distinguishes these two kinds of input, when used with the X Window System. It treats the "special" keys as function keys named `tab`, `return`, `backspace`, `linefeed`, `escape`, and `delete`. These function keys translate automatically into the corresponding ASCII characters *if* they have no bindings of their own. As a result, neither users nor Lisp programs need to pay attention to the distinction unless they care to.

If you do not want to distinguish between (for example) TAB and C-i, make just one binding, for the ASCII character TAB (octal code 011). If you do want to distinguish, make one binding for this ASCII character, and another for the "function key" `tab`.

With an ordinary ASCII terminal, there is no way to distinguish between TAB and C-i (and likewise for other such pairs), because the terminal sends the same character in both cases.

Rebinding Mouse Buttons

Emacs uses Lisp symbols to designate mouse buttons, too. The ordinary mouse events in Emacs are click events; these happen when you press a button and release it without moving the mouse. You can also get drag events, when you move the mouse while holding the button down. Drag events happen when you finally let go of the button.

The symbols for basic click events are `mouse-1` for the leftmost button, `mouse-2` for the next, and so on. Here is how you can redefine the second mouse button to split the current window:

```
(global-set-key [mouse-2] 'split-window-vertically)
```

The symbols for drag events are similar, but have the prefix `drag-' before the word `mouse'. For example, dragging the first button generates a `drag-mouse-1` event.

You can also define bindings for events that occur when a mouse button is pressed down. These events start with `down-' instead of `drag-'. Such events are generated only if they have key bindings. When you get a button-down event, a corresponding click or drag event will always follow.

If you wish, you can distinguish single, double, and triple clicks. A double click means clicking a mouse button twice in approximately the same place. The first click generates an ordinary click event. The second click, if it comes soon enough, generates a double-click event instead. The event type for a double click event starts with `double-': for example, `double-mouse-3`.

This means that you can give a special meaning to the second click at the same place, but it must act on the assumption that the ordinary single click definition has run when the first click was received.

This constrains what you can do with double clicks, but user interface designers say that this constraint ought to be followed in any case. A double click should do something similar to the single click, only

"more so". The command for the double-click event should perform the extra work for the double cli

If a double-click event has no binding, it changes to the corresponding single-click event. Thus, if you don't define a particular double click specially, it executes the single-click command twice.

Emacs also supports triple-click events whose names start with `triple-'. Emacs does not distinguish quadruple clicks as event types; clicks beyond the third generate additional triple-click events. However, the full number of clicks is recorded in the event list, so you can distinguish if you really want to. We don't recommend distinct meanings for more than three clicks, but sometimes it is useful for subsequent clicks to cycle through the same set of three meanings, so that four clicks are equivalent to one click, five are equivalent to two, and six are equivalent to three.

Emacs also records multiple presses in drag and button-down events. For example, when you press a button twice, then move the mouse while holding the button, Emacs gets a `double-drag-' event. And at the moment when you press it down for the second time, Emacs gets a `double-down-' event (which is ignored, like all button-down events, if it has no binding).

The variable `double-click-time` specifies how long may elapse between clicks that are recognized as a pair. Its value is measured in milliseconds. If the value is `nil`, double clicks are not detected at all. If the value is `t`, then there is no time limit.

The symbols for mouse events also indicate the status of the modifier keys, with the usual prefixes `C-', `M-', `H-', `s-', `A-' and `S-'. These always precede `double-' or `triple-', which always precede `drag-' or `down-'.

A frame includes areas that don't show text from the buffer, such as the mode line and the scroll bar. You can tell whether a mouse button comes from a special area of the screen by means of dummy "prefix keys." For example, if you click the mouse in the mode line, you get the prefix key `mode-line` before the ordinary mouse-button symbol. Thus, here is how to define the command for clicking the first button in a mode line to run `scroll-up`:

```
(global-set-key [mode-line mouse-1] 'scroll-up)
```

Here is the complete list of these dummy prefix keys and their meanings:

`mode-line`
: The mouse was in the mode line of a window.

`vertical-line`
: The mouse was in the vertical line separating side-by-side windows. (If you use scroll bars, they appear in place of these vertical lines.)

`vertical-scroll-bar`
: The mouse was in a vertical scroll bar. (This is the only kind of scroll bar Emacs currently supports.)

You can put more than one mouse button in a key sequence, but it isn't usual to do so.

Disabling Commands

Disabling a command marks the command as requiring confirmation before it can be executed. The purpose of disabling a command is to prevent beginning users from executing it by accident and being confused.

An attempt to invoke a disabled command interactively in Emacs displays a window containing the command's name, its documentation, and some instructions on what to do immediately; then Emacs asks for input saying whether to execute the command as requested, enable it and execute it, or cancel. If you decide to enable the command, you are asked whether to do this permanently or just for the current session. Enabling permanently works by automatically editing your `.emacs' file.

The direct mechanism for disabling a command is to put a non-`nil` `disabled` property on the Lisp symbol for the command. Here is the Lisp program to do this:

```
(put 'delete-region 'disabled t)
```

If the value of the `disabled` property is a string, that string is included in the message printed when the command is used:

```
(put 'delete-region 'disabled
     "It's better to use `kill-region' instead.\n")
```

You can make a command disabled either by editing the `.emacs' file directly or with the command M-x disable-command, which edits the `.emacs' file for you. Likewise, M-x enable-command edits `.emacs' to enable a command permanently. See section The Init File, `~/.emacs'.

Whether a command is disabled is independent of what key is used to invoke it; disabling also applies if the command is invoked using M-x. Disabling a command has no effect on calling it as a function from Lisp programs.

Keyboard Translations

Some keyboards do not make it convenient to send all the special characters that Emacs uses. The most common problem case is the DEL character. Some keyboards provide no convenient way to type this very important character--usually because they were designed to expect the character C-h to be used for deletion. On these keyboard, if you press the key normally used for deletion, Emacs handles the C-h as a prefix character and offers you a list of help options, which is not what you want.

You can work around this problem within Emacs by setting up keyboard translations to turn C-h into DEL and DEL into C-h, as follows:

```
;; Translate C-h to DEL.
(keyboard-translate ?\C-h ?\C-?)

;; Translate DEL to C-h.
```


```
(keyboard-translate ?\C-? ?\C-h)
```

Keyboard translations are not the same as key bindings in keymaps (see section Keymaps). Emacs contains numerous keymaps that apply in different situations, but there is only one set of keyboard translations, and it applies to every character that Emacs reads from the terminal. Keyboard translations take place at the lowest level of input processing; the keys that are looked up in keymaps contain the characters that result from keyboard translation.

Under X, the keyboard key named DELETE is a function key and is distinct from the ASCII character named DEL. See section Named ASCII Control Characters. Keyboard translations affect only ASCII character input, not function keys; thus, the above example used under X does not affect the DELETE key. However, the translation above isn't necessary under X, because Emacs can also distinguish between the BACKSPACE key and C-h; and it normally treats BACKSPACE as DEL.

For full information about how to use keyboard translations, see section 'Translating Input' in The Emacs Lisp Reference Manual.

The Syntax Table

All the Emacs commands which parse words or balance parentheses are controlled by the syntax table. The syntax table says which characters are opening delimiters, which are parts of words, which are string quotes, and so on. Each major mode has its own syntax table (though sometimes related major modes use the same one) which it installs in each buffer that uses that major mode. The syntax table installed in the current buffer is the one that all commands use, so we call it "the" syntax table. A syntax table is a Lisp object, a vector of length 256 whose elements are numbers.

To display a description of the contents of the current syntax table, type C-h s (`describe-syntax`). The description of each character includes both the string you would have to give to `modify-syntax-entry` to set up that character's current syntax, and some English to explain that string if necessary.

For full information on the syntax table, see section 'Syntax Tables' in The Emacs Lisp Reference Manual.

The Init File, `~/.emacs'

When Emacs is started, it normally loads a Lisp program from the file `.emacs' in your home directory. We call this file your init file because it specifies how to initialize Emacs for you. You can use the command line switches `-q' and `-u' to tell Emacs whether to load an init file, and which one (see section Entering and Exiting Emacs).

There can also be a default init file, which is the library named `default.el', found via the standard search path for libraries. The Emacs distribution contains no such library; your site may create one for local customizations. If this library exists, it is loaded whenever you start Emacs (except when you specify `-q'). But your init file, if any, is loaded first; if it sets `inhibit-default-init` non-`nil`, then `default' is not loaded.

Your site may also have a site startup file; this is named `site-start.el', if it exists. Emacs lo this library before it loads your init file. To inhibit loading of this library, use the option `-no-site-file'.

If you have a large amount of code in your `.emacs' file, you should move it into another file such as `~/something.el', byte-compile it, and make your `.emacs' file load it with `(load "~/something")'. See section 'Byte Compilation' in the Emacs Lisp Reference Manual, for more information about compiling Emacs Lisp programs.

Init File Syntax

The `.emacs' file contains one or more Lisp function call expressions. Each of these consists of a function name followed by arguments, all surrounded by parentheses. For example, `(setq fill-column 60)` calls the function `setq` to set the variable `fill-column` (see section Filling Text) to 60.

The second argument to `setq` is an expression for the new value of the variable. This can be a constant, a variable, or a function call expression. In `.emacs', constants are used most of the time. They can be:

Numbers:

Numbers are written in decimal, with an optional initial minus sign.

Strings:

Lisp string syntax is the same as C string syntax with a few extra features. Use a double-quote character to begin and end a string constant.

In a string, you can include newlines and special characters literally. But often it is cleaner to use backslash sequences for them: `\n' for newline, `\b' for backspace, `\r' for carriage return, `\t' for tab, `\f' for formfeed (control-L), `\e' for escape, `\\' for a backslash, `\"' for a double-quote, or `\ooo' for the character whose octal code is ooo. Backslash and double-quote are the only characters for which backslash sequences are mandatory.

`\C-' can be used as a prefix for a control character, as in `\C-s' for ASCII control-S, and `\M-' can be used as a prefix for a Meta character, as in `\M-a' for Meta-A or `\M-\C-a' for Control-Meta-A.

Characters:

Lisp character constant syntax consists of a `?' followed by either a character or an escape sequence starting with `\'. Examples: `?x`, `?\n`, `?\"`, `?\)`. Note that strings and characters are not interchangeable in Lisp; some contexts require one and some contexts require the other.

True:

`t` stands for `true'.

False:

`nil` stands for `false'.

Other Lisp objects:

Write a single-quote (') followed by the Lisp object you want.

Init File Examples

Here are some examples of doing certain commonly desired things with Lisp expressions:

- Make TAB in C mode just insert a tab if point is in the middle of a line.

  ```
  (setq c-tab-always-indent nil)
  ```

  Here we have a variable whose value is normally `t` for `true' and the alternative is `nil` for `false'.

- Make searches case sensitive by default (in all buffers that do not override this).

  ```
  (setq-default case-fold-search nil)
  ```

  This sets the default value, which is effective in all buffers that do not have local values for the variable. Setting `case-fold-search` with `setq` affects only the current buffer's local value, which is not what you probably want to do in an init file.

- Specify your own email address, if Emacs can't figure it out correctly.

  ```
  (setq user-mail-address "coon@yoyodyne.com")
  ```

  Various Emacs packages that need your own email address use the value of `user-mail-address`.

- Make Text mode the default mode for new buffers.

  ```
  (setq default-major-mode 'text-mode)
  ```

  Note that `text-mode` is used because it is the command for entering Text mode. The single-quote before it makes the symbol a constant; otherwise, `text-mode` would be treated as a variable name.

- Turn on Auto Fill mode automatically in Text mode and related modes.

  ```
  (add-hook 'text-mode-hook
    '(lambda () (auto-fill-mode 1)))
  ```

  This shows how to add a hook function to a normal hook variable (see section Hooks). The function we supply is a list starting with `lambda`, with a single-quote in front of it to make it a list constant rather than an expression.

  It's beyond the scope of this manual to explain Lisp functions, but for this example it is enough to know that the effect is to execute `(auto-fill-mode 1)` when Text mode is entered. You can replace that with any other expression that you like, or with several expressions in a row.

  Emacs comes with a function named `turn-on-auto-fill` whose definition is `(lambda () (auto-fill-mode 1))`. Thus, a simpler way to write the above example is as follows:

  ```
  (add-hook 'text-mode-hook 'turn-on-auto-fill)
  ```

- Load the installed Lisp library named `foo' (actually a file `foo.elc' or `foo.el' in a standard Emacs directory).

  ```
  (load "foo")
  ```

  When the argument to `load` is a relative file name, not starting with `/' or `~', `load` searches the directories in `load-path` (see section Libraries of Lisp Code for Emacs).

- Load the compiled Lisp file `foo.elc' from your home directory.

  ```
  (load "~/foo.elc")
  ```

  Here an absolute file name is used, so no searching is done.

- Rebind the key C-x l to run the function `make-symbolic-link`.

  ```
  (global-set-key "\C-xl" 'make-symbolic-link)
  ```

  or

  ```
  (define-key global-map "\C-xl" 'make-symbolic-link)
  ```

  Note once again the single-quote used to refer to the symbol `make-symbolic-link` instead of its value as a variable.

- Do the same thing for Lisp mode only.

  ```
  (define-key lisp-mode-map "\C-xl" 'make-symbolic-link)
  ```

- Redefine all keys which now run `next-line` in Fundamental mode so that they run `forward-line` instead.

  ```
  (substitute-key-definition 'next-line 'forward-line
                             global-map)
  ```

- Make C-x C-v undefined.

  ```
  (global-unset-key "\C-x\C-v")
  ```

  One reason to undefine a key is so that you can make it a prefix. Simply defining C-x C-v anything will make C-x C-v a prefix, but C-x C-v must first be freed of its usual non-prefix definition.

- Make `$' have the syntax of punctuation in Text mode. Note the use of a character constant for `$'.

  ```
  (modify-syntax-entry ?\$ "." text-mode-syntax-table)
  ```

- Enable the use of the command `eval-expression` without confirmation.

  ```
  (put 'eval-expression 'disabled nil)
  ```

Terminal-specific Initialization

Each terminal type can have a Lisp library to be loaded into Emacs when it is run on that type of terminal. For a terminal type named termtype, the library is called `term/termtype' and it is found by searching the directories `load-path` as usual and trying the suffixes `.elc' and `.el'. Normally it appears in the subdirectory `term' of the directory where most Emacs libraries are kept.

The usual purpose of the terminal-specific library is to map the escape sequences used by the terminal's function keys onto more meaningful names, using `function-key-map`. See the file `term/lk201.el' for an example of how this is done. Many function keys are mapped automatically according to the information in the Termcap data base; the terminal-specific library needs to map only the function keys that Termcap does not specify.

When the terminal type contains a hyphen, only the part of the name before the first hyphen is significant in choosing the library name. Thus, terminal types `aaa-48' and `aaa-30-rv' both use the library `term/aaa'. The code in the library can use `(getenv "TERM")` to find the full terminal type name.

The library's name is constructed by concatenating the value of the variable `term-file-prefix` and the terminal type. Your `.emacs' file can prevent the loading of the terminal-specific library by setting `term-file-prefix` to `nil`.

Emacs runs the hook `term-setup-hook` at the end of initialization, after both your `.emacs' file and any terminal-specific library have been read in. Add hook functions to this hook if you wish to override part of any of the terminal-specific libraries and to define initializations for terminals that do not have a library. See section Hooks.

How Emacs Finds Your Init File

Normally Emacs uses the environment variable HOME to find `.emacs'; that's what `~' means in a file name. But if you have done `su`, Emacs tries to find your own `.emacs', not that of the user you are currently pretending to be. The idea is that you should get your own editor customizations even if you are running as the super user.

More precisely, Emacs first determines which user's init file to use. It gets the user name from the environment variables LOGNAME and USER; if neither of those exists, it uses effective user-ID. If that user name matches the real user-ID, then Emacs uses HOME; otherwise, it looks up the home directory corresponding to that user name in the system's data base of users.

Go to the previous, next section.

Go to the previous, next section.

Dealing with Common Problems

If you type an Emacs command you did not intend, the results are often mysterious. This chapter tells what you can do to cancel your mistake or recover from a mysterious situation. Emacs bugs and system crashes are also considered.

Quitting and Aborting

C-g
: Quit. Cancel running or partially typed command.

C-]
: Abort innermost recursive editing level and cancel the command which invoked it (`abort-recursive-edit`).

ESC ESC ESC
: Either quit or abort, whichever makes sense (`keyboard-escape-quit`).

M-x top-level
: Abort all recursive editing levels that are currently executing.

C-x u
: Cancel a previously made change in the buffer contents (`undo`).

There are two ways of canceling commands which are not finished executing: quitting with C-g, and aborting with C-] or M-x top-level. Quitting cancels a partially typed command or one which is already running. Aborting exits a recursive editing level and cancels the command that invoked the recursive edit. (See section Recursive Editing Levels.)

Quitting with C-g is used for getting rid of a partially typed command, or a numeric argument that you don't want. It also stops a running command in the middle in a relatively safe way, so you can use it if you accidentally give a command which takes a long time. In particular, it is safe to quit out of killing; either your text will *all* still be in the buffer, or it will *all* be in the kill ring (or maybe both). Quitting an incremental search does special things documented under searching; in general, it may take two successive C-g characters to get out of a search (see section Incremental Search).

C-g works by setting the variable `quit-flag` to `t` the instant C-g is typed; Emacs Lisp checks this variable frequently and quits if it is non-`nil`. C-g is only actually executed as a command if you type it while Emacs is waiting for input.

If you quit with C-g a second time before the first C-g is recognized, you activate the "emergency escape" feature and return to the shell. See section Emergency Escape.

There may be times when you cannot quit. When Emacs is waiting for the operating system to do

something, quitting is impossible unless special pains are taken for the particular system call within Emacs where the waiting occurs. We have done this for the system calls that users are likely to want to quit from, but it's possible you will find another. In one very common case--waiting for file input or output using NFS--Emacs itself knows how to quit, but most NFS implementations simply do not allow user programs to stop waiting for NFS when the NFS server is hung.

Aborting with C-] (`abort-recursive-edit`) is used to get out of a recursive editing level and cancel the command which invoked it. Quitting with C-g does not do this, and could not do this, because it is used to cancel a partially typed command *within* the recursive editing level. Both operations are useful. For example, if you are in a recursive edit and type C-u 8 to enter a numeric argument, you can cancel that argument with C-g and remain in the recursive edit.

The command ESC ESC ESC (`keyboard-escape-quit`) can either quit or abort. This key was defined because ESC is used to "get out" in many PC programs. It can cancel a prefix argument, clear a selected region, or get out of a Query Replace, like C-g. It can get out of the minibuffer or a recursive edit, like C-]. It can also get out of splitting the frame into multiple windows, like C-x 1. One thing it cannot do, however, is stop a command that is running. That's because it executes as an ordinary command, and Emacs doesn't notice it until it is ready for a command.

The command M-x top-level is equivalent to "enough" C-] commands to get you out of all the levels of recursive edits that you are in. C-] gets you out one level at a time, but M-x top-level goes out all levels at once. Both C-] and M-x top-level are like all other commands, and unlike C-g, in that they take effect only when Emacs is ready for a command. C-] is an ordinary key and has its meaning only because of its binding in the keymap. See section Recursive Editing Levels.

C-x u (`undo`) is not strictly speaking a way of canceling a command, but you can think of it as canceling a command that already finished executing. See section Undoing Changes.

Dealing with Emacs Trouble

This section describes various conditions in which Emacs fails to work normally, and how to recognize them and correct them.

If DEL Fails to Delete

If you find that DEL enters Help like Control-h instead of deleting a character, your terminal is sending the wrong code for DEL. You can work around this problem by changing the keyboard translation table (see section Keyboard Translations).

Recursive Editing Levels

Recursive editing levels are important and useful features of Emacs, but they can seem like malfunctions to the user who does not understand them.

If the mode line has square brackets `[...]' around the parentheses that contain the names of the major and minor modes, you have entered a recursive editing level. If you did not do this on purpose, or if you don't

understand what that means, you should just get out of the recursive editing level. To do so, type M-C top-level. This is called getting back to top level. See section Recursive Editing Levels.

Garbage on the Screen

If the data on the screen looks wrong, the first thing to do is see whether the text is really wrong. Type C-l, to redisplay the entire screen. If the screen appears correct after this, the problem was entirely in the previous screen update. (Otherwise, see section Garbage in the Text.)

Display updating problems often result from an incorrect termcap entry for the terminal you are using. The file `etc/TERMS' in the Emacs distribution gives the fixes for known problems of this sort. `INSTALL' contains general advice for these problems in one of its sections. Very likely there is simply insufficient padding for certain display operations. To investigate the possibility that you have this sort of problem, try Emacs on another terminal made by a different manufacturer. If problems happen frequently on one kind of terminal but not another kind, it is likely to be a bad termcap entry, though it could also be due to a bug in Emacs that appears for terminals that have or that lack specific features.

Garbage in the Text

If C-l shows that the text is wrong, try undoing the changes to it using C-x u until it gets back to a state you consider correct. Also try C-h l to find out what command you typed to produce the observed results.

If a large portion of text appears to be missing at the beginning or end of the buffer, check for the word `Narrow' in the mode line. If it appears, the text you don't see is probably still present, but temporarily off-limits. To make it accessible again, type C-x n w. See section Narrowing.

Spontaneous Entry to Incremental Search

If Emacs spontaneously displays `I-search:' at the bottom of the screen, it means that the terminal is sending C-s and C-q according to the poorly designed xon/xoff "flow control" protocol.

If this happens to you, your best recourse is to put the terminal in a mode where it will not use flow control, or give it so much padding that it will never send a C-s. (One way to increase the amount of padding is to set the variable `baud-rate` to a larger value. Its value is the terminal output speed, measured in the conventional units of baud.)

If you don't succeed in turning off flow control, the next best thing is to tell Emacs to cope with it. To do this, call the function `enable-flow-control`.

Typically there are particular terminal types with which you must use flow control. You can conveniently ask for flow control on those terminal types only, using `enable-flow-control-on`. For example, if you find you must use flow control on VT-100 and H19 terminals, put the following in your `.emacs' file:

```
(enable-flow-control-on "vt100" "h19")
```

When flow control is enabled, you must type C-\ to get the effect of a C-s, and type C-^ to get the effect of a C-q. (These aliases work by means of keyboard translations; see section Keyboard Translations.)

Running out of Memory

If you get the error message `Virtual memory exceeded', save your modified buffers with C-x s. This method of saving them has the smallest need for additional memory. Emacs keeps a reserve of memory which it makes available when this error happens; that should be enough to enable C-x s to complete its work.

Once you have saved your modified buffers, you can exit this Emacs job and start another, or you can use M-x kill-some-buffers to free space in the current Emacs job. If you kill buffers containing a substantial amount of text, you can safely go on editing. Emacs refills its memory reserve automatically when it sees sufficient free space available, in case you run out of memory another time.

Do not use M-x buffer-menu to save or kill buffers when you run out of memory, because the buffer menu needs a fair amount memory itself, and the reserve supply may not be enough.

Recovery After a Crash

If Emacs or the computer crashes, you can recover the files you were editing at the time of the crash from their auto-save files. To do this, start Emacs again and type the command M-x recover-session.

This command initially displays a buffer which lists interrupted session files, each with its date. You must choose which session to recover from. Typically the one you want is the most recent one. Move point to the one you choose, and type C-c C-c.

Then `recover-session` asks about each of the files that you were editing during that session; it asks whether to recover that file. If you answer y for a file, it shows the dates of that file and its auto-save file, then asks once again whether to recover that file. For the second question, you must confirm with yes. If you do, Emacs visits the file but gets the text from the auto-save file.

When `recover-session` is done, the files you've chosen to recover are present in Emacs buffers. You should then save them. Only this--saving them--updates the files themselves.

Emergency Escape

Because at times there have been bugs causing Emacs to loop without checking `quit-flag`, a special feature causes Emacs to be suspended immediately if you type a second C-g while the flag is already set, so you can always get out of GNU Emacs. Normally Emacs recognizes and clears `quit-flag` (and quits!) quickly enough to prevent this from happening.

When you resume Emacs after a suspension caused by multiple C-g, it asks two questions before going back to what it had been doing:

```
Auto-save? (y or n)
Abort (and dump core)? (y or n)
```

Answer each one with y or n followed by RET.

Saying y to `Auto-save?' causes immediate auto-saving of all modified buffers in which auto-saving is enabled.

Saying y to `Abort (and dump core)?' causes an illegal instruction to be executed, dumping core. This is to enable a wizard to figure out why Emacs was failing to quit in the first place. Execution does not continue after a core dump. If you answer n, execution does continue. With luck, GNU Emacs will ultimately check `quit-flag` and quit normally. If not, and you type another C-g, it is suspended again.

If Emacs is not really hung, just slow, you may invoke the double C-g feature without really meaning to. Then just resume and answer n to both questions, and you will arrive at your former state. Presumably the quit you requested will happen soon.

The double-C-g feature is turned off when Emacs is running under the X Window System, since you can use the window manager to kill Emacs or to create another window and run another program.

Help for Total Frustration

If using Emacs (or something else) becomes terribly frustrating and none of the techniques described above solve the problem, Emacs can still help you.

First, if the Emacs you are using is not responding to commands, type C-g C-g to get out of it and then start a new one.

Second, type M-x doctor RET.

The doctor will help you feel better. Each time you say something to the doctor, you must end it by typing RET RET. This lets the doctor know you are finished.

Reporting Bugs

Sometimes you will encounter a bug in Emacs. Although we cannot promise we can or will fix the bug, and we might not even agree that it is a bug, we want to hear about problems you encounter. Often we agree they are bugs and want to fix them.

To make it possible for us to fix a bug, you must report it. In order to do so effectively, you must know when and how to do it.

When Is There a Bug

If Emacs executes an illegal instruction, or dies with an operating system error message that indicates a problem in the program (as opposed to something like "disk full"), then it is certainly a bug.

If Emacs updates the display in a way that does not correspond to what is in the buffer, then it is certainly a bug. If a command seems to do the wrong thing but the problem corrects itself if you type C-l, it is a case of incorrect display updating.

Taking forever to complete a command can be a bug, but you must make certain that it was really Emacs's fault. Some commands simply take a long time. Type C-g and then C-h l to see whether the input Emacs received was what you intended to type; if the input was such that you *know* it should have been processed quickly, report a bug. If you don't know whether the command should take a long time, find out by looking in the manual or by asking for assistance.

If a command you are familiar with causes an Emacs error message in a case where its usual definition ought to be reasonable, it is probably a bug.

If a command does the wrong thing, that is a bug. But be sure you know for certain what it ought to have done. If you aren't familiar with the command, or don't know for certain how the command is supposed to work, then it might actually be working right. Rather than jumping to conclusions, show the problem to someone who knows for certain.

Finally, a command's intended definition may not be best for editing with. This is a very important sort of problem, but it is also a matter of judgment. Also, it is easy to come to such a conclusion out of ignorance of some of the existing features. It is probably best not to complain about such a problem until you have checked the documentation in the usual ways, feel confident that you understand it, and know for certain that what you want is not available. If you are not sure what the command is supposed to do after a careful reading of the manual, check the index and glossary for any terms that may be unclear.

If after careful rereading of the manual you still do not understand what the command should do, that indicates a bug in the manual, which you should report. The manual's job is to make everything clear to people who are not Emacs experts--including you. It is just as important to report documentation bugs as program bugs.

If the on-line documentation string of a function or variable disagrees with the manual, one of them must be wrong; that is a bug.

Understanding Bug Reporting

When you decide that there is a bug, it is important to report it and to report it in a way which is useful. What is most useful is an exact description of what commands you type, starting with the shell command to run Emacs, until the problem happens.

The most important principle in reporting a bug is to report *facts*, not hypotheses or categorizations. It is always easier to report the facts, but people seem to prefer to strain to posit explanations and report them instead. If the explanations are based on guesses about how Emacs is implemented, they will be useless; we will have to try to figure out what the facts must have been to lead to such speculations. Sometimes this is impossible. But in any case, it is unnecessary work for us.

For example, suppose that you type C-x C-f /glorp/baz.ugh RET, visiting a file which (you know) happens to be rather large, and Emacs prints out `I feel pretty today'. The best way to report the bug is with a sentence like the preceding one, because it gives all the facts and nothing but the facts.

Do not assume that the problem is due to the size of the file and say, "When I visit a large file, Emacs prints out `I feel pretty today'." This is what we mean by "guessing explanations". The problem is just as likely to be due to the fact that there is a `z' in the file name. If this is so, then when we got your report,

we would try out the problem with some "large file", probably with no `z' in its name, and not find anything wrong. There is no way in the world that we could guess that we should try visiting a file with `z' in its name.

Alternatively, the problem might be due to the fact that the file starts with exactly 25 spaces. For this reason, you should make sure that you inform us of the exact contents of any file that is needed to reproduce the bug. What if the problem only occurs when you have typed the C-x C-a command previously? This is why we ask you to give the exact sequence of characters you typed since starting the Emacs session.

You should not even say "visit a file" instead of C-x C-f unless you *know* that it makes no difference which visiting command is used. Similarly, rather than saying "if I have three characters on the line," say "after I type RET A B C RET C-p," if that is the way you entered the text.

Checklist for Bug Reports

The best way to send a bug report is to mail it electronically to the Emacs maintainers at `bug-gnu-emacs@prep.ai.mit.edu'. (If you want to suggest a change as an improvement, use the same address.)

If you'd like to read the bug reports, you can find them on the newsgroup `gnu.emacs.bug'; keep in mind, however, that as a spectator you should not criticize anything about what you see there. The purpose of bug reports is to give information to the Emacs maintainers. Spectators are welcome only as long as they do not interfere with this. In particular, some bug reports contain large amounts of data; spectators should not complain about this.

Please do not post bug reports using netnews; mail is more reliable than netnews about reporting your correct address, which we may need in order to ask you for more information.

If you can't send electronic mail, then mail the bug report on paper or machine-readable media to this address:

```
GNU Emacs Bugs
Free Software Foundation
59 Temple Place, Suite 330
Boston, MA 02111-1307 USA
```

We do not promise to fix the bug; but if the bug is serious, or ugly, or easy to fix, chances are we will want to.

A convenient way to send a bug report for Emacs is to use the command M-x report-emacs-bug. This sets up a mail buffer (see section Sending Mail) and automatically inserts *some* of the essential information. However, it cannot supply all the necessary information; you should still read and follow the guidelines below, so you can enter the other crucial information by hand before you send the message.

To enable maintainers to investigate a bug, your report should include all these things:

- The version number of Emacs. Without this, we won't know whether there is any point in looking

for the bug in the current version of GNU Emacs.

You can get the version number by typing M-x emacs-version RET. If that command does not work, you probably have something other than GNU Emacs, so you will have to report the bug somewhere else.

- The type of machine you are using, and the operating system name and version number. M-x emacs-version RET provides this information too. Copy its output from the `*Messages*' buffer, so that you get it all and get it accurately.
- The operands given to the `configure` command when Emacs was installed.
- A complete list of any modifications you have made to the Emacs source. (We may not have time to investigate the bug unless it happens in an unmodified Emacs. But if you've made modifications and you don't tell us, you are sending us on a wild goose chase.)

  Be precise about these changes. A description in English is not enough--send a context diff for them.

  Adding files of your own, or porting to another machine, is a modification of the source.
- Details of any other deviations from the standard procedure for installing GNU Emacs.
- The complete text of any files needed to reproduce the bug.

  If you can tell us a way to cause the problem without visiting any files, please do so. This makes it much easier to debug. If you do need files, make sure you arrange for us to see their exact contents. For example, it can often matter whether there are spaces at the ends of lines, or a newline after the last line in the buffer (nothing ought to care whether the last line is terminated, but try telling the bugs that).
- The precise commands we need to type to reproduce the bug.

  The easy way to record the input to Emacs precisely is to write a dribble file. To start the file, execute the Lisp expression

  ```
  (open-dribble-file "~/dribble")
  ```

  using M-: or from the `*scratch*' buffer just after starting Emacs. From then on, Emacs copies all your input to the specified dribble file until the Emacs process is killed.
- For possible display bugs, the terminal type (the value of environment variable `TERM`), the complete termcap entry for the terminal from `/etc/termcap' (since that file is not identical on all machines), and the output that Emacs actually sent to the terminal.

  The way to collect the terminal output is to execute the Lisp expression

  ```
  (open-termscript "~/termscript")
  ```

  using M-: or from the `*scratch*' buffer just after starting Emacs. From then on, Emacs copies all terminal output to the specified termscript file as well, until the Emacs process is killed. If the problem happens when Emacs starts up, put this expression into your `.emacs' file so that the termscript file will be open when Emacs displays the screen for the first time.

Be warned: it is often difficult, and sometimes impossible, to fix a terminal-dependent bug wit access to a terminal of the type that stimulates the bug.

- A description of what behavior you observe that you believe is incorrect. For example, "The Emacs process gets a fatal signal," or, "The resulting text is as follows, which I think is wrong."

  Of course, if the bug is that Emacs gets a fatal signal, then one can't miss it. But if the bug is incorrect text, the maintainer might fail to notice what is wrong. Why leave it to chance?

  Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of the source is out of sync, or you have encountered a bug in the C library on your system. (This has happened!) Your copy might crash and the copy here might not. If you *said* to expect a crash, then when Emacs here fails to crash, we would know that the bug was not happening. If you don't say to expect a crash, then we would not know whether the bug was happening--we would not be able to draw any conclusion from our observations.
- If the manifestation of the bug is an Emacs error message, it is important to report the precise text of the error message, and a backtrace showing how the Lisp program in Emacs arrived at the error.

  To get the error message text accurately, copy it from the `*Messages*' buffer into the bug report. Copy all of it, not just part.

  To make a backtrace for the error, evaluate the Lisp expression `(setq debug-on-error t)` before the error happens (that is to say, you must execute that expression and then make the bug happen). This causes the error to run the Lisp debugger, which shows you a backtrace. Copy the text of the debugger's backtrace into the bug report.

  This use of the debugger is possible only if you know how to make the bug happen again. If you can't make it happen again, at least copy the whole error message.
- Check whether any programs you have loaded into the Lisp world, including your `.emacs' file, set any variables that may affect the functioning of Emacs. Also, see whether the problem happens in a freshly started Emacs without loading your `.emacs' file (start Emacs with the `-q` switch to prevent loading the init file.) If the problem does *not* occur then, you must report the precise contents of any programs that you must load into the Lisp world in order to cause the problem to occur.
- If the problem does depend on an init file or other Lisp programs that are not part of the standard Emacs system, then you should make sure it is not a bug in those programs by complaining to their maintainers first. After they verify that they are using Emacs in a way that is supposed to work, they should report the bug.
- If you wish to mention something in the GNU Emacs source, show the line of code with a few lines of context. Don't just give a line number.

  The line numbers in the development sources don't match those in your sources. It would take extra work for the maintainers to determine what code is in your version at a given line number, and we could not be certain.
- Additional information from a C debugger such as GDB might enable someone to find a problem on a machine which he does not have available. If you don't know how to use GDB, please read

the GDB manual--it is not very long, and using GDB is easy. You can find the GDB distribution, including the GDB manual in online form, in most of the same places you can find the Emacs distribution.

However, you need to think when you collect the additional information if you want it to show what causes the bug.

For example, many people send just a backtrace, but that is not very useful by itself. A simple backtrace with arguments often conveys little about what is happening inside GNU Emacs, because most of the arguments listed in the backtrace are pointers to Lisp objects. The numeric values of these pointers have no significance whatever; all that matters is the contents of the objects they point to (and most of the contents are themselves pointers).

To provide useful information, you need to show the values of Lisp objects in Lisp notation. Do this for each variable which is a Lisp object, in several stack frames near the bottom of the stack. Look at the source to see which variables are Lisp objects, because the debugger thinks of them as integers.

To show a variable's value in Lisp syntax, first print its value, then use the user-defined GDB command `pr` to print the Lisp object in Lisp syntax. (If you must use another debugger, call the function `debug_print` with the object as an argument.) The `pr` command is defined by the file `` `.gdbinit' `` in the `` `src' `` subdirectory, and it works only if you are debugging a running process (not with a core dump).

To make Lisp errors stop Emacs and return to GDB, put a breakpoint at `Fsignal`.

To find out which Lisp functions are running, using GDB, move up the stack, and each time you get to a frame for the function `Ffuncall`, type these GDB commands:

```
p *args
pr
```

To print the first argument that the function received, use these commands:

```
p args[1]
pr
```

You can print the other arguments likewise. The argument `nargs` of `Ffuncall` says how many arguments `Ffuncall` received; these include the Lisp function itself and the arguments for that function.

- If the symptom of the bug is that Emacs fails to respond, don't assume Emacs is "hung"---it may instead be in an infinite loop. To find out which, make the problem happen under GDB and stop Emacs once it is not responding. (If Emacs is using X Windows directly, you can stop Emacs by typing C-c at the GDB job.) Then try stepping with `step'. If Emacs is hung, the `step' command won't return. If it is looping, `step' will return.

  If this shows Emacs is hung in a system call, stop it again and examine the arguments of the call. In your bug report, state exactly where in the source the system call is, and what the arguments are.

If Emacs is in an infinite loop, please determine where the loop starts and ends. The easiest way to do this is to use the GDB command `finish'. Each time you use it, Emacs resumes execution until it exits one stack frame. Keep typing `finish' until it doesn't return--that means the infinite loop is in the stack frame which you just tried to finish.

Stop Emacs again, and use `finish' repeatedly again until you get *back to* that frame. Then use `next' to step through that frame. By stepping, you will see where the loop starts and ends. Also please examine the data being used in the loop and try to determine why the loop does not exit when it should. Include all of this information in your bug report.

Here are some things that are not necessary in a bug report:

- A description of the envelope of the bug--this is not necessary for a reproducible bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

  This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. You might as well save time by not searching for additional examples.

  Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience. Errors in the output will be easier to spot, running under the debugger will take less time, etc.

  However, simplification is not vital; if you can't do this or don't have time to try, please report the bug with your original test case.

- A system call trace of Emacs execution.

  System call traces are very useful for certain special kinds of debugging, but in most cases they give little useful information. It is therefore strange that many people seem to think that *the* way to report information about a crash is to send a system call trace.

  In most programs, a backtrace is normally far, far more informative than a system call trace. Even in Emacs, a simple backtrace is generally more informative, though to give full information you should supplement the backtrace by displaying variable values and printing them as Lisp objects with `pr` (see above).

- A patch for the bug.

  A patch for the bug is useful if it is a good one. But don't omit the other information that a bug report needs, such as the test case, on the assumption that a patch is sufficient. We might see problems with your patch and decide to fix the problem another way, or we might not understand it at all. And if we can't understand what bug you are trying to fix, or why your patch should be an improvement, we mustn't install it.

- A guess about what the bug is or what it depends on.

  Such guesses are usually wrong. Even experts can't guess right about such things without first using the debugger to find the facts.

Sending Patches for GNU Emacs

If you would like to write bug fixes or improvements for GNU Emacs, that is very helpful. When you send your changes, please follow these guidelines to make it easy for the maintainers to use them. If you don't follow these guidelines, your information might still be useful, but using it will take extra work. Maintaining GNU Emacs is a lot of work in the best of circumstances, and we can't keep up unless you do your best to help.

- Send an explanation with your changes of what problem they fix or what improvement they bring about. For a bug fix, just include a copy of the bug report, and explain why the change fixes the bug.

  (Referring to a bug report is not as good as including it, because then we will have to look it up, and we have probably already deleted it if we've already fixed the bug.)
- Always include a proper bug report for the problem you think you have fixed. We need to convince ourselves that the change is right before installing it. Even if it is correct, we might have trouble understanding it if we don't have a way to reproduce the problem.
- Include all the comments that are appropriate to help people reading the source in the future understand why this change was needed.
- Don't mix together changes made for different reasons. Send them *individually*.

  If you make two changes for separate reasons, then we might not want to install them both. We might want to install just one. If you send them all jumbled together in a single set of diffs, we have to do extra work to disentangle them--to figure out which parts of the change serve which purpose. If we don't have time for this, we might have to ignore your changes entirely.

  If you send each change as soon as you have written it, with its own explanation, then two changes never get tangled up, and we can consider each one properly without any extra work to disentangle them.
- Send each change as soon as that change is finished. Sometimes people think they are helping us by accumulating many changes to send them all together. As explained above, this is absolutely the worst thing you could do.

  Since you should send each change separately, you might as well send it right away. That gives us the option of installing it immediately if it is important.
- Use `diff -c' to make your diffs. Diffs without context are hard to install reliably. More than that, they are hard to study; we must always study a patch to decide whether we want to install it. Unidiff format is better than contextless diffs, but not as easy to read as `-c' format.

  If you have GNU diff, use `diff -c -F'^[_a-zA-Z0-9$]+ *(" when making diffs of C code. This shows the name of the function that each change occurs in.
- Write the change log entries for your changes. This is both to save us the extra work of writing them, and to help explain your changes so we can understand them.

  The purpose of the change log is to show people where to find what was changed. So you need to be specific about what functions you changed; in large functions, it's often helpful to indicate where within the function the change was.

On the other hand, once you have shown people where to find the change, you need not explain its purpose in the change log. Thus, if you add a new function, all you need to say about it is that it is new. If you feel that the purpose needs explaining, it probably does--but put the explanation in comments in the code. It will be more useful there.

Please read the `ChangeLog' files in the `src' and `lisp' directories to see what sorts of information to put in, and to learn the style that we use. If you would like your name to appear in the header line, showing who made the change, send us the header line. See section Change Logs.

- When you write the fix, keep in mind that we can't install a change that would break other systems. Please think about what effect your change will have if compiled on another type of system.

  Sometimes people send fixes that *might* be an improvement in general--but it is hard to be sure of this. It's hard to install such changes because we have to study them very carefully. Of course, a good explanation of the reasoning by which you concluded the change was correct can help convince us.

  The safest changes are changes to the configuration files for a particular machine. These are safe because they can't create new bugs on other machines.

  Please help us keep up with the workload by designing the patch in a form that is clearly safe to install.

Contributing to Emacs Development

If you would like to help pretest Emacs releases to assure they work well, or if you would like to work on improving Emacs, please contact the maintainers at `bug-gnu-emacs@prep.ai.mit.edu`. A pretester should be prepared to investigate bugs as well as report them. If you'd like to work on improving Emacs, please ask for suggested projects or suggest your own ideas.

If you have already written an improvement, please tell us about it. If you have not yet started work, it is useful to contact `bug-gnu-emacs@prep.ai.mit.edu` before you start; it might be possible to suggest ways to make your extension fit in better with the rest of Emacs.

How To Get Help with GNU Emacs

If you need help installing, using or changing GNU Emacs, there are two ways to find it:

- Send a message to the mailing list `help-gnu-emacs@prep.ai.mit.edu`, or post your request on newsgroup `gnu.emacs.help`. (This mailing list and newsgroup interconnect, so it does not matter which one you use.)
- Look in the service directory for someone who might help you for a fee. The service directory is found in the file named `etc/SERVICE' in the Emacs distribution.

Go to the previous, next section.

Go to the previous, next section.

Command Line Arguments

GNU Emacs supports command line arguments to request various actions when invoking Emacs. These are for compatibility with other editors and for sophisticated activities. We don't recommend using them for ordinary editing.

Arguments starting with `-' are options. Other arguments specify files to visit. Emacs visits the specified files while it starts up. The last file name on your command line becomes the current buffer; the other files are also present in other buffers.

You can use options to specify various other things, such as the size and position of the X window Emacs uses, its colors, and so on. A few options support advanced usage, such as running Lisp functions on files in batch mode. The sections of this chapter describe the available options, arranged according to their purpose.

There are two ways of writing options: the short forms that start with a single `-', and the long forms that start with `--'. For example, `-d' is a short form and `--display' is the corresponding long form.

The long forms with `--' are easier to remember, but longer to type. However, you don't have to spell out the whole option name; any unambiguous abbreviation is enough. When a long option takes an argument, you can use either a space or an equal sign to separate the option name and the argument. Thus, you can write either `--display sugar-bombs:0.0' or `--display=sugar-bombs:0.0'. We recommend an equal sign because it makes the relationship clearer, and the tables below always show an equal sign.

Most options specify how to initialize Emacs, or set parameters for the Emacs session. We call them initial options. A few options specify things to do: for example, load libraries, call functions, or exit Emacs. These are called action options. These and file names together are called action arguments. Emacs processes all the action arguments in the order they are written.

Action Arguments

Here is a table of the action arguments and options:

`file'

Visit file using `find-file`. See section Visiting Files.

`+linenum file'

Visit file using `find-file`, then go to line number linenum in it.

`-l file'

`--load=file'

Load a file file of Lisp code with the function `load`. See section Libraries of Lisp Code for Emacs.

`-f function'

`--funcall=function'

Call Lisp function function with no arguments.

`--eval expression'

Evaluate Lisp expression expression.

`--insert=file'

Insert the contents of file into the current buffer. This is like what M-x insert-file does. See section Miscellaneous File Operations.

`--kill'

Exit from Emacs without asking for confirmation.

The init file can access the values of the action arguments as the elements of a list in the variable `command-line-args`. The init file can override the normal processing of the action arguments, or define new ones, by reading and setting this variable.

Initial Options

The initial options specify parameter for the Emacs session. This section describes the more general initial options; some other options specifically related to X Windows appear in the following sections.

Some initial options affect the loading of init files. The normal actions of Emacs are to first load `` `site-start.el' `` if it exists, then your own init file `` `~/.emacs' `` if it exists, and finally `` `default.el' `` if it exists; certain options prevent loading of some of these files or substitute other files for them.

`-t device'

`--terminal=device'

Use device as the device for terminal input and output.

`-d display'

`--display=display'

Use the X Window System and use the display named display to open the initial Emacs frame.

`-nw'

`--no-windows'

Don't communicate directly with X, disregarding the `DISPLAY` environment variable even if it is set.

`-batch'

`--batch'

Run Emacs in batch mode, which means that the text being edited is not displayed and the standard terminal interrupt characters such as C-z and C-c continue to have their normal effect. Emacs in batch mode outputs to `stderr` only what would normally be printed in the echo area under program control.

Batch mode is used for running programs written in Emacs Lisp from shell scripts, makefiles, and so on. Normally the `-l' option or `-f' option will be used as well, to invoke a Lisp program to do the batch processing.

`-batch' implies `-q' (do not load an init file). It also causes Emacs to kill itself after all command options have been processed. In addition, auto-saving is not done except in buffers for which it has been explicitly requested.

`-q'

`--no-init-file'

Do not load your Emacs init file `~/.emacs', or `default.el' either.

`--no-site-file'

Do not load `site-start.el'. The options `-q', `-u' and `-batch' have no effect on the loading of this file--this is the only option that blocks it.

`-u user'

`--user=user'

Load user's Emacs init file `~user/.emacs' instead of your own.

`--debug-init'

Enable the Emacs Lisp debugger for errors in the init file.

Command Argument Example

Here is an example of using Emacs with arguments and options. It assumes you have a Lisp program file called `hack-c.el' which, when loaded, performs some useful operation on current buffer, expected to be a C program.

```
emacs -batch foo.c -l hack-c -f save-buffer >& log
```

This says to visit `foo.c', load `hack-c.el' (which makes changes in the visited file), save `foo.c' (note that `save-buffer' is the function that C-x C-s is bound to), and then exit back to the shell (because of `-batch'). `-batch' also guarantees there will be no problem redirecting output to `log', because Emacs will not assume that it has a display terminal to work with.

Resuming Emacs with Arguments

You can specify action arguments for Emacs when you resume it after a suspension. To prepare for this, put the following code in your `.emacs' file (see section Hooks):

```
(add-hook 'suspend-hook 'resume-suspend-hook)
(add-hook 'suspend-resume-hook 'resume-process-args)
```

As further preparation, you must execute the shell script `emacs.csh' (if you use csh as your shell) or `emacs.bash' (if you use bash as your shell). These scripts define an alias named `edit', which will resume Emacs giving it new command line arguments such as files to visit.

Only action arguments work properly when you resume Emacs. Initial arguments are not recognized--it's too late to execute them anyway.

Note that resuming Emacs (with or without arguments) must be done from within the shell that is the parent of the Emacs job. This is why `edit' is an alias rather than a program or a shell script. It is not possible to implement a resumption command that could be run from other subjobs of the shell; no way to define a command that could be made the value of `EDITOR', for example. Therefore, this feature does not take the

place of the Emacs Server feature (see section Using Emacs as a Server).

The aliases use the Emacs Server feature if you appear to have a server Emacs running. However, they cannot determine this with complete accuracy. They may think that a server is still running when in actuality you have killed that Emacs, because the file `/tmp/.esrv...' still exists. If this happens, find that file and delete it.

Environment Variables

This appendix describes how Emacs uses environment variables. An environment variable is a string passed from the operating system to Emacs, and the collection of environment variables is known as the environment. Environment variable names are case sensitive and it is conventional to use upper case letters only.

Because environment variables come from the operating system there is no general way to set them; it depends on the operating system and especially the shell that you are using. For example, here's how to set the environment variable ORGANIZATION to `not very much' using bash:

```
export ORGANIZATION="not very much"
```

and here's how to do it in csh or tcsh:

```
setenv ORGANIZATION "not very much"
```

When Emacs is set-up to use the X windowing system, it inherits the use of a large number of environment variables from the X library. See the X documentation for more information.

General Variables

AUTHORCOPY
: The name of a file used to archive news articles posted with the GNUS package.

CDPATH
: Used by the cd command.

DOMAINNAME
: The name of the internet domain that the machine running Emacs is located in. Used by the GNUS package.

EMACSDATA
: Used to initialize the variable data-directory used to locate the architecture-independent files that come with Emacs. Setting this variable overrides the setting in `paths.h' when Emacs was built.

EMACSLOADPATH
: A colon-separated list of directories from which to load Emacs Lisp files. Setting this variable overrides the setting in `paths.h' when Emacs was built.

EMACSLOCKDIR

TOC

The directory that Emacs places lock files--files used to protect users from editing the same files simultaneously. Setting this variable overrides the setting in `paths.h' when Emacs was built.

EMACSPATH

The location of Emacs-specific binaries. Setting this variable overrides the setting in `paths.h' when Emacs was built.

ESHELL

Used for shell-mode to override the SHELL environment variable.

HISTFILE

The name of the file that shell commands are saved in between logins. This variable defaults to `~/.history' if you use (t)csh as shell, to `~/.bash_history' if you use bash, to `~/.sh_history' if you use ksh, and to `~/.history' otherwise.

HOME

The location of the user's files in the directory tree; used for expansion of file names starting with a tilde (`~'). On MS-DOS, it defaults to the directory from which Emacs was started, with `/bin' removed from the end if it was present.

HOSTNAME

The name of the machine that Emacs is running on.

INCPATH

A colon-separated list of directories. Used by the `complete' package to search for files.

INFOPATH

A colon separated list of directories holding info files. Setting this variable overrides the setting in `paths.el' when Emacs was built.

LOGNAME

The user's login name. See also USER.

MAIL

The name of the user's system mail box.

MAILRC

Name of file containing mail aliases. This defaults to `~/.mailrc'.

MH

Name of setup file for the mh system. This defaults to `~/.mh_profile'.

NAME

The real-world name of the user.

NNTPSERVER

The name of the news server. Used by the mh and GNUS packages.

ORGANIZATION

The name of the organization to which you belong. Used for setting the `Organization:' header in your posts from the GNUS package.

PATH

A colon-separated list of directories in which executables reside. (On MS-DOS, it is semicolon-separated instead.) This variable is used to set the Emacs Lisp variable `exec-path' which

you should consider to use instead.

`PWD`

If set, this should be the default directory when Emacs was started.

`REPLYTO`

If set, this specifies an initial value for the variable `mail-default-reply-to`. See section Mail Header Fields.

`SAVEDIR`

The name of a directory in which news articles are saved by default. Used by the GNUS package.

`SHELL`

The name of an interpreter used to parse and execute programs run from inside Emacs.

`TERM`

The name of the terminal that Emacs is running on. The variable must be set unless Emacs is run in batch mode. On MS-DOS, it defaults to `internal', which specifies a built-in terminal emulation that handles the machine's own display.

`TERMCAP`

The name of the termcap library file describing how to program the terminal specified by the `TERM` variable. This defaults to `` `/etc/termcap' ``.

`TMPDIR`

Used by the Emerge package as a prefix for temporary files.

`TZ`

This specifies the current time zone and possibly also daylight savings information. On MS-DOS, the default is based on country code; see the file `` `msdos.c' `` for details.

`USER`

The user's login name. See also `LOGNAME`. On MS-DOS, this defaults to `root'.

`VERSION_CONTROL`

Used to initialize the `version-control` variable (see section Single or Numbered Backups).

Misc Variables

These variables are used only on particular configurations:

`COMSPEC`

On MS-DOS, the name of the command interpreter to use. This is used to make a default value for the `SHELL` environment variable.

`NAME`

On MS-DOS, this variable defaults to the value of the `USER` variable.

`TEMP`

`TMP`

On MS-DOS, these specify the name of the directory for storing temporary files in.

`EMACSTEST`

On MS-DOS, this specifies a file to use to log the operation of the internal terminal emulator. This feature is useful for submitting bug reports.

`EMACSCOLORS`

Used on MS-DOS systems to set screen colors early, so that the screen won't momentarily flash the default colors when Emacs starts up. The value of this variable should be two-character encoding of the foreground (the first character) and the background (the second character) colors of the default face. Each character should be the hexadecimal code for the desired color on a standard PC text-mode display.

Only the low three bits of the background color are actually used, because the PC display supports only eight background colors.

`WINDOW_GFX`

Used when initializing the Sun windows system.

Specifying the Display Name

The environment variable `DISPLAY` tells all X clients, including Emacs, where to display their windows. Its value is set up by default in ordinary circumstances, when you start an X server and run jobs locally. Occasionally you may need to specify the display yourself; for example, if you do a remote login and want to run a client program remotely, displaying on your local screen.

With Emacs, the main reason people change the default display is to let them log into another system, run Emacs on that system, but have the window displayed at their local terminal. You might need to use login to another system because the files you want to edit are there, or because the Emacs executable file you want to run is there.

The syntax of the `DISPLAY` environment variable is `host:display.screen', where host is the host name of the X Window System server machine, display is an arbitrarily-assigned number that distinguishes your server (X terminal) from other servers on the same machine, and screen is a rarely-used field that allows an X server to control multiple terminal screens. The period and the screen field are optional. If included, screen is usually zero.

For example, if your host is named `glasperle' and your server is the first (or perhaps the only) server listed in the configuration, your `DISPLAY` is `glasperle:0.0'.

You can specify the display name explicitly when you run Emacs, either by changing the `DISPLAY` variable, or with the option `-d display' or `--display=display'. Here is an example:

```
emacs --display=glasperle:0 &
```

You can inhibit the direct use of X with the `-nw' option. This is also an initial option. It tells Emacs to display using ordinary ASCII on its controlling terminal.

Sometimes, security arrangements prevent a program on a remote system from displaying on your local system. In this case, trying to run Emacs produces messages like this:

```
Xlib:  connection to "glasperle:0.0" refused by server
```

You might be able to overcome this problem by using the `xhost` command on the local system to give permission for access from your remote machine.

Font Specification Options

By default, Emacs displays text in the font named `9x15', which makes each character nine pixels wide and fifteen pixels high. You can specify a different font on your command line through the option `-fn name'.

`-fn name'

Use font name as the default font.

`--font=name'

`--font' is an alias for `-fn'.

Under X, each font has a long name which consists of eleven words or numbers, separated by dashes. Some fonts also have shorter nicknames---`9x15' is such a nickname. You can use either kind of name. You can use wild card patterns for the font name; then Emacs lets X choose one of the fonts that match the pattern. Here is an example, which happens to specify the font whose nickname is `6x13':

```
emacs -fn "-misc-fixed-medium-r-semicondensed--13-*-*-*-c-60-iso8859-1" &
```

You can also specify the font in your `.Xdefaults' file:

```
emacs.font: -misc-fixed-medium-r-semicondensed--13-*-*-*-c-60-iso8859-1
```

A long font name has the following form:

```
-maker-family-weight-slant-widthtype-style...
...-pixels-height-horiz-vert-spacing-width-charset
```

family

This is the name of the font family--for example, `courier'.

weight

This is normally `bold', `medium' or `light'. Other words may appear here in some font names.

slant

This is `r' (roman), `i' (italic), `o' (oblique), `ri' (reverse italic), or `ot' (other).

widthtype

This is normally `condensed', `extended', `semicondensed' or `normal'. Other words may appear here in some font names.

style

This is an optional additional style name. Usually it is empty--most long font names have two hyphens in a row at this point.

pixels

This is the font height, in pixels.

height

This is the font height on the screen, measured in printer's points (approximately 1/72 of an inch), times ten. For a given vertical resolution, height and pixels are proportional; therefore, it is comm specify just one of them and use `*' for the other.

horiz

This is the horizontal resolution, in pixels per inch, of the screen for which the font is intended.

vert

This is the vertical resolution, in dots per inch, of the screen for which the font is intended. Normally the resolution of the fonts on your system is the right value for your screen; therefore, you normally specify `*' for this and horiz.

spacing

This is `m' (monospace), `p' (proportional) or `c' (character cell). Emacs can use `m' and `c' fonts.

width

This is the average character width, in pixels, times ten.

charset

This is the character set that the font depicts. Normally you should use `iso8859-1'.

Use only fixed width fonts--that is, fonts in which all characters have the same width; Emacs cannot yet handle display properly for variable width fonts. Any font with `m' or `c' in the spacing field of the long name is a fixed width font. Here's how to use the `xlsfonts` program to list all the fixed width fonts available on your system:

```
xlsfonts -fn '*x*'
xlsfonts -fn '*-*-*-*-*-*-*-*-*-*-m*'
xlsfonts -fn '*-*-*-*-*-*-*-*-*-*-c*'
```

To see what a particular font looks like, use the `xfd` command. For example:

```
xfd -fn 6x13
```

displays the entire font `6x13'.

While running Emacs, you can set the font of the current frame (see section Setting Frame Parameters) or for a specific kind of text (see section Using Multiple Typefaces).

Window Color Options

On a color display, you can specify which color to use for various parts of the Emacs display. To find out what colors are available on your system, look at the `/usr/lib/X11/rgb.txt' file. If you do not specify colors, the default for the background is white and the default for all other colors is black. On a monochrome display, the foreground is black, the background is white, and the border is gray if the display supports that.

Here is a list of the options for specifying colors:

`-fg color'

`--foreground-color=color'

Specify the foreground color.

`-bg color'

`--background-color=color'

Specify the background color.

`-bd color'

`--border-color=color'

Specify the color of the border of the X window.

`-cr color'

`--cursor-color=color'

Specify the color of the Emacs cursor which indicates where point is.

`-ms color'

`--mouse-color=color'

Specify the color for the mouse cursor when the mouse is in the Emacs window.

`-r'

`--reverse-video'

Reverse video--swap the foreground and background colors.

For example, to use a coral mouse cursor and a slate blue text cursor, enter:

```
emacs -ms coral -cr 'slate blue' &
```

You can reverse the foreground and background colors through the `-r' option or with the X resource `reverseVideo'.

Options for Window Geometry

The `-geometry' option controls the size and position of the initial Emacs frame. Here is the format for specifying the window geometry:

`-g widthxheight{+-}xoffset{+-}yoffset'

Specify window size width and height (measured in character columns and lines), and positions xoffset and yoffset (measured in pixels).

`--geometry=widthxheight{+-}xoffset{+-}yoffset'

This is another way of writing the same thing.

`{+-}` means either a plus sign or a minus sign. A plus sign before xoffset means it is the distance from the left side of the screen; a minus sign means it counts from the right side. A plus sign before yoffset means it is the distance from the top of the screen, and a minus sign there indicates the distance from the bottom. The values xoffset and yoffset may themselves be positive or negative, but that doesn't change their meaning, only their direction.

Emacs uses the same units as `xterm` does to interpret the geometry. The width and height are measured in characters, so a large font creates a larger frame than a small font. The xoffset and yoffset are measured in

pixels.

Since the mode line and the echo area occupy the last 2 lines of the frame, the height of the initial text window is 2 less than the height specified in your geometry. In non-X-toolkit versions of Emacs, the menu bar also takes one line of the specified number.

You do not have to specify all of the fields in the geometry specification.

If you omit both xoffset and yoffset, the window manager decides where to put the Emacs frame, possibly by letting you place it with the mouse. For example, `164x55' specifies a window 164 columns wide, enough for two ordinary width windows side by side, and 55 lines tall.

The default width for Emacs is 80 characters and the default height is 40 lines. You can omit either the width or the height or both. If you start the geometry with an integer, Emacs interprets it as the width. If you start with an `x' followed by an integer, Emacs interprets it as the height. Thus, `81' specifies just the width; `x45' specifies just the height.

If you start with `+' or `-', that introduces an offset, which means both sizes are omitted. Thus, `-3' specifies the xoffset only. (If you give just one offset, it is always xoffset.) `+3-3' specifies both the xoffset and the yoffset, placing the frame near the bottom left of the screen.

You can specify a default for any or all of the fields in `.Xdefaults' file, and then override selected fields with a `--geometry' option.

Internal and External Borders

An Emacs frame has an internal border and an external border. The internal border is an extra strip of the background color around all four edges of the frame. Emacs itself adds the internal border. The external border is added by the window manager outside the internal border; it may contain various boxes you can click on to move or iconify the window.

`-ib width'

`--internal-border=width'

Specify width as the width of the internal border.

`-bw width'

`--border-width=width'

Specify width as the width of the main border.

When you specify the size of the frame, that does not count the borders. The frame's position is measured from the outside edge of the external border.

Use the `-ib n' option to specify an internal border n pixels wide. The default is 1. Use `-bw n' to specify the width of the external border (though the window manager may not pay attention to what you specify). The default width of the external border is 2.

TOC

Frame Titles

An Emacs frame may or may not have a specified title. The frame title, if specified, appears in window decorations and icons as the name of the frame. If an Emacs frame has no specified title, the default title is the name of the executable program (if there is only one frame) or the selected window's buffer name (if there is more than one frame).

You can specify a title for the initial Emacs frame with a command line option:

`-title title'

`--title=title'

`-T title'

Specify title as the title for the initial Emacs frame.

The `--name' option (see section X Resources) also specifies the title for the initial Emacs frame.

Icons

Most window managers allow the user to "iconify" a frame, removing it from sight, and leaving a small, distinctive "icon" window in its place. Clicking on the icon window makes the frame itself appear again. If you have many clients running at once, you can avoid cluttering up the screen by iconifying most of the clients.

`-i'

`--icon-type'

Use a picture of a gnu as the Emacs icon.

`-iconic'

`--iconic'

Start Emacs in iconified state.

The `-i' or `--icon-type' option tells Emacs to use an icon window containing a picture of the GNU gnu. If omitted, Emacs lets the window manager choose what sort of icon to use--usually just a small rectangle containing the frame's title.

The `-iconic' option tells Emacs to begin running as an icon, rather than opening a frame right away. In this situation, the icon window provides only indication that Emacs has started; the usual text frame doesn't appear until you deiconify it.

X Resources

Programs running under the X Window System organize their user options under a hierarchy of classes and resources. You can specify default values for these options in your X resources file, usually named `~/.Xdefaults'.

Each line in the file specifies a value for one option or for a collection of related options, for one program or for several programs (optionally even for all programs).

Programs define named resources with particular meanings. They also define how to group resources into named classes. For instance, in Emacs, the `internalBorder' resource controls the width of the internal border, and the `borderWidth' resource controls the width of the external border. Both of these resources are part of the `BorderWidth' class. Case distinctions are significant in these names.

In `~/.Xdefaults', you can specify a value for a single resource on one line, like this:

```
emacs.borderWidth: 2
```

Or you can use a class name to specify the same value for all resources in that class. Here's an example:

```
emacs.BorderWidth: 2
```

If you specify a value for a class, it becomes the default for all resources in that class. You can specify values for individual resources as well; these override the class value, for those particular resources. Thus, this example specifies 2 as the default width for all borders, but overrides this value with 4 for the external border:

```
emacs.Borderwidth: 2
emacs.borderwidth: 4
```

The order in which the lines appear in the file does not matter. Also, command-line options always override the X resources file.

The string `emacs' in the examples above is also a resource name. It actually represents the name of the executable file that you invoke to run Emacs. If Emacs is installed under a different name, it looks for resources under that name instead of `emacs'.

`-name name'

`--name=name'

> Use name as the resource name (and the title) for the initial Emacs frame. This option does not affect subsequent frames, but Lisp programs can specify frame names when they create frames.
>
> If you don't specify this option, the default is to use the Emacs executable's name as the resource name.

`-xrm resource-values'

`--xrm=resource-values'

> Specify X resource values for this Emacs job (see below).

For consistency, `-name' also specifies the name to use for other resource values that do not belong to any particular frame.

The resources that name Emacs invocations also belong to a class; its name is `Emacs'. If you write `Emacs' instead of `emacs', the resource applies to all frames in all Emacs jobs, regardless of frame titles and regardless of the name of the executable file. Here is an example:

```
Emacs.BorderWidth: 2
Emacs.borderWidth: 4
```

You can specify a string of additional resource values for Emacs to use with the command line option `-xrm resources'. The text resources should have the same format that you would use inside a file of X resources. To include multiple resource specifications in data, put a newline between them, just as you would in a file. You can also use `#include "filename"' to include a file full of resource specifications. Resource values specified with `-xrm' take precedence over all other resource specifications.

The following table lists the resource names that designate options for Emacs, each with the class that it belongs to:

`background` (class `Background`)

Background color name.

`bitmapIcon` (class `BitmapIcon`)

Use a bitmap icon (a picture of a gnu) if `on', let the window manager choose an icon if `off'.

`borderColor` (class `BorderColor`)

Color name for the external border.

`borderWidth` (class `BorderWidth`)

Width in pixels of the external border.

`cursorColor` (class `Foreground`)

Color name for text cursor (point).

`font` (class `Font`)

Font name for text.

`foreground` (class `Foreground`)

Color name for text.

`geometry` (class `Geometry`)

Window size and position. Be careful not to specify this resource as `emacs*geometry', because that may affect individual menus as well as the Emacs frame itself.

If this resource specifies a position, that position applies only to the initial Emacs frame (or, in the case of a resource for a specific frame name, only that frame). However, the size if specified here applies to all frames.

`iconName` (class `Title`)

Name to display in the icon.

`internalBorder` (class `BorderWidth`)

Width in pixels of the internal border.

`menuBar` (class `MenuBar`)

Give frames menu bars if `on'; don't have menu bars if `off'.

`paneFont` (class `Font`)

Font name for menu pane titles, in non-toolkit versions of Emacs.

`pointerColor` (class `Foreground`)

Color of the mouse cursor.

`reverseVideo` (class `ReverseVideo`)

Switch foreground and background default colors if `on', use colors as specified if `off'.

`verticalScrollBars` (class `ScrollBars`)

Give frames scroll bars if `on'; don't have scroll bars if `off'.

`selectionFont` (class `Font`)

Font name for pop-up menu items, in non-toolkit versions of Emacs. (For toolkit versions, see section Lucid Menu X Resources, also see section Motif Menu X Resources.)

`title` (class `Title`)

Name to display in the title bar of the initial Emacs frame.

Here are resources for controlling the appearance of particular faces (see section Using Multiple Typefaces):

`face.attributeFont`

Font for face face.

`face.attributeForeground`

Foreground color for face face.

`face.attributeBackground`

Background color for face face.

`face.attributeUnderline`

Underline flag for face face. Use `on' or `true' for yes.

Lucid Menu X Resources

If the Emacs installed at your site was built to use the X toolkit with the Lucid menu widgets, then the menu bar is a separate widget and has its own resources. The resource names contain `pane.menubar' (following, as always, the name of the Emacs invocation or `Emacs' which stands for all Emacs invocations). Specify them like this:

```
Emacs.pane.menubar.resource:  value
```

For example, to specify the font `8x16' for the menu bar items, write this:

```
Emacs.pane.menubar.font:  8x16
```

Resources for *non-menubar* toolkit popup menus have `menu*', in like fashion. For example, to specify the font `8x16' for the popup menu items, write this:

```
Emacs.menu*.font:       8x16
```

For dialog boxes, use `dialog' instead of `menu':

```
Emacs.dialog*.font:     8x16
```

Experience shows that on some systems you may need to add `shell.' before the `pane.menubar' or `menu*'. On some other systems, you must not add `shell.'.

Here is a list of the specific resources for menu bars and popup menus:

TOC

`font`
: Font for menu item text.

`foreground`
: Color of the foreground.

`background`
: Color of the background.

`buttonForeground`
: In the menu bar, the color of the foreground for a selected item.

`horizontalSpacing`
: Horizontal spacing in pixels between items. Default is 3.

`verticalSpacing`
: Vertical spacing in pixels between items. Default is 1.

`arrowSpacing`
: Horizontal spacing between the arrow (which indicates a submenu) and the associated text. Default is 10.

`shadowThickness`
: Thickness of shadow line around the widget.

Motif Menu X Resources

If the Emacs installed at your site was built to use the X toolkit with the Motif widgets, then the menu bar is a separate widget and has its own resources. The resource names contain `pane.menubar' (following, as always, the name of the Emacs invocation or `Emacs' which stands for all Emacs invocations). Specify them like this:

```
Emacs.pane.menubar.subwidget.resource:  value
```

Each individual string in the menu bar is a subwidget; the subwidget's name is the same as the menu item string. For example, the word `Files' in the menu bar is part of a subwidget named `emacs.pane.menubar.Files'. Most likely, you want to specify the same resources for the whole menu bar. To do this, use `*' instead of a specific subwidget name. For example, to specify the font `8x16' for the menu bar items, write this:

```
Emacs.pane.menubar.*.fontList:  8x16
```

This also specifies the resource value for submenus.

Each item in a submenu in the menu bar also has its own name for X resources; for example, the `Files' submenu has an item named `Save Buffer'. A resource specification for a submenu item looks like this:

```
Emacs.pane.menubar.popup_*.menu.item.resource: value
```

For example, here's how to specify the font for the `Save Buffer' item:


```
Emacs.pane.menubar.popup_*.Files.Save Buffer.fontList: 8x16
```

For an item in a second-level submenu, such as `Check Message' under `Spell' under `Edit', the resource fits this template:

```
Emacs.pane.menubar.popup_*.popup_*.menu.resource: value
```

For example,

```
Emacs.pane.menubar.popup_*.popup_*.Spell.Check Message: value
```

It's impossible to specify a resource for all the menu bar items without also specifying it for the submenus as well. So if you want the submenu items to look different from the menu bar itself, you must ask for that in two steps. First, specify the resource for all of them; then, override the value for submenus alone. Here is an example:

```
Emacs.pane.menubar.*.fontList:  8x16
Emacs.pane.menubar.popup_*.fontList: 8x16
```

For toolkit popup menus, use `menu*' instead of `pane.menubar'. For example, to specify the font `8x16' for the popup menu items, write this:

```
Emacs.menu*.fontList:  8x16
```

Here is a list of the specific resources for menu bars and popup menus:

`armColor`
: The color to show in an armed button.

`fontList`
: The font to use.

`marginBottom`

`marginHeight`

`marginLeft`

`marginRight`

`marginTop`

`marginWidth`
: Amount of space to leave around the item, within the border.

`borderWidth`
: The width of border around the menu item, on all sides.

`shadowThickness`
: The width of the border shadow.

`bottomShadowColor`
: The color for the border shadow, on the bottom and the right.

`topShadowColor`

The color for the border shadow, on the bottom and the right.

Go to the previous, next section.

Go to the previous, next section.

Emacs 19.28 and 19.29 Antinews

For those users who live backwards in time, here is information about downgrading to Emacs version 19.28 or 19.29. We hope you will enjoy the greater simplicity that results from the absence of certain Emacs 19.34 features.

- This version doesn't support Windows NT, or the DEC Alpha.
- In Emacs 19.28, or even 19.29, the features for controlling indentation in C are much simpler. There is a separate variable for each aspect of indentation which you can control. Here is a list of them:

  `c-indent-level`
  : Level of indentation of C statements with respect to containing block.

  `c-label-offset`
  : Offset of C label lines and case statements relative to usual indentation.

  `c-continued-brace-offset`
  : Extra indentation for substatements that start with open-braces. This is in addition to `c-continued-statement-offset`.

  `c-continued-statement-offset`
  : Extra indentation for lines not starting new statements.

  `c-brace-offset`
  : Extra indentation for braces, compared with other text in same context.

  `c-brace-imaginary-offset`
  : Imagined indentation of a C open brace that actually follows a statement.

  `c-argdecl-indent`
  : Indentation level of declarations of C function arguments.
- There is no support for editing formatted text. The text formatter TeX does a much better job of formatting than any formatted text editor; we recommend you learn to use it.
- C-Mouse-2 now runs the menu for setting the default font.
- F1 is no longer an alias for the Help key; you must actually type C-h if you want help.
- Integers and buffer sizes are limited to 24 bits on most machines. But as memory gets more expensive, you won't want to edit such large files any more.
- There are no indirect buffers, so you can only display one view of an outline. Meanwhile, the prefix key for Outline minor mode is now C-c C-o.
- When you are in Transient Mark mode, incremental search always deactivates the mark.
- Dynamic abbrev completion has been eliminated in 19.28, and some of the other dynamic abbrev customization features are also gone.
- In Dired, Occur mode, Compilation mode, and other such modes, you must use C-c C-c to select

the item point is on. RET won't do it.

- M-x buffer-menu now displays the menu buffer in another window.
- The VC (version control) package no longer supports CVS or selecting branches other than the principal branch.
- There is no `recover-session` command; if Emacs crashes, you simply have to remember which files you were editing before the crash, and use `recover-file` on the individual files.
- In Emacs Lisp mode, C-M-x now lets `defvar` operate as it usually does--setting the value of the variable only if it has no value yet. Use ESC ESC to evaluate a Lisp expression, instead of M-:.
- GNU-standard long option names are not supported. (Real hackers prefer the shorter single-dash names, to save typing.) All the initial options must come before all the action options, and whatever initial options you use must appear in this order: `-t', `-d', `-nw', `-batch', `-q' or `-no-init-file', `-no-site-file', `-u' or `-user', `-debug-init'.
- Many special-case kludges for MS-DOS have been removed. This means that many features don't work on MS-DOS; however, the code of Emacs is much simpler.

  For instance, you cannot print from within Emacs, Dired doesn't support shell wildcards in filenames, some Lisp packages won't work because their standard filenames are invalid on MS-DOS, `display-time` doesn't work, Font Lock mode won't work unless you create some faces by hand first, and `call-process` cannot redirect `stderr`.

  As an additional bonus, you get random characters inserted into the buffer without a warning (unless you're clever enough to discover that setting `visible-bell` makes this problem to go away).

  In a word, it's a great relief for those who still need proof that MS-DOG isn't a real operating system.

Go to the previous, next section.

Go to the previous, next section.

MS-DOS Issues

This section briefly describes the peculiarities of using Emacs under the MS-DOS "operating system" (also known as "MS-DOG"). If you build Emacs for MS-DOS, the binary will also run on Windows 3, Windows NT, Windows 95, or OS-2 as a DOS application; the information in this chapter applies for all of those systems, if you use an Emacs that was built for MS-DOS.

Note that it is possible to build Emacs specifically for Windows NT or Windows 95. If you do that, most of this chapter does not apply; instead, you get behavior much closer to what is documented in the rest of the manual, including support for long file names, multiple frames, scroll bars, mouse menus, and subprocesses. However, the section on text files and binary files does still apply. There are also two sections at the end of this chapter which apply specifically for Windows NT and 95.

Keyboard and Mouse on MS-DOS

The PC keyboard maps use the left Alt key as the META key. You have two choices for emulating the SUPER and HYPER keys: either the right CONTROL key or the right ALT key by setting the variables `dos-hyper-key` and `dos-super-key` to 1 or 2 respectively.

The variable `dos-keypad-mode` is a flag variable which controls what key codes are returned by keys in the numeric keypad. There is no dedicated LFD key; use C-j instead. You can also define the kp-enter key to act as LFD, by putting the following line into your `` `_emacs' `` file:

```
;; Make the Enter key from the Numeric keypad act as LFD.
(define-key function-key-map [kp-enter] [?\C-j])
```

The key which is called DEL in Emacs (because that's how it is designated on most workstations) is known as BS (backspace) on a PC. That is why the PC-specific terminal initialization remaps the BS key to act as DEL; the DEL key is remapped to act as C-d for the same reasons.

Emacs on MS-DOS supports a mouse (on the default terminal only). The mouse commands work as documented, including those that use menus and the menu bar (see section Menu Bars). Scroll bars don't work in MS-DOS Emacs. PC mice usually have only two buttons; these act as Mouse-1 and Mouse-2, but if you press both of them together, that has the effect of Mouse-3.

The variable `dos-display-scancodes`, when non-`nil`, directs Emacs to display the ASCII value and the keyboard scan code of each keystroke; this feature serves as a complement to the `view-lossage` command, for debugging.

Display on MS-DOS

Display on MS-DOS cannot use multiple fonts, but it does support multiple faces, each of which can specify a foreground and a background color. Therefore, you can get the full functionality of Emacs packages which use fonts (such as `font-lock`, Enriched Text mode, and others) by defining the relevant faces to use different colors. Use the `list-colors-display` and `list-faces-display` commands (see section Modifying Faces) to see what colors and faces are available and what they look like.

Multiple frames (see section Frames and X Windows) are supported on MS-DOS, but they all overlap, so you only see a single frame at any given moment. That single visible frame occupies the entire screen. When you run Emacs under Windows version 3, you can make the visible frame smaller than the full screen, but Emacs still cannot display more than a single frame at a time.

The `mode4350` command switches the display to 43 or 50 lines, depending on your hardware; the `mode25` command switches to the default 80x25 screen size.

By default, Emacs only knows how to set screen sizes of 80 columns by 25 or 43/50 rows. However, if your video adapter has special video modes that will switch the display to other sizes, you can have Emacs support those too. When you ask Emacs to switch the frame to n rows by m cols dimensions, it checks if there is a variable called `screen-dimensions-nxm`, and if so, uses its value (which must be an integer) as the video mode to switch to. (Emacs switches to that video mode by calling the BIOS `Set Video Mode` function with the value of `screen-dimensions-nxm` in the `AL` register.) For example, suppose your adapter will switch to 66x80 dimensions when put into video mode 85. Then you can make Emacs support this screen size by putting the following into your `` `_emacs' `` file:

```
(setq screen-dimensions-66x80 85)
```

Since Emacs on MS-DOS can only set the frame size to specific supported dimensions, it cannot honor every possible frame resizing request. When an unsupported size is requested, Emacs chooses the next larger supported size beyond the specified size. For example, if you ask for 36x80 frame, you will get 50x80 instead.

The variables `screen-dimensions-nxm` are used only when they exactly match the specified size; the search for the next larger supported size ignores them. In the above example, even if your VGA supports 44x80 dimensions and you define a variable `screen-dimensions-44x80` with a suitable value, you will still get 50x80 screen when you ask for a 36x80 frame. If you want to get the 44x80 size in this case, you can do it by setting the variable named `screen-dimensions-36x80` with the same video mode value as `screen-dimensions-44x80`.

Changing frame dimensions on MS-DOS has the effect of changing all the other frames to the new dimensions.

File Names on MS-DOS

MS-DOS normally uses a backslash, `\', to separate name units within a file name, instead of the slash used on other systems. Emacs on MS-DOS permits use of either slash or backslash, and also knows about drive letters in file names.

On MS-DOS, file names are case-insensitive and limited to eight characters, plus optionally a period and three more characters. Emacs knows enough about these limitations to handle file names that were meant for other operating systems. For instance, leading dots `.' in file names are invalid in MS-DOS, so Emacs transparently converts them to underscores `_'; thus your default init file (see section The Init File, `~/.emacs') is called `_emacs' on MS-DOS. Excess characters before or after the period are generally ignored by MS-DOS itself, so if you, e.g., visit a file `LongFileName.EvenLongerExtension', you will silently get `longfile.eve'; but Emacs will still display the long file name on the mode line. Other than that, it's up to you to specify file names which are valid under MS-DOS; the transparent conversion as described above only works on file names built into Emacs.

The above restrictions on the file names on MS-DOS make it almost impossible to construct the name of a backup file (see section Single or Numbered Backups) without losing some of the original file name characters. For example, the name of a backup file for `docs.txt' is `docs.tx~' even if single backup is used.

If you run Emacs as a DOS application under Windows 95 or NT, you can turn on support for long file names. If you do that, Emacs doesn't truncate file names or convert them to lower case; instead, it uses the file names that you specify, verbatim. To enable long file name support, set the environment variable `LFN` to `y' before starting Emacs.

MS-DOS has no notion of home directory, so Emacs on MS-DOS pretends that the directory where it is installed is the value of `HOME` environment variable. That is, if your Emacs binary, `emacs.exe', is in the directory `c:/utils/emacs/bin', then Emacs acts as if `HOME` were set to `c:/utils/emacs'. In particular, that is where Emacs looks for the init file `_emacs'. With this in mind, you can use `~' in file names as an alias for the home directory, as you would in Unix. You can also set `HOME` variable in the environment before starting Emacs; its value will then override the above default behavior.

Text Files and Binary Files

Emacs on MS-DOS distinguishes between text and binary files. This distinction is not part of MS-DOS; it is made by Emacs only. Emacs treats files of human-readable text (including program source code) as text files, and treats executable programs, compressed archives, etc., as binary files. Emacs uses the file name to decide whether to treat a file as text or binary: the variable `file-name-buffer-file-type-alist` defines the file name patterns which denote binary files.

Emacs reads and writes binary files verbatim. Text files use a two character sequence to end a line: carriage-return (control-m) followed by newline (control-j). When you visit a text file, Emacs strips off these control-m characters; when you write a text file to disk, Emacs puts them back in. Thus, the text

appears within Emacs with just a newline character at the end of each line.

You can tell whether Emacs considers the visited file as text or binary based on the mode line (see section The Mode Line). Text files have a `T:' marker prefixed to the major mode name; binary files have a `B:' prefix.

One consequence of this special format-conversion of text files is that character positions as reported by Emacs (see section Cursor Position Information) do not agree with the file size information known to the operating system.

Printing and MS-DOS

Printing commands, such as `lpr-buffer` (see section Hardcopy Output) and `ps-print-buffer` (see section Postscript Hardcopy) can work in MS-DOS by sending the output to one of the printer ports, if a Unix-style `lpr` program is unavailable. A few DOS-specific variables control how this works.

If you want to use your local printer, printing on it in the usual DOS manner, then set the Lisp variable `dos-printer` to the name of the printer port--for example. `"PRN"`, the usual local printer port (that's the default), or `"LPT2"` or `"COM1"` for a serial printer. You can also set `dos-printer` to a file name, in which case "printed" output is actually appended to that file. If you set `dos-printer` to `"NUL"`, printed output is silently discarded.

If you set `dos-printer` to a file name, it's best to use an absolute file name. Emacs changes the working directory according to the default directory of the current buffer, so if the file name in `dos-printer` is relative, you will end up with several such files, each one in the directory of the buffer from which the printing was done.

The commands `print-buffer` and `print-region` call the `pr` program, or use special switches to the `lpr` program, to produce headers on each printed page. MS-DOS doesn't normally have these programs, so by default, the variable `lpr-headers-switches` is set so that the requests to print page headers are silently ignored. Thus, `print-buffer` and `print-region` produce the same output as `lpr-buffer` and `lpr-region`, respectively. If you do have a suitable `pr` program (e.g., from GNU Textutils), set `lpr-headers-switches` to `nil`; Emacs will then call `pr` to produce the page headers, and print the resulting output as specified by `dos-printer`.

Finally, if you do have an `lpr` work-alike, you can set `print-region-function` to `nil`. Then Emacs uses `lpr` for printing, as on other systems. (If the name of the program isn't `lpr`, set the `lpr-command` variable to specify where to find it.)

A separate variable, `dos-ps-printer`, defines how PostScript files should be printed. If its value is a string, it is used as the name of the device (or file) to which PostScript output is sent, just as `dos-printer` is used for non-PostScript printing. (These are two distinct variables in case you have two printers attached to two different ports, and only one of them is a PostScript printer.) If the value of `dos-ps-printer` is not a string, then the variables `ps-lpr-command` and `ps-lpr-switches` (see section Postscript Hardcopy) control how to print PostScript files. Thus, if you have a non-PostScript printer, you can set these variables to the name and the switches appropriate for a PostScript interpreter program (e.g., Ghostscript).

For example, to use Ghostscript for printing on an Epson printer connected to `LPT2' port, put this c
your `.emacs' file:

```
(setq dos-ps-printer t)  ; Anything but a string.
(setq ps-lpr-command "c:/gs/gs386")
(setq ps-lpr-switches '("-q" "-dNOPAUSE"
                        "-sDEVICE=epson"
                        "-r240x72"
                        "-sOutputFile=LPT2"
                        "-Ic:/gs"
                        "-"))
```

(This assumes that Ghostscript is installed in the `"c:/gs"' directory.)

Subprocesses on MS-DOS

Because MS-DOS is a single-process "operating system", asynchronous subprocesses are not available. In particular, Shell mode and its variants do not work. Most Emacs features that use asynchronous subprocesses also don't work on MS-DOS, including spelling correction and GUD. When in doubt, try and see; commands that don't work print an error message saying that asynchronous processes aren't supported.

Compilation under Emacs with M-x compile and grep with M-x grep do work, by running the inferior processes synchronously. This means you cannot do any more editing until the compilation or the grep process finishes.

Printing commands, such as `lpr-buffer` (see section Hardcopy Output) and `ps-print-buffer` (see section Postscript Hardcopy), work in MS-DOS by sending the output to one of the printer ports. See section Printing and MS-DOS.

When you run a subprocess synchronously on MS-DOS, make sure the program terminates and does not try to read keyboard input. If the program does not terminate on its own, you will be unable to terminate it, because MS-DOS provides no general way to terminate a process.

Accessing files on other machines is not supported on MS-DOS. Other network-oriented commands such as sending mail, Web browsing, remote login, etc., don't work either, unless network access is built into MS-DOS with some network redirector.

Dired on MS-DOS uses the `ls-lisp` package where other platforms use the system `ls` command. Therefore, Dired on MS-DOS supports only some of the possible options you can mention in the `dired-listing-switches` variable. The options that work are `-A', `-a', `-c', `-i', `-r', `-S', `-s', `-t', and `-u'.

Subprocesses on Windows 95 and Windows NT

Subprocesses, both synchronous and asynchronous, work fine on both Windows 95 and Windows NT as long as you run only 32-bit Windows applications. However, when you run a DOS application in a subprocess, you may encounter problems or be unable to run the application at all; and if you run two DOS applications at the same time in two subprocesses, you may have to reboot your system.

Since the standard command interpreter (and most command line utilities) on Windows 95 are DOS applications, these problems are significant when using that system. But there's nothing we can do about them; only Microsoft can fix them.

If you run just one DOS application subprocess, the subprocess should work as expected as long as it is "well-behaved" and does not perform direct screen access or other unusual actions. If you have a CPU monitor application, your machine will appear to be 100% busy even when the DOS application is idle, but this is only an artefact of the way CPU monitors measure processor load.

You must terminate the DOS application before you start any other DOS application in a different subprocess. Emacs is unable to interrupt or terminate a DOS subprocess. The only way you can terminate such a subprocess is by giving it a command that tells its program to exit.

If you attempt to run two DOS applications at the same time in separate subprocesses, the second one that is started will be suspended until the first one finishes, even if either or both of them are asynchronous.

If you can go to the first subprocess, and tell it to exit, the second subprocess should continue normally. However, if the second subprocess is synchronous, Emacs itself will be hung until the first subprocess finishes. If it will not finish without user input, then you have no choice but to reboot if you are running on Windows 95. If you are running on Windows NT, you can use a process viewer application to kill the appropriate instance of ntvdm instead (this will terminate both DOS subprocesses).

If you have to reboot Windows 95 in this situation, do not use the `Shutdown` command on the `Start` menu; that usually hangs the system. Instead, type CTL-ALT-DEL and then choose `Shutdown`. That usually works, although it may take a few minutes to do its job.

Using the System Menu on Windows

Emacs normally turns off the Windows feature that tapping the ALT key invokes the Windows menu. The reason is that the ALT also serves as META in Emacs. When using Emacs, users often press the META key temporarily and then change their minds; if this has the effect of bringing up the Windows menu, it alters the meaning of subsequent commands. Many users find this frustrating.

You can reenable Windows's default handling of tapping the ALT key by setting `win32-pass-alt-to-system` to a non-`nil` value.

Go to the previous, next section.

Go to the previous, next section.

The GNU Manifesto

The GNU Manifesto which appears below was written by Richard Stallman at the beginning of the GNU project, to ask for participation and support. For the first few years, it was updated in minor ways to account for developments, but now it seems best to leave it unchanged as most people have seen it.

Since that time, we have learned about certain common misunderstandings that different wording could help avoid. Footnotes added in 1993 help clarify these points.

For up-to-date information about the available GNU software, please see the latest issue of the GNU's Bulletin. The list is much too long to include here.

What's GNU? Gnu's Not Unix!

GNU, which stands for Gnu's Not Unix, is the name for the complete Unix-compatible software system which I am writing so that I can give it away free to everyone who can use it.(5) Several other volunteers are helping me. Contributions of time, money, programs and equipment are greatly needed.

So far we have an Emacs text editor with Lisp for writing editor commands, a source level debugger, a yacc-compatible parser generator, a linker, and around 35 utilities. A shell (command interpreter) is nearly completed. A new portable optimizing C compiler has compiled itself and may be released this year. An initial kernel exists but many more features are needed to emulate Unix. When the kernel and compiler are finished, it will be possible to distribute a GNU system suitable for program development. We will use TeX as our text formatter, but an nroff is being worked on. We will use the free, portable X window system as well. After this we will add a portable Common Lisp, an Empire game, a spreadsheet, and hundreds of other things, plus on-line documentation. We hope to supply, eventually, everything useful that normally comes with a Unix system, and more.

GNU will be able to run Unix programs, but will not be identical to Unix. We will make all improvements that are convenient, based on our experience with other operating systems. In particular, we plan to have longer file names, file version numbers, a crashproof file system, file name completion perhaps, terminal-independent display support, and perhaps eventually a Lisp-based window system through which several Lisp programs and ordinary Unix programs can share a screen. Both C and Lisp will be available as system programming languages. We will try to support UUCP, MIT Chaosnet, and Internet protocols for communication.

GNU is aimed initially at machines in the 68000/16000 class with virtual memory, because they are the easiest machines to make it run on. The extra effort to make it run on smaller machines will be left to someone who wants to use it on them.

To avoid horrible confusion, please pronounce the `G' in the word `GNU' when it is the name of this project.

Why I Must Write GNU

I consider that the golden rule requires that if I like a program I must share it with other people who like it. Software sellers want to divide the users and conquer them, making each user agree not to share with others. I refuse to break solidarity with other users in this way. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. For years I worked within the Artificial Intelligence Lab to resist such tendencies and other inhospitalities, but eventually they had gone too far: I could not remain in an institution where such things are done for me against my will.

So that I can continue to use computers without dishonor, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free. I have resigned from the AI lab to deny MIT any legal excuse to prevent me from giving GNU away.

Why GNU Will Be Compatible with Unix

Unix is not my ideal system, but it is not too bad. The essential features of Unix seem to be good ones, and I think I can fill in what Unix lacks without spoiling them. And a system compatible with Unix would be convenient for many other people to adopt.

How GNU Will Be Available

GNU is not in the public domain. Everyone will be permitted to modify and redistribute GNU, but no distributor will be allowed to restrict its further redistribution. That is to say, proprietary modifications will not be allowed. I want to make sure that all versions of GNU remain free.

Why Many Other Programmers Want to Help

I have found many other programmers who are excited about GNU and want to help.

Many programmers are unhappy about the commercialization of system software. It may enable them to make more money, but it requires them to feel in conflict with other programmers in general rather than feel as comrades. The fundamental act of friendship among programmers is the sharing of programs; marketing arrangements now typically used essentially forbid programmers to treat others as friends. The purchaser of software must choose between friendship and obeying the law. Naturally, many decide that friendship is more important. But those who believe in law often do not feel at ease with either choice. They become cynical and think that programming is just a way of making money.

By working on and using GNU rather than proprietary programs, we can be hospitable to everyone and obey the law. In addition, GNU serves as an example to inspire and a banner to rally others to join us in sharing. This can give us a feeling of harmony which is impossible if we use software that is not free. For about half the programmers I talk to, this is an important happiness that money cannot replace.

How You Can Contribute

I am asking computer manufacturers for donations of machines and money. I'm asking individuals for donations of programs and work.

One consequence you can expect if you donate machines is that GNU will run on them at an early date. The machines should be complete, ready to use systems, approved for use in a residential area, and not in need of sophisticated cooling or power.

I have found very many programmers eager to contribute part-time work for GNU. For most projects, such part-time distributed work would be very hard to coordinate; the independently-written parts would not work together. But for the particular task of replacing Unix, this problem is absent. A complete Unix system contains hundreds of utility programs, each of which is documented separately. Most interface specifications are fixed by Unix compatibility. If each contributor can write a compatible replacement for a single Unix utility, and make it work properly in place of the original on a Unix system, then these utilities will work right when put together. Even allowing for Murphy to create a few unexpected problems, assembling these components will be a feasible task. (The kernel will require closer communication and will be worked on by a small, tight group.)

If I get donations of money, I may be able to hire a few people full or part time. The salary won't be high by programmers' standards, but I'm looking for people for whom building community spirit is as important as making money. I view this as a way of enabling dedicated people to devote their full energies to working on GNU by sparing them the need to make a living in another way.

Why All Computer Users Will Benefit

Once GNU is written, everyone will be able to obtain good system software free, just like air.(6)

This means much more than just saving everyone the price of a Unix license. It means that much wasteful duplication of system programming effort will be avoided. This effort can go instead into advancing the state of the art.

Complete system sources will be available to everyone. As a result, a user who needs changes in the system will always be free to make them himself, or hire any available programmer or company to make them for him. Users will no longer be at the mercy of one programmer or company which owns the sources and is in sole position to make changes.

Schools will be able to provide a much more educational environment by encouraging all students to study and improve the system code. Harvard's computer lab used to have the policy that no program could be installed on the system if its sources were not on public display, and upheld it by actually refusing to install certain programs. I was very much inspired by this.

Finally, the overhead of considering who owns the system software and what one is or is not entitled to do with it will be lifted.

Arrangements to make people pay for using a program, including licensing of copies, always incur a tremendous cost to society through the cumbersome mechanisms necessary to figure out how much (that

is, which programs) a person must pay for. And only a police state can force everyone to obey them. Consider a space station where air must be manufactured at great cost: charging each breather per li air may be fair, but wearing the metered gas mask all day and all night is intolerable even if everyone can afford to pay the air bill. And the TV cameras everywhere to see if you ever take the mask off are outrageous. It's better to support the air plant with a head tax and chuck the masks.

Copying all or parts of a program is as natural to a programmer as breathing, and as productive. It ought to be as free.

Some Easily Rebutted Objections to GNU's Goals

> "Nobody will use it if it is free, because that means they can't rely on any support."
>
> "You have to charge for the program to pay for providing the support."

If people would rather pay for GNU plus service than get GNU free without service, a company to provide just service to people who have obtained GNU free ought to be profitable.(7)

We must distinguish between support in the form of real programming work and mere handholding. The former is something one cannot rely on from a software vendor. If your problem is not shared by enough people, the vendor will tell you to get lost.

If your business needs to be able to rely on support, the only way is to have all the necessary sources and tools. Then you can hire any available person to fix your problem; you are not at the mercy of any individual. With Unix, the price of sources puts this out of consideration for most businesses. With GNU this will be easy. It is still possible for there to be no available competent person, but this problem cannot be blamed on distribution arrangements. GNU does not eliminate all the world's problems, only some of them.

Meanwhile, the users who know nothing about computers need handholding: doing things for them which they could easily do themselves but don't know how.

Such services could be provided by companies that sell just hand-holding and repair service. If it is true that users would rather spend money and get a product with service, they will also be willing to buy the service having got the product free. The service companies will compete in quality and price; users will not be tied to any particular one. Meanwhile, those of us who don't need the service should be able to use the program without paying for the service.

> "You cannot reach many people without advertising, and you must charge for the program to support that."
>
> "It's no use advertising a program people can get free."

There are various forms of free or very cheap publicity that can be used to inform numbers of computer users about something like GNU. But it may be true that one can reach more microcomputer users with advertising. If this is really so, a business which advertises the service of copying and mailing GNU for a fee ought to be successful enough to pay for its advertising and more. This way, only the users who benefit from the advertising pay for it.

On the other hand, if many people get GNU from their friends, and such companies don't succeed, th will show that advertising was not really necessary to spread GNU. Why is it that free market advoca don't want to let the free market decide this?(8)

"My company needs a proprietary operating system to get a competitive edge."

GNU will remove operating system software from the realm of competition. You will not be able to get an edge in this area, but neither will your competitors be able to get an edge over you. You and they will compete in other areas, while benefiting mutually in this one. If your business is selling an operating system, you will not like GNU, but that's tough on you. If your business is something else, GNU can save you from being pushed into the expensive business of selling operating systems.

I would like to see GNU development supported by gifts from many manufacturers and users, reducing the cost to each.(9)

"Don't programmers deserve a reward for their creativity?"

If anything deserves a reward, it is social contribution. Creativity can be a social contribution, but only in so far as society is free to use the results. If programmers deserve to be rewarded for creating innovative programs, by the same token they deserve to be punished if they restrict the use of these programs.

"Shouldn't a programmer be able to ask for a reward for his creativity?"

There is nothing wrong with wanting pay for work, or seeking to maximize one's income, as long as one does not use means that are destructive. But the means customary in the field of software today are based on destruction.

Extracting money from users of a program by restricting their use of it is destructive because the restrictions reduce the amount and the ways that the program can be used. This reduces the amount of wealth that humanity derives from the program. When there is a deliberate choice to restrict, the harmful consequences are deliberate destruction.

The reason a good citizen does not use such destructive means to become wealthier is that, if everyone did so, we would all become poorer from the mutual destructiveness. This is Kantian ethics; or, the Golden Rule. Since I do not like the consequences that result if everyone hoards information, I am required to consider it wrong for one to do so. Specifically, the desire to be rewarded for one's creativity does not justify depriving the world in general of all or part of that creativity.

"Won't programmers starve?"

I could answer that nobody is forced to be a programmer. Most of us cannot manage to get any money for standing on the street and making faces. But we are not, as a result, condemned to spend our lives standing on the street making faces, and starving. We do something else.

But that is the wrong answer because it accepts the questioner's implicit assumption: that without ownership of software, programmers cannot possibly be paid a cent. Supposedly it is all or nothing.

The real reason programmers will not starve is that it will still be possible for them to get paid for programming; just not paid as much as now.

Restricting copying is not the only basis for business in software. It is the most common basis because it brings in the most money. If it were prohibited, or rejected by the customer, software business would

move to other bases of organization which are now used less often. There are always numerous ways to organize any kind of business.

Probably programming will not be as lucrative on the new basis as it is now. But that is not an argument against the change. It is not considered an injustice that sales clerks make the salaries that they now do. If programmers made the same, that would not be an injustice either. (In practice they would still make considerably more than that.)

"Don't people have a right to control how their creativity is used?"

"Control over the use of one's ideas" really constitutes control over other people's lives; and it is usually used to make their lives more difficult.

People who have studied the issue of intellectual property rights carefully (such as lawyers) say that there is no intrinsic right to intellectual property. The kinds of supposed intellectual property rights that the government recognizes were created by specific acts of legislation for specific purposes.

For example, the patent system was established to encourage inventors to disclose the details of their inventions. Its purpose was to help society rather than to help inventors. At the time, the life span of 17 years for a patent was short compared with the rate of advance of the state of the art. Since patents are an issue only among manufacturers, for whom the cost and effort of a license agreement are small compared with setting up production, the patents often do not do much harm. They do not obstruct most individuals who use patented products.

The idea of copyright did not exist in ancient times, when authors frequently copied other authors at length in works of non-fiction. This practice was useful, and is the only way many authors' works have survived even in part. The copyright system was created expressly for the purpose of encouraging authorship. In the domain for which it was invented--books, which could be copied economically only on a printing press--it did little harm, and did not obstruct most of the individuals who read the books.

All intellectual property rights are just licenses granted by society because it was thought, rightly or wrongly, that society as a whole would benefit by granting them. But in any particular situation, we have to ask: are we really better off granting such license? What kind of act are we licensing a person to do?

The case of programs today is very different from that of books a hundred years ago. The fact that the easiest way to copy a program is from one neighbor to another, the fact that a program has both source code and object code which are distinct, and the fact that a program is used rather than read and enjoyed, combine to create a situation in which a person who enforces a copyright is harming society as a whole both materially and spiritually; in which a person should not do so regardless of whether the law enables him to.

"Competition makes things get done better."

The paradigm of competition is a race: by rewarding the winner, we encourage everyone to run faster. When capitalism really works this way, it does a good job; but its defenders are wrong in assuming it always works this way. If the runners forget why the reward is offered and become intent on winning, no matter how, they may find other strategies--such as, attacking other runners. If the runners get into a fist fight, they will all finish late.

Proprietary and secret software is the moral equivalent of runners in a fist fight. Sad to say, the only

referee we've got does not seem to object to fights; he just regulates them ("For every ten yards you r
you can fire one shot"). He really ought to break them up, and penalize runners for even trying to fig

> "Won't everyone stop programming without a monetary incentive?"

Actually, many people will program with absolutely no monetary incentive. Programming has an irresistible fascination for some people, usually the people who are best at it. There is no shortage of professional musicians who keep at it even though they have no hope of making a living that way.

But really this question, though commonly asked, is not appropriate to the situation. Pay for programmers will not disappear, only become less. So the right question is, will anyone program with a reduced monetary incentive? My experience shows that they will.

For more than ten years, many of the world's best programmers worked at the Artificial Intelligence Lab for far less money than they could have had anywhere else. They got many kinds of non-monetary rewards: fame and appreciation, for example. And creativity is also fun, a reward in itself.

Then most of them left when offered a chance to do the same interesting work for a lot of money.

What the facts show is that people will program for reasons other than riches; but if given a chance to make a lot of money as well, they will come to expect and demand it. Low-paying organizations do poorly in competition with high-paying ones, but they do not have to do badly if the high-paying ones are banned.

> "We need the programmers desperately. If they demand that we stop helping our neighbors, we have to obey."

You're never so desperate that you have to obey this sort of demand. Remember: millions for defense, but not a cent for tribute!

> "Programmers need to make a living somehow."

In the short run, this is true. However, there are plenty of ways that programmers could make a living without selling the right to use a program. This way is customary now because it brings programmers and businessmen the most money, not because it is the only way to make a living. It is easy to find other ways if you want to find them. Here are a number of examples.

A manufacturer introducing a new computer will pay for the porting of operating systems onto the new hardware.

The sale of teaching, hand-holding and maintenance services could also employ programmers.

People with new ideas could distribute programs as freeware, asking for donations from satisfied users, or selling hand-holding services. I have met people who are already working this way successfully.

Users with related needs can form users' groups, and pay dues. A group would contract with programming companies to write programs that the group's members would like to use.

All sorts of development can be funded with a Software Tax:

> Suppose everyone who buys a computer has to pay x percent of the price as a software tax. The government gives this to an agency like the NSF to spend on software development.

But if the computer buyer makes a donation to software development himself, he can take a credit against the tax. He can donate to the project of his own choosing--often, chosen because he hopes to use the results when it is done. He can take a credit for any amount of donation up to the total tax he had to pay.

The total tax rate could be decided by a vote of the payers of the tax, weighted according to the amount they will be taxed on.

The consequences:

- The computer-using community supports software development.
- This community decides what level of support is needed.
- Users who care which projects their share is spent on can choose this for themselves.

In the long run, making programs free is a step toward the post-scarcity world, where nobody will have to work very hard just to make a living. People will be free to devote themselves to activities that are fun, such as programming, after spending the necessary ten hours a week on required tasks such as legislation, family counseling, robot repair and asteroid prospecting. There will be no need to be able to make a living from programming.

We have already greatly reduced the amount of work that the whole society must do for its actual productivity, but only a little of this has translated itself into leisure for workers because much nonproductive activity is required to accompany productive activity. The main causes of this are bureaucracy and isometric struggles against competition. Free software will greatly reduce these drains in the area of software production. We must do this, in order for technical gains in productivity to translate into less work for us.

Go to the previous, next section.

Go to the previous, next section.

Glossary

Abbrev

An abbrev is a text string which expands into a different text string when present in the buffer. For example, you might define a few letters as an abbrev for a long phrase that you want to insert frequently. See section Abbrevs.

Aborting

Aborting means getting out of a recursive edit (q.v.). The commands C-] and M-x top-level are used for this. See section Quitting and Aborting.

Alt

Alt is the name of a modifier bit which a keyboard input character may have. To make a character Alt, type it while holding down the ALT key. Such characters are given names that start with Alt- (usually written A- for short). (Note that many terminals have a key labeled ALT which is really a META key.) See section Kinds of User Input.

Auto Fill Mode

Auto Fill mode is a minor mode in which text that you insert is automatically broken into lines of fixed width. See section Filling Text.

Auto Saving

Auto saving is the practice of saving the contents of an Emacs buffer in a specially-named file, so that the information will not be lost if the buffer is lost due to a system error or user error. See section Auto-Saving: Protection Against Disasters.

Backup File

A backup file records the contents that a file had before the current editing session. Emacs makes backup files automatically to help you track down or cancel changes you later regret making. See section Backup Files.

Balance Parentheses

Emacs can balance parentheses manually or automatically. Manual balancing is done by the commands to move over balanced expressions (see section Lists and Sexps). Automatic balancing is done by blinking or highlighting the parenthesis that matches one just inserted (see section Automatic Display Of Matching Parentheses).

Bind

To bind a key sequence means to give it a binding (q.v.). See section Changing Key Bindings Interactively.

Binding

A key sequence gets its meaning in Emacs by having a binding, which is a command (q.v.), a Lisp function that is run when the user types that sequence. See section Keys and Commands.

Customization often involves rebinding a character to a different command function. The bind of all key sequences are recorded in the keymaps (q.v.). See section Keymaps.

Blank Lines

Blank lines are lines that contain only whitespace. Emacs has several commands for operating on the blank lines in the buffer.

Buffer

The buffer is the basic editing unit; one buffer corresponds to one text being edited. You can have several buffers, but at any time you are editing only one, the `selected' buffer, though several can be visible when you are using multiple windows (q.v.). Most buffers are visiting (q.v.) some file. See section Using Multiple Buffers.

Buffer Selection History

Emacs keeps a buffer selection history which records how recently each Emacs buffer has been selected. This is used for choosing a buffer to select. See section Using Multiple Buffers.

Button Down Event

A button down event is the kind of input event generated right away when you press a mouse button. See section Rebinding Mouse Buttons.

C-

C- in the name of a character is an abbreviation for Control. See section Kinds of User Input.

C-M-

C-M- in the name of a character is an abbreviation for Control-Meta. See section Kinds of User Input.

Case Conversion

Case conversion means changing text from upper case to lower case or vice versa. See section Case Conversion Commands, for the commands for case conversion.

Character

Characters form the contents of an Emacs buffer; see section Character Set for Text. Also, key sequences (q.v.) are usually made up of characters (though they may include other input events as well). See section Kinds of User Input.

Click Event

A click event is the kind of input event generated when you press a mouse button and release it without moving the mouse. See section Rebinding Mouse Buttons.

Command

A command is a Lisp function specially defined to be able to serve as a key binding in Emacs. When you type a key sequence (q.v.), its binding (q.v.) is looked up in the relevant keymaps (q.v.) to find the command to run. See section Keys and Commands.

Command Name

A command name is the name of a Lisp symbol which is a command (see section Keys and Commands). You can invoke any command by its name using M-x (see section Running

Commands by Name).

Comment

A comment is text in a program which is intended only for humans reading the program, and which is marked specially so that it will be ignored when the program is loaded or compiled. Emacs offers special commands for creating, aligning and killing comments. See section Manipulating Comments.

Compilation

Compilation is the process of creating an executable program from source code. Emacs has commands for compiling files of Emacs Lisp code (see section 'Byte Compilation' in the Emacs Lisp Reference Manual) and programs in C and other languages (see section Running Compilations under Emacs).

Complete Key

A complete key is a key sequence which fully specifies one action to be performed by Emacs. For example, X and C-f and C-x m are complete keys. Complete keys derive their meanings from being bound (q.v.) to commands (q.v.). Thus, X is conventionally bound to a command to insert `X' in the buffer; C-x m is conventionally bound to a command to begin composing a mail message. See section Keys.

Completion

Completion is what Emacs does when it automatically fills out an abbreviation for a name into the entire name. Completion is done for minibuffer (q.v.) arguments when the set of possible valid inputs is known; for example, on command names, buffer names, and file names. Completion occurs when TAB, SPC or RET is typed. See section Completion.

Continuation Line

When a line of text is longer than the width of the window, it takes up more than one screen line when displayed. We say that the text line is continued, and all screen lines used for it after the first are called continuation lines. See section Basic Editing Commands.

Control Character

ASCII characters with octal codes 0 through 037, and also code 0177, do not have graphic images assigned to them. These are the Control characters. To type a Control character, hold down the CTRL key and type the corresponding non-Control character. RET, TAB, ESC, LFD and DEL are all control characters. See section Kinds of User Input.

When you are using the X Window System, every non-control character has a corresponding control character variant.

Copyleft

A copyleft is a notice giving the public legal permission to redistribute a program or other work of art. Copylefts are used by left-wing programmers to give people equal rights, just as copyrights are used by right-wing programmers to gain power over other people.

The particular form of copyleft used by the GNU project is called the GNU General Public License. See section GNU GENERAL PUBLIC LICENSE.

Current Buffer

The current buffer in Emacs is the Emacs buffer on which most editing commands operate. You can select any Emacs buffer as the current one. See section Using Multiple Buffers.

Current Line

The line point is on (see section Point).

Current Paragraph

The paragraph that point is in. If point is between paragraphs, the current paragraph is the one that follows point. See section Paragraphs.

Current Defun

The defun (q.v.) that point is in. If point is between defuns, the current defun is the one that follows point. See section Defuns.

Cursor

The cursor is the rectangle on the screen which indicates the position called point (q.v.) at which insertion and deletion takes place. The cursor is on or under the character that follows point. Often people speak of `the cursor' when, strictly speaking, they mean `point'. See section Basic Editing Commands.

Customization

Customization is making minor changes in the way Emacs works. It is often done by setting variables (see section Variables) or by rebinding key sequences (see section Keymaps).

Default Argument

The default for an argument is the value that will be assumed if you do not specify one. When the minibuffer is used to read an argument, the default argument is used if you just type RET. See section The Minibuffer.

Default Directory

When you specify a file name that does not start with `/' or `~', it is interpreted relative to the current buffer's default directory. See section Minibuffers for File Names.

Defun

A defun is a list at the top level of parenthesis or bracket structure in a program. It is so named because most such lists in Lisp programs are calls to the Lisp function `defun`. See section Defuns.

DEL

DEL is a character that runs the command to delete one character of text. See section Basic Editing Commands.

Deletion

Deletion means erasing text without copying it into the kill ring (q.v.). The alternative is killing (q.v.). See section Deletion and Killing.

Deletion of Files

Deleting a file means erasing it from the file system. See section Miscellaneous File Operations.

Deletion of Messages

Deleting a message means flagging it to be eliminated from your mail file. Until you expunge (q.v.) the Rmail file, you can still undelete the messages you have deleted. See section Deleting Messages.

Deletion of Windows

Deleting a window means eliminating it from the screen. Other windows expand to use up the space. The deleted window can never come back, but no actual text is thereby lost. See section Multiple Windows.

Directory

File directories are named collections in the file system, within which you can place individual files or subdirectories. See section File Directories.

Dired

Dired is the Emacs facility that displays the contents of a file directory and allows you to "edit the directory", performing operations on the files in the directory. See section Dired, the Directory Editor.

Disabled Command

A disabled command is one that you may not run without special confirmation. The usual reason for disabling a command is that it is confusing for beginning users. See section Disabling Commands.

Down Event

Short for `button down event'.

Drag Event

A drag event is the kind of input event generated when you press a mouse button, move the mouse, and then release the button. See section Rebinding Mouse Buttons.

Dribble File

A file into which Emacs writes all the characters that the user types on the keyboard. Dribble files are used to make a record for debugging Emacs bugs. Emacs does not make a dribble file unless you tell it to. See section Reporting Bugs.

Echo Area

The echo area is the bottom line of the screen, used for echoing the arguments to commands, for asking questions, and printing brief messages (including error messages). The messages are stored in the buffer `*Messages*' so you can review them later. See section The Echo Area.

Echoing

Echoing is acknowledging the receipt of commands by displaying them (in the echo area). Emacs never echoes single-character key sequences; longer key sequences echo only if you pause while typing them.

Error

An error occurs when an Emacs command cannot execute in the current circumstances. When an error occurs, execution of the command stops (unless the command has been programmed to do

otherwise) and Emacs reports the error by printing an error message (q.v.). Type-ahead is discarded. Then Emacs is ready to read another editing command.

Error Message

An error message is a single line of output displayed by Emacs when the user asks for something impossible to do (such as, killing text forward when point is at the end of the buffer). They appear in the echo area, accompanied by a beep.

ESC

ESC is a character used as a prefix for typing Meta characters on keyboards lacking a META key. Unlike the META key (which, like the SHIFT key, is held down while another character is typed), the ESC key is pressed once and applies to the next character typed.

Expunging

Expunging an Rmail file or Dired buffer means really discarding the messages or files you have previously flagged for deletion.

File Name

A file name is a name that refers to a file. File names may be relative or absolute; the meaning of a relative file name depends on the current directory, but an absolute file name refers to the same file regardless of which directory is current. On GNU and Unix systems, an absolute file name starts with a slash (the root directory) or with `~/' or `~user/' (a home directory).

Some people use the term "pathname" for file names, but we do not; we use the word "path" only in the term "search path" (q.v.).

File Name Component

A file name component names a file directly within a particular directory. On GNU and Unix systems, a file name is a sequence of file name components, separated by slashes. For example, `foo/bar' is a file name containing two components, `foo' and `bar'; it refers to the file named `bar' in the directory named `foo' in the current directory.

Fill Prefix

The fill prefix is a string that should be expected at the beginning of each line when filling is done. It is not regarded as part of the text to be filled. See section Filling Text.

Filling

Filling text means shifting text between consecutive lines so that all the lines are approximately the same length. See section Filling Text.

Formatted Text

Formatted text is text that displays with formatting information while you edit. Formatting information includes fonts, colors, and specified margins. See section Editing Formatted Text.

Frame

A frame is a rectangular cluster of Emacs windows. Emacs starts out with one frame, but you can create more. You can subdivide each frame into Emacs windows (q.v.). When you are using X windows, all the frames can be visible at the same time. See section Frames and X Windows.

Function Key

A function key is a key on the keyboard that sends input but does not correspond to any chara
See section Rebinding Function Keys.

Global

Global means `independent of the current environment; in effect throughout Emacs'. It is the opposite of local (q.v.). Particular examples of the use of `global' appear below.

Global Abbrev

A global definition of an abbrev (q.v.) is effective in all major modes that do not have local (q.v.) definitions for the same abbrev. See section Abbrevs.

Global Keymap

The global keymap (q.v.) contains key bindings that are in effect except when overridden by local key bindings in a major mode's local keymap (q.v.). See section Keymaps.

Global Mark Ring

The global mark ring records the series of buffers you have recently set a mark in. In many cases you can use this to backtrack through buffers you have been editing in, or in which you have found tags. See section The Global Mark Ring.

Global Substitution

Global substitution means replacing each occurrence of one string by another string through a large amount of text. See section Replacement Commands.

Global Variable

The global value of a variable (q.v.) takes effect in all buffers that do not have their own local (q.v.) values for the variable. See section Variables.

Graphic Character

Graphic characters are those assigned pictorial images rather than just names. All the non-Meta (q.v.) characters except for the Control (q.v.) characters are graphic characters. These include letters, digits, punctuation, and spaces; they do not include RET or ESC. In Emacs, typing a graphic character inserts that character (in ordinary editing modes). See section Basic Editing Commands.

Highlighting

Highlighting text means displaying it with a different foreground and/or background color to make it stand out from the rest of the text in the buffer.

Hardcopy

Hardcopy means printed output. Emacs has commands for making printed listings of text in Emacs buffers. See section Hardcopy Output.

HELP

You can type HELP at any time to ask what options you have, or to ask what any command does. The character HELP is really C-h. See section Help.

Hyper

Hyper is the name of a modifier bit which a keyboard input character may have. To make a character Hyper, type it while holding down the HYPER key. Such characters are given names that

start with Hyper- (usually written H- for short). See section Kinds of User Input.

Inbox

An inbox is a file in which mail is delivered by the operating system. Rmail transfers mail from inboxes to Rmail files (q.v.) in which the mail is then stored permanently or until explicitly deleted. See section Rmail Files and Inboxes.

Indentation

Indentation means blank space at the beginning of a line. Most programming languages have conventions for using indentation to illuminate the structure of the program, and Emacs has special commands to adjust indentation. See section Indentation.

Indirect Buffer

An indirect buffer is a buffer that shares the text of another buffer, called its base buffer. See section Indirect Buffers.

Input Event

An input event represents, within Emacs, one action taken by the user on the terminal. Input events include typing characters, typing function keys, pressing or releasing mouse buttons, and switching between Emacs frames. See section Kinds of User Input.

Insertion

Insertion means copying text into the buffer, either from the keyboard or from some other place in Emacs.

Justification

Justification means adding extra spaces to lines of text to make them come exactly to a specified width. See section Filling Text.

Keyboard Macro

Keyboard macros are a way of defining new Emacs commands from sequences of existing ones, with no need to write a Lisp program. See section Keyboard Macros.

Key Sequence

A key sequence (key, for short) is a sequence of input events (q.v.) that are meaningful as a single unit. If the key sequence is enough to specify one action, it is a complete key (q.v.); if it is not enough, it is a prefix key (q.v.). See section Keys.

Keymap

The keymap is the data structure that records the bindings (q.v.) of key sequences to the commands that they run. For example, the global keymap binds the character C-n to the command function `next-line`. See section Keymaps.

Keyboard Translation Table

The keyboard translation table is an array that translates the character codes that come from the terminal into the character codes that make up key sequences. See section Keyboard Translations.

Kill Ring

The kill ring is where all text you have killed recently is saved. You can reinsert any of the killed text still in the ring; this is called yanking (q.v.). See section Yanking.

Killing

Killing means erasing text and saving it on the kill ring so it can be yanked (q.v.) later. Some other systems call this "cutting". Most Emacs commands to erase text do killing, as opposed to deletion (q.v.). See section Deletion and Killing.

Killing Jobs

Killing a job (such as, an invocation of Emacs) means making it cease to exist. Any data within it, if not saved in a file, is lost. See section Exiting Emacs.

List

A list is, approximately, a text string beginning with an open parenthesis and ending with the matching close parenthesis. In C mode and other non-Lisp modes, groupings surrounded by other kinds of matched delimiters appropriate to the language, such as braces, are also considered lists. Emacs has special commands for many operations on lists. See section Lists and Sexps.

Local

Local means `in effect only in a particular context'; the relevant kind of context is a particular function execution, a particular buffer, or a particular major mode. It is the opposite of `global' (q.v.). Specific uses of `local' in Emacs terminology appear below.

Local Abbrev

A local abbrev definition is effective only if a particular major mode is selected. In that major mode, it overrides any global definition for the same abbrev. See section Abbrevs.

Local Keymap

A local keymap is used in a particular major mode; the key bindings (q.v.) in the current local keymap override global bindings of the same key sequences. See section Keymaps.

Local Variable

A local value of a variable (q.v.) applies to only one buffer. See section Local Variables.

M-

M- in the name of a character is an abbreviation for META, one of the modifier keys that can accompany any character. See section Kinds of User Input.

M-C-

M-C- in the name of a character is an abbreviation for Control-Meta; it means the same thing as C-M-. If your terminal lacks a real META key, you type a Control-Meta character by typing ESC and then typing the corresponding Control character. See section Kinds of User Input.

M-x

M-x is the key sequence which is used to call an Emacs command by name. This is how you run commands that are not bound to key sequences. See section Running Commands by Name.

Mail

Mail means messages sent from one user to another through the computer system, to be read at the recipient's convenience. Emacs has commands for composing and sending mail, and for reading and editing the mail you have received. See section Sending Mail. See section Reading Mail with Rmail, for how to read mail.

Major Mode

The Emacs major modes are a mutually exclusive set of options, each of which configures Emacs for editing a certain sort of text. Ideally, each programming language has its own major mode. See section Major Modes.

Mark

The mark points to a position in the text. It specifies one end of the region (q.v.), point being the other end. Many commands operate on all the text from point to the mark. Each buffer has its own mark. See section The Mark and the Region.

Mark Ring

The mark ring is used to hold several recent previous locations of the mark, just in case you want to move back to them. Each buffer has its own mark ring; in addition, there is a single global mark ring (q.v.). See section The Mark Ring.

Menu Bar

The menu bar is the line at the top of an Emacs frame. It contains words you can click on with the mouse to bring up menus. The menu bar feature is supported only with X. See section Menu Bars.

Message

See `mail'.

Meta

Meta is the name of a modifier bit which a command character may have. It is present in a character if the character is typed with the META key held down. Such characters are given names that start with Meta- (usually written M- for short). For example, M-< is typed by holding down META and at the same time typing < (which itself is done, on most terminals, by holding down SHIFT and typing ,). See section Kinds of User Input.

Meta Character

A Meta character is one whose character code includes the Meta bit.

Minibuffer

The minibuffer is the window that appears when necessary inside the echo area (q.v.), used for reading arguments to commands. See section The Minibuffer.

Minibuffer History

The minibuffer history records the text you have specified in the past for minibuffer arguments, so you can conveniently use the same text again. See section Minibuffer History.

Minor Mode

A minor mode is an optional feature of Emacs which can be switched on or off independently of all other features. Each minor mode has a command to turn it on or off. See section Minor Modes.

Minor Mode Keymap

A keymap that belongs to a minor mode and is active when that mode is enabled. Minor mode keymaps take precedence over the buffer's local keymap, just as the local keymap takes precedence over the global keymap. See section Keymaps.

Mode Line

The mode line is the line at the bottom of each window (q.v.), giving status information on the buffer displayed in that window. See section The Mode Line.

Modified Buffer

A buffer (q.v.) is modified if its text has been changed since the last time the buffer was saved (or since when it was created, if it has never been saved). See section Saving Files.

Moving Text

Moving text means erasing it from one place and inserting it in another. The usual way to move text by killing (q.v.) and then yanking (q.v.). See section Deletion and Killing.

Named Mark

A named mark is a register (q.v.) in its role of recording a location in text so that you can move point to that location. See section Registers.

Narrowing

Narrowing means creating a restriction (q.v.) that limits editing in the current buffer to only a part of the text in the buffer. Text outside that part is inaccessible to the user until the boundaries are widened again, but it is still there, and saving the file saves it all. See section Narrowing.

Newline

Linefeed characters in the buffer terminate lines of text and are therefore also called newlines. See section Character Set for Text.

Numeric Argument

A numeric argument is a number, specified before a command, to change the effect of the command. Often the numeric argument serves as a repeat count. See section Numeric Arguments.

Overwrite Mode

Overwrite mode is a minor mode. When it is enabled, ordinary text characters replace the existing text after point rather than pushing it to the right. See section Minor Modes.

Page

A page is a unit of text, delimited by formfeed characters (ASCII control-L, code 014) coming at the beginning of a line. Some Emacs commands are provided for moving over and operating on pages. See section Pages.

Paragraph

Paragraphs are the medium-size unit of English text. There are special Emacs commands for moving over and operating on paragraphs. See section Paragraphs.

Parsing

We say that certain Emacs commands parse words or expressions in the text being edited. Really, all they know how to do is find the other end of a word or expression. See section The Syntax Table.

Point

Point is the place in the buffer at which insertion and deletion occur. Point is considered to be between two characters, not at one character. The terminal's cursor (q.v.) indicates the location of

point. See section Basic Editing Commands.

Prefix Argument

See `numeric argument'.

Prefix Key

A prefix key is a key sequence (q.v.) whose sole function is to introduce a set of longer key sequences. C-x is an example of prefix key; any two-character sequence starting with C-x is therefore a legitimate key sequence. See section Keys.

Primary Rmail File

Your primary Rmail file is the file named `RMAIL' in your home directory. That's where Rmail stores your incoming mail, unless you specify a different file name. See section Reading Mail with Rmail.

Primary Selection

The primary selection is one particular X selection (q.v.); it is the selection that most X applications use for transferring text to and from other applications.

The Emacs kill commands set the primary selection and the yank command uses the primary selection when appropriate. See section Deletion and Killing.

Prompt

A prompt is text printed to ask the user for input. Displaying a prompt is called prompting. Emacs prompts always appear in the echo area (q.v.). One kind of prompting happens when the minibuffer is used to read an argument (see section The Minibuffer); the echoing which happens when you pause in the middle of typing a multi-character key sequence is also a kind of prompting (see section The Echo Area).

Quitting

Quitting means canceling a partially typed command or a running command, using C-g. See section Quitting and Aborting.

Quoting

Quoting means depriving a character of its usual special significance. In Emacs this is usually done with C-q. What constitutes special significance depends on the context and on convention. For example, an "ordinary" character as an Emacs command inserts itself; so in this context, a special character is any character that does not normally insert itself (such as DEL, for example), and quoting it makes it insert itself as if it were not special. Not all contexts allow quoting. See section Basic Editing Commands.

Read-Only Buffer

A read-only buffer is one whose text you are not allowed to change. Normally Emacs makes buffers read-only when they contain text which has a special significance to Emacs; for example, Dired buffers. Visiting a file that is write protected also makes a read-only buffer. See section Using Multiple Buffers.

Rectangle

A rectangle consists of the text in a given range of columns on a given range of lines. Normally

you specify a rectangle by putting point at one corner and putting the mark at the opposite cor
See section Rectangles.

Recursive Editing Level

A recursive editing level is a state in which part of the execution of a command involves asking the user to edit some text. This text may or may not be the same as the text to which the command was applied. The mode line indicates recursive editing levels with square brackets (`[' and `]'). See section Recursive Editing Levels.

Redisplay

Redisplay is the process of correcting the image on the screen to correspond to changes that have been made in the text being edited. See section The Organization of the Screen.

Regexp

See `regular expression'.

Region

The region is the text between point (q.v.) and the mark (q.v.). Many commands operate on the text of the region. See section The Mark and the Region.

Registers

Registers are named slots in which text or buffer positions or rectangles can be saved for later use. See section Registers.

Regular Expression

A regular expression is a pattern that can match various text strings; for example, `l[0-9]+' matches `l' followed by one or more digits. See section Syntax of Regular Expressions.

Repeat Count

See `numeric argument'.

Replacement

See `global substitution'.

Restriction

A buffer's restriction is the amount of text, at the beginning or the end of the buffer, that is temporarily inaccessible. Giving a buffer a nonzero amount of restriction is called narrowing (q.v.). See section Narrowing.

RET

RET is a character that in Emacs runs the command to insert a newline into the text. It is also used to terminate most arguments read in the minibuffer (q.v.). See section Kinds of User Input.

Rmail File

An Rmail file is a file containing text in a special format used by Rmail for storing mail. See section Reading Mail with Rmail.

Saving

Saving a buffer means copying its text into the file that was visited (q.v.) in that buffer. This is the way text in files actually gets changed by your Emacs editing. See section Saving Files.

Scroll Bar

A scroll bar is a tall thin hollow box that appears at the side of a window. You can use mouse commands in the scroll bar to scroll the window. The scroll bar feature is supported only with X. See section Scroll Bars.

Scrolling

Scrolling means shifting the text in the Emacs window so as to see a different part of the buffer. See section Controlling the Display.

Searching

Searching means moving point to the next occurrence of a specified string or the next match for a specified regular expression. See section Searching and Replacement.

Search Path

A search path is a list of directory names, to be used for searching for files for certain purposes. For example, the variable `load-path` holds a search path for finding Lisp library files. See section Libraries of Lisp Code for Emacs.

Secondary Selection

The secondary selection is one particular X selection; some X applications can use it for transferring text to and from other applications. Emacs has special mouse commands for transferring text using the secondary selection. See section Secondary Selection.

Selecting

Selecting a buffer means making it the current (q.v.) buffer. See section Using Multiple Buffers.

Selection

The X window system allows an application program to specify named selections whose values are text. A program can also read the selections that other programs have set up. This is the principal way of transferring text between window applications. Emacs has commands to work with the primary (q.v.) selection and the secondary (q.v.) selection.

Self-Documentation

Self-documentation is the feature of Emacs which can tell you what any command does, or give you a list of all commands related to a topic you specify. You ask for self-documentation with the help character, C-h. See section Help.

Sentences

Emacs has commands for moving by or killing by sentences. See section Sentences.

Sexp

A sexp (short for `s-expression') is the basic syntactic unit of Lisp in its textual form: either a list, or Lisp atom. Many Emacs commands operate on sexps. The term `sexp' is generalized to languages other than Lisp, to mean a syntactically recognizable expression. See section Lists and Sexps.

Simultaneous Editing

Simultaneous editing means two users modifying the same file at once. Simultaneous editing if not detected can cause one user to lose his work. Emacs detects all cases of simultaneous editing and

warns one of the users to investigate. See section Protection against Simultaneous Editing.

String

A string is a kind of Lisp data object which contains a sequence of characters. Many Emacs variables are intended to have strings as values. The Lisp syntax for a string consists of the characters in the string with a `"' before and another `"' after. A `"' that is part of the string must be written as `\"' and a `\' that is part of the string must be written as `\\'. All other characters, including newline, can be included just by writing them inside the string; however, backslash sequences as in C, such as `\n' for newline or `\241' using an octal character code, are allowed as well.

String Substitution

See `global substitution'.

Syntax Table

The syntax table tells Emacs which characters are part of a word, which characters balance each other like parentheses, etc. See section The Syntax Table.

Super

Super is the name of a modifier bit which a keyboard input character may have. To make a character Super, type it while holding down the SUPER key. Such characters are given names that start with Super- (usually written s- for short). See section Kinds of User Input.

Tags Table

A tags table is a file that serves as an index to the function definitions in one or more other files. See section Tags Tables.

Termscript File

A termscript file contains a record of all characters sent by Emacs to the terminal. It is used for tracking down bugs in Emacs redisplay. Emacs does not make a termscript file unless you tell it to. See section Reporting Bugs.

Text

Two meanings (see section Commands for Human Languages):

> Data consisting of a sequence of characters, as opposed to binary numbers, images, graphics commands, executable programs, and the like. The contents of an Emacs buffer are always text in this sense.
>
> Data consisting of written human language, as opposed to programs, or following the stylistic conventions of human language.

Top Level

Top level is the normal state of Emacs, in which you are editing the text of the file you have visited. You are at top level whenever you are not in a recursive editing level (q.v.) or the minibuffer (q.v.), and not in the middle of a command. You can get back to top level by aborting (q.v.) and quitting (q.v.). See section Quitting and Aborting.

Transposition

Transposing two units of text means putting each one into the place formerly occupied by the other. There are Emacs commands to transpose two adjacent characters, words, sexps (q.v.) or

lines (see section Transposing Text).

Truncation

Truncating text lines in the display means leaving out any text on a line that does not fit within the right margin of the window displaying it. See also `continuation line'. See section Basic Editing Commands.

Undoing

Undoing means making your previous editing go in reverse, bringing back the text that existed earlier in the editing session. See section Undoing Changes.

User Option

A user option is a variable (q.v.) that exists so that you can customize Emacs by setting it to a new value. See section Variables.

Variable

A variable is an object in Lisp that can store an arbitrary value. Emacs uses some variables for internal purposes, and has others (known as `user options' (q.v.)) just so that you can set their values to control the behavior of Emacs. The variables used in Emacs that you are likely to be interested in are listed in the Variables Index in this manual. See section Variables, for information on variables.

Version Control

Version control systems keep track of multiple versions of a source file. They provide a more powerful alternative to keeping backup files (q.v.). See section Version Control.

Visiting

Visiting a file means loading its contents into a buffer (q.v.) where they can be edited. See section Visiting Files.

Whitespace

Whitespace is any run of consecutive formatting characters (space, tab, newline, and backspace).

Widening

Widening is removing any restriction (q.v.) on the current buffer; it is the opposite of narrowing (q.v.). See section Narrowing.

Window

Emacs divides a frame (q.v.) into one or more windows, each of which can display the contents of one buffer (q.v.) at any time. See section The Organization of the Screen, for basic information on how Emacs uses the screen. See section Multiple Windows, for commands to control the use of windows.

Word Abbrev

Synonymous with `abbrev'.

Word Search

Word search is searching for a sequence of words, considering the punctuation between them as insignificant. See section Word Search.

WYSIWYG

WYSIWYG stands for "What you see is what you get." Emacs generally provides WYSIWYG editing for files of characters; in Enriched mode (see section Editing Formatted Text), it provides WYSIWYG editing for files that include text formatting information.

Yanking

Yanking means reinserting text previously killed. It can be used to undo a mistaken kill, or for copying or moving text. Some other systems call this "pasting". See section Yanking.

Go to the previous, next section.

Go to the previous, next section.

TOC

Key (Character) Index

!

"

#

$

%

*

+

.

/

<

=

>

@

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

- q (Rmail)

r

- R (Dired)
- r (Rmail)
- RET
- RET (Dired)
- RET (Occur mode)
- RET (Shell mode)
- RIGHT

s

- S (Calendar mode)
- s (Calendar mode)
- s (Dired)
- S (Dired)
- s (Gnus Summary mode)
- s (Rmail)
- S-Mouse-1
- SPC
- SPC (Calendar mode)
- SPC (Dired)
- SPC (Gnus)
- SPC (Rmail)

t

- t (Calendar mode)
- t (Rmail)
- TAB
- TAB (and major modes)
- TAB (completion)
- TAB (GUD)

u

v

w

x

z

~

Go to the previous, next section.

Go to the previous, next section.

Command and Function Index

2

a

b

- bookmark-save
- bookmark-set
- bookmark-write
- buffer-menu

c

- c-add-style
- c-backslash-region
- c-backward-conditional
- c-backward-into-nomenclature
- c-beginning-of-statement
- c-end-of-statement
- c-fill-paragraph
- c-forward-conditional
- c-forward-into-nomenclature
- c-indent-command
- c-indent-defun
- c-indent-exp
- c-indent-line
- c-macro-expand
- c-mark-function
- c-scope-operator
- c-set-offset
- c-set-style
- c-show-syntactic-information
- c-toggle-auto-hungry-state
- c-toggle-auto-state
- c-toggle-hungry-state
- c-up-conditional
- calendar
- calendar-backward-day
- calendar-backward-month
- calendar-backward-week

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

S

t

u

V

W

x

y

z

Go to the previous, next section.

Go to the previous, next section.

Variable Index

a

b

c

d

e

f

g

h

i

k

l

m

n

o

p

r

S

t

u

v

w

X

Go to the previous, next section.

Go to the previous section.

Concept Index

b

C

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

S

t

u

V

W

X

y

z

Go to the previous section.

GNU Emacs Manual

(1)

In some editors, search-and-replace operations are the only convenient way to make a single change in the text.

(2)

Some systems use Mouse-3 for a mode-specific menu. We took a survey of users, and found they preferred to keep Mouse-3 for selecting and killing regions. Hence the decision to use C-Mouse-3 for this menu.

(3)

You should not suspend the shell process. Suspending a subjob of the shell is a completely different matter--that is normal practice, but you must use the shell to continue the subjob; this command won't do it.

(4)

This dissociword actually appeared during the Vietnam War, when it was very appropriate.

(5)

The wording here was careless. The intention was that nobody would have to pay for *permission* to use the GNU system. But the words don't make this clear, and people often interpret them as saying that copies of GNU should always be distributed at little or no charge. That was never the intent; later on, the manifesto mentions the possibility of companies providing the service of distribution for a profit. Subsequently I have learned to distinguish carefully between "free" in the sense of freedom and "free" in the sense of price. Free software is software that users have the freedom to distribute and change. Some users may obtain copies at no charge, while others pay to obtain copies--and if the funds help support improving the software, so much the better. The important thing is that everyone who has a copy has the freedom to cooperate with others in using it.

(6)

This is another place I failed to distinguish carefully between the two different meanings of "free". The statement as it stands is not false--you can get copies of GNU software at no charge, from your friends or over the net. But it does suggest the wrong idea.

(7)

Several such companies now exist.

(8)

The Free Software Foundation raises most of its funds from a distribution service, although it is a charity rather than a company. If *no one* chooses to obtain copies by ordering from the FSF, it will be unable to do its work. But this does not mean that proprietary restrictions are justified to force every user to pay. If a small fraction of all the users order copies from the FSF, that is sufficient to keep the FSF afloat. So we ask users to choose to support us in this way. Have you done your part?

(9)

A group of computer companies recently pooled funds to support maintenance of the GNU C Compiler.

Expanded Plain TeX

March 1993

For version 2.3.

Karl Berry Steven Smith

Go to the next section.

@paragraphindent 2

Introduction

The Eplain macro package expands on and extends the definitions in plain TeX. This manual describes the definitions that you, as either an author or a macro writer, might like to use. It doesn't discuss the implementation; see comments in the source code for that.

Eplain is not intended to provide "generic typesetting capabilities, as do LaTeX (written by Leslie Lamport) or Texinfo (written by Richard Stallman and others). Instead, it provides definitions that are intended to be useful regardless of the high-level commands that you use when you actually prepare your manuscript.

For example, Eplain does not have a command `\section`, which would format section headings in an "appropriate" way, such as LaTeX's `\section`. The philosophy of Eplain is that some people will always need or want to go beyond the macro designer's idea of "appropriate". Such "canned" macros are fine--as long as you are willing to accept the resulting output. If you don't like the results, or if you are trying to match a different format, you are out of luck.

On the other hand, almost everyone would like capabilities such as cross-referencing by labels, so that you don't have to put actual page numbers in the manuscript. The author of Eplain is not aware of any generally available macro packages that (1) do not force their typographic style on an author, and yet (2) provide such capabilities.

Besides such generic macros as cross-referencing, Eplain contains another set of definitions: ones that change the conventions of plain TeX's output. For example, math displays in TeX are, by default, centered. If you want your displays to come out left-justified, you have to plow through The TeXbook to find some way to do it, and then adapt the code to your own needs. Eplain tries to take care of the messy details of such things, while still leaving the detailed appearance of the output up to you.

Finally, numerous definitions turned out to be useful as Eplain was developed. They are also documented

in this manual, on the chance that people writing other macros will be able to use them.

You can send bug reports or suggestions to `karl@cs.umb.edu`. The current version number of Eplain is defined as the macro `\fmtversion` at the end of the source file `` `eplain.tex' ``. When corresponding, please refer to it.

Go to the next section.

Go to the previous, next section.

Installation

The simplest way to install Eplain is simply to install the file `eplain.tex' in a directory where TeX will find it. What that directory is obviously depends on your operating system and TeX installation. I install `eplain.tex' in `/usr/local/lib/tex/macros/plain'.

If you want, you can also create a format (`.fmt') file for Eplain, which will eliminate the time spent reading the macro source file with \input. You do this by issuing a sequence of commands something like this:

```
initex
This is TeX, ...
**&plain eplain
(eplain.tex)
*\dump
... messages ...
```

You must make sure that `eplain.aux' exists *before* you run `initex'; otherwise, warning messages about undefined labels will never be issued.

You then have to install the resulting `eplain.fmt' in some system directory or set an environment variable to tell TeX how to find it. I install the format files in `/usr/local/lib/tex/formats'; the environment variable for the Web2C port of TeX to Unix is TEXFORMATS.

Some implementations of TeX (including Web2C) use the name by which TeX is invoked to determine what format to read. For them, you should make a link to the `virtex' program named `etex', and then install the format file with the name `etex.fmt'. This lets users invoke TeX as `etex' and get the format file read automatically.

For convenience, the file `etex.tex' in the distribution directory does \input eplain and then \dump, so that if you replace `eplain' with `etex' in the example above, the format file will end up with the right name.

The install target in the `Makefile' does all this properly for Unix systems and Web2C. You may have to change the pathnames.

Go to the previous, next section.

Go to the previous, next section.

Invoking Eplain

The simplest way to use Eplain is simply to put:

```
\input eplain
```

at the beginning of your input file. The macro file is small enough that reading it does not take an unbearably long time--at least on contemporary machines.

In addition, if a format (`.fmt') file has been created for Eplain (see the previous section), you can eliminate the time spent reading the macro source file. You do this by responding `&eplain` or `&etex` to TeX's `**' prompt. For example:

```
initex
This is TeX, ...
**&eplain myfile
```

Depending on the implementation of TeX which you are using, you might also be able to invoke TeX as `etex' and have the format file automatically read.

If you write something which you will be distributing to others, you won't know if the Eplain format will be loaded already. If it is, then doing `\input eplain` will waste time; if it isn't, then you must load it. To solve this, Eplain defines the control sequence `\eplain` to be the letter `t` (a convention borrowed from Lisp; it doesn't actually matter what the definition is, only that the definition exists). Therefore, you can do the following:

```
\ifx\eplain\undefined \input eplain \fi
```

where `\undefined` must never acquire a definition.

Eplain consists of several source files:

`xeplain.tex'
: most of the macros;

`arrow.tex'
: commutative diagram macros, see section Arrow theoretic diagrams (written by Steven Smith);

`btxmac.tex'
: bibliography-related macros, see section Citations;

`texnames.sty'
: abbreviations for various TeX-related names, see section Logos (edited by Nelson Beebe).

The file `eplain.tex' is all of these files merged together, with comments removed.

All of these files except `xeplain.tex' can be input individually, if all you want are the definit in that file.

Also, since the bibliography macros are fairly extensive, you might to not load them, while loading the rest of Eplain, to conserve memory in TeX. Therefore, you can control whether or not `eplain.tex' defines the bibliography-related stuff when you run TeX: if the control sequence \nobibtex is defined, then the definitions aren't made. You must set \nobibtex before `eplain.tex' is read, naturally. For example, you could start your input file like this:

```
\let\nobibtex = t
\input eplain
```

By default, \nobibtex is undefined, and so the bibliography definitions *are* made.

Sometimes you may want to be even more drastic than \nobibtex---not read or write an `aux' file at all, for any kind of cross-referencing. This happens if you define \noauxfile before reading `eplain.tex'. This also turns off all warnings about undefined labels.

Go to the previous, next section.

Go to the previous, next section.

User definitions

This chapter describes definitions that are meant to be used directly in a document. When appropriate, ways to change the default formatting are described in subsections.

Diagnostics

Plain TeX provides the `\tracingall` command, to turn on the maximum amount of tracing possible in TeX. The (usually voluminous) output from `\tracingall` goes both on the terminal and into the transcript file. It is sometimes easier to have the output go only to the transcript file, so you can peruse it at your leisure and not obscure other output to the terminal. So, Eplain provides the command `\loggingall`. (For some reason, this command is available in Metafont, but not in TeX.)

It is also sometimes useful to see the complete contents of boxes. `\tracingboxes` does this. (It doesn't affect whether or not the contents are shown on the terminal.)

You can turn off all tracing with `\tracingoff`.

You can also turn logging on and off globally, so you don't have to worry about whether or not you're inside a group at the time of command. These variants are named `\gloggingall` and `\gtracingall`.

Finally, if you write your own help messages (see `\newhelp` in The TeXbook), you want a convenient way to break lines in them. This is what TeX's `\newlinechar` parameter is for; however, plain TeX doesn't set `\newlinechar`. Therefore, Eplain defines it to be the character `^^J`.

For example, one of Eplain's own error messages is defined as follows:

```
\newhelp\envhelp{Perhaps you forgot to end the previous^^J%
   environment? I'm finishing off the current group,^^J%
   hoping that will fix it.}%
```

Rules

The default dimensions of rules are defined in chapter 21 of the The TeXbook. To sum up what is given there, the "thickness" of rules is 0.4pt by default. Eplain defines three parameters that let you change this dimension: `\hruledefaultheight`, `\hruledefaultdepth`, and `\vruledefaultwidth`. By default, they are defined as The TeXbook describes.

But it would be wrong to redefine `\hrule` and `\vrule`. For one thing, some macros in plain TeX depend on the default dimensions being used; for another, rules are used quite heavily, and the performance impact of making it a macro can be noticeable. Therefore, to take advantage of the default

rule parameters, you must use `\ehrule` and `\evrule`.

Citations

Bibliographies are part of almost every technical document. To handle them easily, you need two things: a program to do the tedious formatting, and a way to cite references by labels, rather than by numbers. The BibTeX program, written by Oren Patashnik, takes care of the first item; the citation commands in LaTeX, written to be used with BibTeX, take care of the second. Therefore, Eplain adopts the use of BibTeX, and virtually the same interface as LaTeX.

The general idea is that you put citation commands in the text of your document, and commands saying where the bibliography data is. When you run TeX, these commands produce output on the file with the same root name as your document (by default) and the extension `` `.aux' ``. BibTeX reads this file. You should put the bibliography data in a file or files with the extension `` `.bib' ``. BibTeX writes out a file with the same root name as your document and extension `` `.bbl' ``. Eplain reads this file the next time you run your document through TeX. (It takes multiple passes to get everything straight, because usually after seeing your bibliography typeset, you want to make changes in the `` `.bib' `` file, which means you have to run BibTeX again, which means you have to run TeX again...) An annotated example of the whole process is given below.

If your document has more than one bibliography--for example, if it is a collection of papers--you can tell Eplain to use a different root name for the `` `.bbl' `` file by defining the control sequence `\bblfilebasename`. The default definition is simply `\jobname`.

See the document BibTeXing (whose text is in the file `` `btxdoc.tex' ``, which should be in the Eplain distribution you got) for information on how to write your `.bib` files. Both the BibTeX and the Eplain distributions contain several examples, also.

The `\cite` command produces a citation in the text of your document. The exact printed form the citation will take is under your control; see section Formatting citations. `\cite` takes one required argument, a comma-separated list of cross-reference labels (see section Cross-references, for exactly what characters are allowed in such labels). Warning: spaces in this list are taken as part of the following label name, which is probably not what you expect. The `\cite` command also produces a command in the `.aux` file that tells BibTeX to retrieve the given reference(s) from the `.bib` file. `\cite` also takes one optional argument, which you specify within square brackets, as in LaTeX. This text is simply typeset after the citations. (See the example below.)

Another command, `\nocite`, puts the given reference(s) into the bibliography, but produces nothing in the text.

The `\bibliography` command is next. It serves two purposes: producing the typeset bibliography, and telling BibTeX the root names of the `.bib` files. Therefore, the argument to `\bibliography` is a comma separated list of the `.bib` files (without the `.bib'). Again, spaces in this list are significant.

You tell BibTeX the particular style in which you want your bibliography typeset with one more command: `\bibliographystyle`. The argument to this is a single filename style, which tells BibTeX to look for a file style`.bst`. See the document Designing BibTeX styles (whose text is in the

`btxhak.tex') for information on how to write your own styles.

Eplain automatically reads the citations from the `.aux` file when your job starts.

If you don't want to see the messages about undefined citations, you can say `\xrefwarningfalse` before making any citations. Eplain automatically does this if the `.aux` file does not exist. You can restore the default by saying `\xrefwarningtrue`.

Here is a TeX input file that illustrates the various commands.

```
\input eplain                    % Reads the .aux file.
Two citations to Knuthian works:
  \cite[note]{surreal,concrete-math}.
\beginsection{References.}\par   % Title for the bibliography.
\bibliography{knuth}             % Use knuth.bib for the labels.
\bibliographystyle{plain}        % Number the references.
\end                             % End of the document.
```

If we suppose that this file was named `citex.tex' and that the bibliography data is in `knuth.bib' (as the `\bibliography` command says), the following commands do what's required. (`$ ' represents the shell prompt.)

```
$ tex citex       (produces undefined citation messages)
$ bibtex citex    (read knuth.bib and citex.aux, write citex.bbl)
$ tex citex       (read citex.bbl, still have undefined citations)
$ tex citex       (one more time, to resolve the references)
```

The output looks something like (because we used the `plain` bibliography style):

> Two citations to Knuthian works: [2,1 note].
>
> **References**
>
> [1] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading, Massachusetts, 1989.
>
> [2] Donald E. Knuth. *Surreal Numbers*. Addison-Wesley, Reading, Massachusetts, 1974.

See the BibTeX documentation for information on how to write the bibliography databases, and the bibliography styles that are available. (If you want your references printed with names, as in [Knu74], instead of numbered, the bibliography style is `alpha`.)

Formatting citations

You may wish to change Eplain's formatting of citations; i.e., the result of your `\cite` commands. By default, the citation labels are printed one after another, separated by commas and enclosed in brackets, using the main text font. Some formats require other styles, such as superscripted labels. You can accomodate such formats by redefining the following macros.

`\printcitestart`

`\printcitefinish`

Eplain expands these macros at the begining and end of the list of citations for each `\cite` command. By default, they produce a `[' and `]', respectively.

`\printbetweencitations`

If a `\cite` command has multiple citations, as in `\cite{acp,texbook}`, Eplain expands this macro in between each pair of citations. By default, it produces a comma followed by a space.

`\printcitenote`

This macro takes one argument, which is the optional note to the `\cite` command. If the `\cite` command had no note, this macro isn't used. Otherwise, it should print the note. By default, the note is preceded with a comma and a space.

Here is an example, showing you could produce citations as superscripted labels, with the optional notes in parentheses.

```
\def\printcitestart{\unskip $^\bgroup}
\def\printbetweencitations{,}
\def\printcitefinish{\egroup$}
\def\printcitenote#1{\hbox{\sevenrm\space (#1)}}
```

Formatting bibliographies

You may wish to change Eplain's formatting of the bibliography, especially with respect to the fonts that are used. Therefore, Eplain provides the following control sequences:

`\biblabelwidth`

This control sequence represents a `\dimen` register, and its value is the width of the widest label in the bibliography. Although it is unlikely you will ever want to redefine it, you might want to use it if you redefine `\biblabelprint`, below.

`\biblabelprint`

This macro takes one argument, the label to print. By default, the label is put in a box of width `\biblabelwidth`, and is followed by an enspace. When you want to change the spacing around the labels, this is the right macro to redefine.

`\biblabelcontents`

This macro also takes one argument, the label to print. By default, the label is printed using the font `\bblrm` (below), and enclosed in brackets. When you want to change the appearance of the label, but not the spacing around it, this is the right macro to redefine.

`\bblrm`

The default font used for printing the bibliography.

`\bblem`

The font used for printing the titles and other "emphasized" material.

`\bblsc`

In some styles, authors' names are printed in a caps-and-small-caps font. In those cases, this font is used.

`\bblnewblock`

This is invoked between each of the parts of a bibliography entry. The default is to leave some extra space between the parts; you could redefine it to start each part on a new line (for example). A part is simply a main element of the entry; for example, the author is a part. (It was LaTeX that introduced the (misleading, as far as I am concerned) term `block' for this.)

`\biblabelextraspace`

Bibliography entries are typeset with a hanging indentation of `\biblabelwidth` plus this. The default is `.5em`, where the em width is taken from the `\bblrm` font. If you want to change this, you should do it inside `\bblhook`.

`\bblhook`

This is invoked before reading the `.bbl` file. By default, it does nothing. You could, for example, define it to set the bibliography fonts, or produce the heading for the references. Two spacing parameters must be changed inside `\bblhook`: `\parskip`, which produces extra space between the items; and `\biblabelextraspace`, which is described above.

If you are really desperate, you can also hand-edit the `.bbl` file that BibTeX produces to do anything you wish.

Displays

By default, TeX centers displayed material. (Displayed material is just whatever you put between $$'s--it's not necessarily mathematics.) Many layouts would be better served if the displayed material was left-justified. Therefore, Eplain provides the command `\leftdisplays`, which indents displayed material by `\parindent` plus `\leftskip`, plus `\leftdisplayindent`.

You can go back to centering displays with `\centereddisplays`. (It is usually poor typography to have both centered and left-justified displays in a single publication, though.)

`\leftdisplays` also changes the plain TeX commands that deal with alignments inside math displays, `\displaylines`, `\eqalignno`, and `\leqalignno`, to produce left-justified text. You can still override this formatting by inserting `\hfill` glue, as explained in The TeXbook.

TOC

Formatting displays

If you want some other kind of formatting, you can write a definition of your own, analogous to `\leftdisplays`. You need only make sure that `\leftdisplaysetup` is called at the beginning of every display (presumably by invoking it in TeX's `\everydisplay` parameter), and to define `\generaldisplay`.

`\leftdisplays` expands the old value of `\everydisplay` before calling `\leftdisplaysetup`, so that any changes you have made to it won't be lost. That old token list as available as the value of the token register `\previouseverydisplay`.

Time of day

TeX provides the day, month, and year as numeric quantities (unless your TeX implementation is woefully deficient). Eplain provides some control sequences to make them a little more friendly to humans.

`\monthname` produces the name of the current month, abbreviated to three letters.

`\fullmonthname` produces the name of the current month, unabbreviated (in English).

`\timestring` produces the current time, as in `1:14 p.m.'

`\timestamp` produces the current date and time, as in `23 Apr 64 1:14 p.m.'. (Except the spacing is slightly different.)

`\today` produces the current date, as in `23 April 1964'.

Lists

Many documents require lists of items, either numbered or simply enumerated. Plain TeX defines one macro to help with creating lists, `\item`, but that is insufficient in many cases. Therefore, Eplain provides two pairs of commands:

`\numberedlist ... \endnumberedlist`

`\orderedlist ... \endorderedlist`

> These commands (they are synonyms) produce a list with the items numbered sequentially, starting from one. A nested `\numberedlist` labels the items with lowercase letters, starting with `a'. Another nested `\numberedlist` labels the items with roman numerals. Yet more deeply nested numbered lists label items with `*'.

`\unorderedlist ... \endunorderedlist`

> This produces a list with the items labelled with small black boxes ("square bullets"). A nested `\unorderedlist` labels items with em-dashes. Doubly (and deeper) nested unordered lists label items with `*'s.

The two kinds of lists can be nested within each other, as well.

In both kinds of lists, you begin an item with `\li`. An item may continue for several paragraphs. Ea(item starts a paragraph.

You can give `\li` an optional argument, a cross-reference label. It's defined to be the "marker" for the current item. This is useful if the list items are numbered. You can produce the value of the label with `\xrefn`. See section Cross-references.

You can also say `\listcompact` right after `\numberedlist` or `\unorderedlist`. The items in the list will then not have any extra space between them (see section Formatting lists). You might want to do this if the items in this particular list are short.

Here is an example:

```
\numberedlist\listcompact
\li The first item.
\li The second item.

The second paragraph of the second item.
\endnumberedlist
```

Formatting lists

Several registers define the spacing associated with lists. It is likely that their default values won't suit your particular layout.

`\abovelistskipamount, \belowlistskipamount`

> The vertical glue inserted before and after every list, respectively.

`\interitemskipamount`

> The vertical glue inserted before each item except the first. `\listcompact` resets this to zero, as mentioned above.

`\listleftindent, \listrightindent`

> `\listrightindent` is the amount of space by which the list is indented on the right; i.e., it is added to `\rightskip`. `\listleftindent` is the amount of space, *relative to* `\parindent`, by which the list is indented on the left. Why treat the two parameters differently? Because (a) it is more useful to make the list indentation depend on the paragraph indentation; (b) footnotes aren't formatted right if `\parindent` is reset to zero.

The three vertical glues are inserted by macros, and preceded by penalties: `\abovelistskip` does `\vpenalty\abovelistpenalty` and then `\vskip\abovelistskip`. `\belowlistskip` and `\interitemskip` are analogous.

In addition, the macro `\listmarkerspace` is called to separate the item label from the item text. This is set to `\enspace` by default.

If you want to change the labels on the items, you can redefine these macros: `\numberedmarker` or `\unorderedmarker`. The following registers might be useful if you do:

`\numberedlistdepth, \unorderedlistdepth`

These keep track of the depth of nesting of the two kinds of lists.

`\itemnumber, \itemletter`

These keep track of the number of items that have been seen in the current numbered list. They are both integer registers. The difference is that `\itemnumber` starts at one, and `\itemletter` starts at 97, i.e., lowercase `a'.

You can also redefine the control sequences that are used internally, if you want to do something radically different: `\beginlist` is invoked to begin both kinds of lists; `\printitem` is invoked to print the label (and space following the label) for each item; and `\endlist` is invoked to end both kinds of lists.

Verbatim listing

It is sometimes useful to include a file verbatim in your document; for example, part of a computer program. The `\listing` command is given one argument, a filename, and produces the contents of that file in your document. `\listing` expands `\listingfont` to set the current font. The default value of `listingfont` is `\tt`.

You can take arbitrary actions before reading the file by defining `\setuplistinghook`. This is expanded just before the file is input.

If you want to have line numbers on the output, you can say `\let\setuplistinghook = \linenumberedlisting`. The line numbers are stored in the count register `\lineno` while the file is being read. You can redefine the macro `\printlistinglineno` to change how they are printed.

You can produce verbatim text in your document with `\verbatim`. End the text with `|endverbatim`. If you need a `|' in the text, double it. For example:

```
\verbatim ||\#%&!|endverbatim
```

produces `|\#%&!`.

Line breaks and spaces in the verbatim text are preserved.

Because `\verbatim` must change the category code of special characters, calling inside a macro definition of your own does not work properly. For example:

```
\def\mymacro{\verbatim &#%|endverbatim}% Doesn't work!
```

If you need to do this, you must change the category codes yourself before making the macro definition. Perhaps the `\uncatcodespecials` will help you (see section Category codes).

Contents

Producing a table of contents that is both useful and aesthetic is one of the most difficult design problems in any work. Naturally, Eplain does not pretend to solve the design problem. Collecting the raw data for a table of contents, however, is much the same across documents. Eplain uses an auxiliary file with extension `.toc` (and the same root name as your document) to save the information.

To write an entry for the table of contents, you say `\writetocentry{part}{text}`, where part is the type of part this entry is, e.g., `chapter', and text is the text of the title. `\writetocentry` puts an entry into the `.toc` file that looks like `\tocpartentry{text}{page number}`. The text is written unexpanded.

A related command, `\writenumberedtocentry`, takes one additional argument, and writes it unchanged. The first token is expanded at the point of the `\writenumberedtocentry`, but the rest of the argument is not expanded. The usual application is when the parts of the document are numbered. On the other hand, the one-level expansion allows you to use the argument for other things as well (author's names in a proceedings, say), and not have accents or other control sequences expanded. The downside is that if you *want* full expansion of the third argument, you don't get it--you must expand it yourself, before you call `\writenumberedtocentry`.

Here is a concrete example:

```
\writenumberedtocentry{chapter}{A $\sin$ wave}{\the\chapno}
\writetocentry{section}{A section title}
```

Supposing `\the\chapno` expanded to `3' and that the `\write`'s occurred on pages eight and nine, respectively, those lines would write the following to the `.toc` file:

```
\tocchapterentry{A $\sin$ wave}{3}{8}
\tocsectionentry{A section title}{9}
```

You read the `.toc` file with the command `\readtocfile`. Naturally, whatever `\toc ... entry` commands that were written to the file must be defined when `\readtocfile` is invoked. Eplain has simple definitions for `\tocchapterentry`, `\tocsectionentry`, and `\tocsubsectionentry` (unnumbered), but they aren't suitable for anything but preliminary proofs.

After reading the `.toc` file, `\readtocfile` opens the file for writing, thereby deleting the information from the previous run. You should therefore arrange that `\readtocfile` be called *before* the first call to a `\writetoc ...` macro. On the other hand, if you don't want to rewrite the `.toc` file, perhaps because you are only running TeX on part of your manuscript, you can set `\rewritetocfilefalse`.

By default, the `` `.toc' `` file has the root `\jobname`. If your document has more than one contents--for example, if it is a collection of papers--you can tell Eplain to use a different root name by defining the control sequence `\tocfilebasename`.

Cross-references

It is often useful to refer the reader to other parts of your document; but putting literal page, section, equation, or whatever numbers in the text is certainly a bad thing.

Eplain therefore provides commands for symbolic cross-references. It uses an auxiliary file with extension `.aux` (and the same root name as your document) to keep track of the information. Therefore, it takes two passes to get the cross-references right--one to write them out, and one to read them in. Eplain automatically reads the `.aux` file at the first reference; after reading it, Eplain reopens it for writing.

You can control whether or not Eplain warns you about undefined labels. See section Citations.

Labels in Eplain's cross-reference commands can use characters of category code eleven (letter), twelve (other), ten (space), three (math shift), four (alignment tab), seven (superscript), or eight (subscript). For example, `(a1 $&^_' is a valid label (assuming the category codes of plain TeX), but `%#\{' has no valid characters.

You can also do symbolic cross-references for bibliographic citations and list items. See section Citations, and section Lists.

Defining generic references

Eplain provides the command `\definexref` for general cross-references. It takes three arguments: the name of the label (see section above for valid label names), the value of the label (which can be anything), and the "class" of the reference--whether it's a section, or theorem, or what. For example:

```
\definexref{sec-intro}{3.1}{section}
```

Of course, the label value is usually generated by another macro using TeX count registers or some such.

`\definexref` doesn't actually define label; instead, it writes out the definition to the `.aux` file, where Eplain will read it on the next TeX run.

The class argument is used by the `\ref` and `\refs` commands. See the next section.

Using generic references

To retrieve the value of the label defined via `\definexref` (see the previous section), Eplain provides the following macros:

`\refn{label}`

`\xrefn{label}`

> `\refn` and `\xrefn` (they are synonyms) produce the bare definition of label. If label isn't defined, issue a warning, and produce label itself instead, in typewriter. (The warning isn't given if `\xrefwarningfalse`.)

`\ref{label}`

Given the class c for label (see the description of `\definexref` in the previous section), expand the control sequence `\c word` (if it's defined) followed by a tie. Then call `\refn` on label. (Example below.)

`\refs{label}`

Like `\ref`, but append the letter `s' to the `\...word`.

The purpose of the `\...word` macro is to produce the word `Section' or `Figure' or whatever that usually precedes the actual reference number.

Here is an example:

```
\def\sectionword{Section}
\definexref{sec-intro}{3.1}{section}
\definexref{sec-next}{3.2}{section}
See \refs{sec-intro} and \refn{sec-next} ...
```

This produces `See Sections 3.1 and 3.2 ...'

Page references

Eplain provides two commands for handling references to page numbers, one for definition and one for use.

`\xrdef{label}`

Define label to be the current page number. This produces no printed output, and ignores following spaces.

`\xref{label}`

Produce the text `p. page-number', which is the usual form for cross-references. The page-number is actually label's definition; if label isn't defined, the text of the label itself is printed.

Equation references

Instead of referring to pages, it's most useful if equation labels refer to equation numbers. Therefore, Eplain reserves a `\count` register, `\eqnumber`, for the current equation number, and increments it at each numbered equation.

Here are the commands to define equation labels and then refer to them:

`\eqdef{label}`

This defines label to be the current value of `\eqnumber`, and, if the current context is not inner, then produces a `\eqno` command. (The condition makes it possible to use `\eqdef` in an `\eqalignno` construction, for example.) The text of the equation number is produced using `\eqprint`. See section Formatting equation references.

`\eqdefn{label}`

This is like `\eqdef`, except it always omits the `\eqno` command. It can therefore be used in

places where `\eqdef` can't; for example, in a non-displayed equation. The text of the equation number is not produced, so you can also use it in the (admittedly unusual) circumstance where you want to define an equation label but not print that label.

`\eqref{label}`

This produces a formatted reference to label. If label is undefined (perhaps because it is a forward reference), it just produces the text of the label itself. Otherwise, it calls `\eqprint`.

`\eqrefn{label}`

This produces the cross-reference text for label. That is, it is like `\eqref`, except it doesn't call `\eqprint`.

Equation labels can contain the same characters that are valid in general cross-references.

Formatting equation references

Both defining an equation label and referring to it should usually produce output. This output is produced with the `\eqprint` macro, which takes one argument, the equation number being defined or referred to. By default, this just produces `(number)', where number is the equation number. To produce the equation number in a different font, or with different surrounding symbols, or whatever, you can redefine `\eqprint`. For example, the following definition would print all equation numbers in italics. (The extra braces define a group, to keep the font change from affecting surrounding text.)

```
\def\eqprint#1{{\it (#1)}}
```

In addition to changing the formatting of equation numbers, you might to add more structure to the equation number; for example, you might want to include the chapter number, to get equation numbers like `(1.2)'. To achieve this, you redefine `\eqconstruct`. For example:

```
\def\eqconstruct#1{\the\chapternumber.#1}
```

(If you are keeping the chapter number in a count register named `\chapternumber`, naturally.)

The reason why both `\eqconstruct` and `\eqprint` may not be immediately apparent, since they appear to perform similar functions. The difference is that `\eqconstruct` affects the text that cross-reference label is defined to be, while `\eqprint` affects only what is typeset on the page. The example just below might help.

Usually, you want equation labels to refer to equation numbers. But sometimes you might want a more complicated text. For example, you might have an equation `(1)', and then have a variation several pages later which you want to refer to as `(1*)'.

Therefore, Eplain allows you to give an optional argument (i.e., arbitrary text in square brackets) before the cross-reference label to \eqdef. Then, when you refer to the equation, that text is produced. Here's how to get the example just mentioned:

```
$$...\eqdef{a-eq}$$
...
```


```
$$...\eqdef[\eqrefn{a-eq}*]{a-eq-var}$$
In \eqref{a-eq-var}, we expand on \eqref{a-eq}, ...
```

We use `\eqrefn` in the cross-reference text, not `\eqref`, so that `\eqprint` is called only once.

Subequation references

Eplain also provides for one level of substructure for equations. That is, you might want to define a related group of equations with numbers like `2.1' and `2.2', and then be able to refer to the group as a whole: "... in the system of equations (2)...".

The commands to do this are `\eqsubdef` and `\eqsubdefn`. They take one label argument like their counterparts above, and generally behave in the same way. The difference is in how they construct the equation number: instead of using just `\eqnumber`, they also use another counter, `\subeqnumber`. This counter is advanced by one at every `\eqsubdef` or `\eqsubdefn`, and reset to zero at every `\eqdef` or `\eqdefn`.

You use `\eqref` to refer to subequations as well as main equations.

To put the two together to construct the text that the label will produce, they use a macro `\eqsubreftext`. This macros takes two arguments, the "main" equation number (which, because the equation label can be defined as arbitrary text, as described in the previous section, might be anything at all) and the "sub" equation number (which is always just a number). Eplain's default definition just puts a period between them:

```
  \def\eqsubreftext#1#2{#1.#2}%
```

You can redefine `\eqsubreftext` to print however you like. For example, this definition makes the labels print as `2a', `2b', and so on.

```
\newcount\subref
\def\eqsubreftext#1#2{%
  \subref = #2           % The space stops a <number>.
  \advance\subref by 96  % `a' is character code 97.
  #1\char\subref
}
```

Sadly, we must define a new count register, `\subref`, instead of using the scratch count register `\count255`, because `#1' might include other macro calls which use `\count255`.

Justification

Eplain defines three commands to conveniently justify multiple lines of text: `\flushright`, `\flushleft`, and `\center`.

They all work in the same way; let's take `\center` as the example. To start centering lines, you say `\center` inside a group; to stop, you end the group. Between the two commands, each end-of-line in

the input file also starts a new line in the output file.

The entire block of text is broken into paragraphs at blank lines, so all the TeX paragraph-shaping parameters apply in the usual way. This is convenient, but it implies something else that isn't so convenient: changes to any linespacing parameters, such as `\baselineskip`, will have *no effect* on the paragraph in which they are changed. TeX does not handle linespacing changes within a paragraph (because it doesn't know where the line breaks are until the end of the paragraph).

The space between paragraphs is by default one blank line's worth. You can adjust this space by assigning to `\blanklineskipamount`; this (vertical) glue is inserted after each blank line.

Here is an example:

```
{\center First line.

   Second line, with a blank line before.
}
```

This produces:

First line. Second line, with a blank line before.

You may wish to use the justification macros inside of your own macros. Just be sure to put them in a group. For example, here is how a title macro might be defined:

```
\def\title{\begingroup\titlefont\center}
\def\endtitle{\endgroup}
```

Tables

Eplain provides a single command, `\makecolumns`, to make generating one particular kind of table easier. More ambitious macro packages might be helpful to you for more difficult applications. The files `` `ruled.tex' `` and `` `TXSruled.tex' ``, available from `lifshitz.ph.utexas.edu' in `` `texis/tables' ``, is the only one I know of.

Many tables are homogenous, i.e., all the entries are semantically the same. The arrangement into columns is to save space on the page, not to encode different meanings. In this kind of the table, it is useful to have the column breaks chosen automatically, so that you can add or delete entries without worrying about the column breaks.

`\makecolumns` takes two arguments: the number of entries in the table, and the number of columns to break them into. As you can see from the example below, the first argument is delimited by a slash, and the second by a colon and a space (or end-of-line). The entries for the table then follow, one per line (not including the line with the `\makecolumns` command itself).

`\parindent` defines the space to the left of the table. `\hsize` defines the width of the table. So you can adjust the position of the table on the page by assignments to these parameters, probably inside a group.

You can also control the penalty at a page break before the `\makecolumns` by setting the parameter `\abovecolumnspenalty`. Usually, the table is preceded by some explanatory text. You wouldn't want a page break to occur after the text and before the table, so Eplain sets it to `10000`. But if the table produced by `\makecolumns` is standing on its own, `\abovecolumnspenalty` should be decreased.

If you happen to give `\makecolumns` a smaller number of entries than you really have, some text beyond the (intended) end of the table will be incorporated into the table, probably producing an error message, or at least some strange looking entries. And if you give `\makecolumns` a larger number of entries than you really have, some of the entries will be typeset as straight text, probably also looking somewhat out of place.

Here is an example:

```
% Arrange 6 entries into 2 columns:
\makecolumns 6/2: % This line doesn't have an entry.
one
two
three
four
five
six
Text after the table.
```

This produces `one', `two', and `three' in the first column, and `four', `five', and `six' in the second.

Margins

TeX's primitives describe the type area in terms of an offset from the upper left corner, and the width and height of the type. Some people prefer to think in terms of the margins at the top, bottom, left, and right of the page, and most composition systems other than TeX conceive of the page laid out in this way. Therefore, Eplain provides commands to directly assign and increment the margins.

`\topmargin = dimen`

`\bottommargin = dimen`

`\leftmargin = dimen`

`\rightmargin = dimen`

> These commands set the specified margin to the dimen given. The = and the spaces around it are optional. The control sequences here are not TeX registers, despite appearances; therefore, commands like `\showthe\topmargin` will not do what you expect.

`\advancetopmargin by dimen`

`\advancebottommargin by dimen`

`\advanceleftmargin by dimen`

`\advancerightmargin by dimen`

These commands change the specified margin by the dimen given.

Regardless of whether you use the assignment or the advance commands, Eplain always changes the type area in response, not the other margins. For example, when TeX starts, the left and right margins are both one inch. If you then say `\leftmargin = 2in`, the right margin will remain at one inch, and the width of the lines (i.e., `\hsize`) will decrease by one inch.

When you use any of these commands, Eplain computes the old value of the particular margin, by how much you want to change it, and then resets the values of TeX's primitive parameters to correspond. Unfortunately, Eplain cannot compute the right or bottom margin without help: you must tell it the full width and height of the final output page. It defines two new parameters for this:

`\paperheight`

The height of the output page; default is 11in.

`\paperwidth`

The width of the output page; default is 8.5in.

If your output page has different dimensions than this, you must reassign to these parameters, as in

```
\paperheight = 11in
\paperwidth = 17in
```

Double columns

Eplain provides for double column output: you just say `\doublecolumns`, and from that point on, the manuscript will be set in two columns. If you want to go back to one column, say `\singlecolumn`.

`\doublecolumns` inserts the value of the glue parameter `\abovedoublecolumnskip` before the double column text, and the value of the glue parameter `\belowdoublecolumnskip` after it. The default value for both of these parameters is `\bigskipamount`, i.e., one linespace.

The two columns are separated by the value of the dimen parameter `\gutter`. The column widths are therefore (`\hsize` - `\gutter`)/2. The default value for `\gutter` is two picas.

`\doublecolumns` takes into account only the insertion classes defined by plain TeX, namely, footnotes and `\topinsert`s. If you have defined additional insertion classes, you will need to change the macro `\@doublecolumnsplit`. Furthermore, the insertions do not respect the column widths; if you want them to, you have to change the way your output routine works. (Eplain uses whatever the current output routine is; it is not tied to `\plainoutput`.)

Footnotes

The most common reference mark for footnotes is a raised number, incremented on each footnote. The `\numberedfootnote` macro provides this. It takes one argument, the footnote text.

If your document uses only numbered footnotes, you could make typing `\numberedfootnote` more

convenient with a command such as:

```
\let\footnote = \numberedfootnote
```

After doing this, you can type your footnotes as `\footnote{footnote text}`, instead of as `\numberedfootnote{footnote text}`.

Eplain keeps the current footnote number in the count register `\footnotenumber`. So, to reset the footnote number to zero, as you might want to do at, for example, the beginning of a chapter, you could say `\footnotenumber=0`.

Plain TeX separates the footnote marker from the footnote text by an en space (it uses the `\textindent` macro). In Eplain, you can change this space by setting the dimension register `\footnotemarkseparation`. The default is still an en.

You can produce a space between footenotes by setting the glue register `\interfootnoteskip`. The default is zero.

You can also control footnote formatting in a more general way: Eplain expands the token register `\everyfootnote` before a footnote is typeset, but after the default values for all the parameters have been established. For example, if you want your footnotes to be printed in seven-point type, indented by one inch, you could say:

```
\everyfootnote = {\sevenrm \leftskip = 1in}
```

By default, an `\hrule` is typeset above each group of footnotes on a page. You can control the dimensions of this rule by setting the dimension registers `\footnoterulewidth` and `\footnoteruleheight`. The space between the rule and the first footnote on the page is determined by the dimension register `\belowfootnoterulespace`. If you don't want any rule at all, set `\footenoteruleheight=0pt`, and, most likely, `\belowfootnoterulespace=0pt`. The defaults for these parameters typeset the rule in the same way as plain TeX: the rule is 0.4 points high, 2 true inches wide, with 2.6 points below it.

The space above the rule and below the text on the page is controlled by the glue register `\skip\footins`. The default is a Plain `\bigskip`.

Fractions

Exercise 11.6 of The TeXbook describes a macro `\frac` for setting fractions, but `\frac` never made it into plain TeX. So Eplain includes it.

`\frac` typesets the numerator and denominator in `\scriptfont0`, slightly raised and lowered. The numerator and denominator are separated by a slash. The denominator must be enclosed in braces if it's more than one token long, but the numerator need not be. (This is a consequence of `\frac` taking delimited arguments; see page 203 of The TeXbook for an explanation of delimited macro arguments.)

For example, `\frac 23/{64}` turns `23/64' into .

Paths

When you typeset long pathnames, electronic mail addresses, or other such "computer" names, you would like TeX to break lines at punctuation characters within the name, rather than trying to find hyphenation points within the words. For example, it would be better to break the email address `letters@alpha.gnu.ai.mit.edu` at the `@' or a `.', rather than at the hyphenation points in `letters' and `alpha'.

If you use the `\path` macro to typeset the names, TeX will find these good breakpoints. The argument to `\path` is delimited by any other other than `\' which does not appear in the name itself. `|' is often a good choice, as in:

```
\path|letters@alpha.gnu.ai.mit.edu|
```

You can control the exact set of characters at which breakpoints will be allowed by calling `\discretionaries`. This takes the same sort of delimited argument; any character in the argument will henceforth be a valid breakpoint within `\path`. The default set is essentially all the punctuation characters:

```
\discretionaries |~!@$%^&*()_+`-=#{}[]:";'<>,.?\/|
```

If for some reason you absolutely must use `\` as the delimiter character for `\path`, you can set `\specialpathdelimiterstrue`. (Other delimiter characters can still be used.) TeX then processes the `\path` argument about four times more slowly.

Logos

Eplain redefines the `\TeX` macro of plain TeX to end with `\null`, so that the proper spacing is produced when `\TeX` is used at the end of a sentence. The other ...TeX macros listed here do this, also.

Eplain defines `\AMSTeX`, `\BibTeX` `\AMSLaTeX`, `\LAMSTeX`, `\LaTeX` `\MF`, and `\SLiTeX` to produce their respective logos. (Sorry, the logos are not shown here.) Some spelling variants of these are also supported.

Boxes

The solid rectangle that Eplain uses as a marker in unordered lists (see section Lists) is available by itself: just say `\blackbox`.

You can create black boxes of arbitrary size with `\hrule` or `\vrule`.

You can also get unfilled rectangles with `\makeblankbox`. This takes two explicit arguments: the height and depth of the rules that define the top and bottom of the rectangle. (The two arguments are added to get the width of the left and right borders, so that the thickness of the border is the same on all four sides.) It also uses, as implicit arguments, the dimensions of `\box0` to define the dimensions of the

rectangle it produces. (The contents of `\box0` are ignored.)

Here is an example. This small raised open box is suitable for putting next to numbers in, e.g., a table of contents.

```
\def\openbox{%
  \ht0 = 1.75pt \dp0 = 1.75pt \wd0 = 3.5pt
  \raise 2.75pt \makeblankbox{.2pt}{.2pt}
}
```

Finally, you can put a box around arbitrary text with `\boxit`. This takes one argument, which must itself be a (TeX) box, and puts a printed box around it, separated by `\boxitspace` white space (3 points by default) on all four sides. For example:

```
\boxit{\hbox{This text is boxed.}}
```

The reason that the argument must be a box is that when the text is more than one line long, TeX cannot figure out the line length for itself. Eplain does set `\parindent` to zero inside `\boxit`, since it is very unlikely you would want indentation there. (If you do, you can always reset it yourself.)

`\boxit` uses `\ehrule` and `\evrule` so that you can easily adjust the thicknesses of the box rules. See section Rules.

@catcode`@$=3 @catcode`@%=14 @catcode`@&=4 @catcode`@#=6 @catcode`@^=7 @catcode`@_=8 @catcode`@"=@other @catcode`@<=@other @catcode`@>=@other @catcode`@\=0 \catcode`\@=\other \input arrow \catcode`\@=0 @catcode`@\=@active @catcode`@$=@other @catcode`@%=@other @catcode`@&=@other @catcode`@#=@other @catcode`@^=@active @catcode`@_=@active @catcode`@"=@active @catcode`@<=@active @catcode`@>=@active

Go to the previous, next section.

Go to the previous, next section.

TOC

Arrow theoretic diagrams

This chapter describes definitions for producing commutative diagrams.

Steven Smith wrote this documentation (and the macros).

Slanted lines and vectors

The macros `\drawline` and `\drawvector` provide the capability found in LaTeX's picture mode to draw slanted lines and vectors of certain directions. Both of these macros take three arguments: two integer arguments to specify the direction of the line or vector, and one argument to specify its length. For example, `\drawvector(-4,1){60pt}' produces the vector which lies in the 2d quadrant, has a slope of minus 1/4, and a width of 60 pt.

Note that if an `\hbox` is placed around `\drawline` or `\drawvector`, then the width of the `\hbox` will be the positive dimension specified in the third argument, except when a vertical line or vector is specified, e.g., `\drawline(0,1){1in}`, which has zero width. If the specified direction lies in the 1st or 2d quadrant (e.g., `(1,1)` or `(-2,3)`), then the `\hbox` will have positive height and zero depth. Conversely, if the specified direction lies in the 3d or 4th quadrant (e.g., `(-1,-1)` or `(2,-3)`), then the `\hbox` will have positive depth and zero height.

There are a finite number of directions that can be specified. For `\drawline`, the absolute value of each integer defining the direction must be less than or equal to six, i.e., `(7,-1)` is incorrect, but `(6,-1)` is acceptable. For `\drawvector`, the absolute value of each integer must be less than or equal to four. Furthermore, the two integers cannot have common divisors; therefore, if a line with slope 2 is desired, say `(2,1)` instead of `(4,2)`. Also, specify `(1,0)` instead of, say, `(3,0)` for horizontal lines and likewise for vertical lines.

Finally, these macros depend upon the LaTeX font `line10`. If your site doesn't have this font, ask your system administrator to get it. Future enhancements will include macros to draw dotted lines and dotted vectors of various directions.

Commutative diagrams

The primitive commands `\drawline` and `\drawvector` can be used to typeset arrow theoretic diagrams. This section describes (1) macros to facilitate typesetting arrows and morphisms, and (2) macros to facilitate the construction of commutative diagrams. All macros described in this section must be used in math mode.

Arrows and morphisms

The macros `\mapright` and `\mapleft` produce right and left pointing arrows, respectively. Use superscript (^) to place a morphism above the arrow, e.g., `\mapright^\alpha'; use subscript (_) to place a morphism below the arrow, e.g., `\mapright_{\tilde l}'. Superscripts and subscripts may be used simulataneously, e.g., `\mapright^\pi_{\rm epimor.}'.

Similarly, the macros `\mapup` and `\mapdown` produce up and down pointing arrows, respectively. Use `\rt` to place a morphism to the right of the arrow, e.g., `\mapup\rt{\rm id}'; use `\lft` to place a morphism to the left of the arrow, e.g., `\mapup\lft\omega'. `\lft` and `\rt` may be used simultaneously, e.g., `\mapdown\lft\pi\rt{\rm monomor.}'.

Slanted arrows are produced by the macro `\arrow`, which takes a direction argument (e.g., `\arrow(3,-4)'). Use `\rt` and `\lft` to place morphisms to the right and left, respectively, of the arrow. A slanted line (no arrowhead) is produced with the macro `\sline`, whose syntax is identical to that of `\arrow`.

The length of these macros is predefined by the default TeX dimensions `\harrowlength`, for horizontal arrows (or lines), `\varrowlength`, for vertical arrows (or lines), and `\sarrowlength`, for slanted arrows (or lines). To change any of these dimensions, say, e.g., `\harrowlength=40pt'. As with all other TeX dimensions, the change may be as global or as local as you like. Furthermore, the placement of morphisms on the arrows is controlled by the dimensions `\hmorphposn`, `\vmorphposn`, and `\morphdist`. The first two dimensions control the horizontal and vertical position of the morphism from its default position; the latter dimension controls the distance of the morphism from the arrow. If you have more than one morphism per arrow (i.e., a ^/_ or `\lft`/`\rt` construction), use the parameters `\hmorphposnup`, `\hmorphposndn`, `\vmorphposnup`, `\vmorphposndn`, `\hmorphposnrt`, `\hmorphposnlft`, `\vmorphposnrt`, and `\vmorphposnlft`. The default values of all these dimensions are provided in the section on parameters that follows below.

There is a family of macros to produce horizontal lines, arrows, and adjoint arrows. The following macros produce horizontal maps and have the same syntax as `\mapright`:

```
\mapright
    $X\mapright Y$
\mapleft
    $X\mapleft Y$
\hline
    $X\hline Y$
\bimapright
    $X\bimapright Y$
\bimapleft
    $X\bimapleft Y$
\adjmapright
    $X\adjmapright Y$
\adjmapleft
    $X\adjmapleft Y$
\bihline
    $X\bihline Y$
```

There is also a family of macros to produce vertical lines, arrows, and adjoint arrows. The following macros produce vertical maps and have the same syntax as `\mapdown`:

`\mapdown`

(a down arrow)

`\mapup`
: (an up arrow)

`\vline`
: (vertical line)

`\bimapdown`
: (two down arrows)

`\bimapup`
: (two up arrows)

`\adjmapdown`
: (two adjoint arrows; down then up)

`\adjmapup`
: (two adjoint arrows; up then down)

`\bivline`
: (two vertical lines)

Finally, there is a family of macros to produce slanted lines, arrows, and adjoint arrows. The following macros produce slanted maps and have the same syntax as `\arrow`:

`\arrow`
: (a slanted arrow)

`\sline`
: (a slanted line)

`\biarrow`
: (two straight arrows)

`\adjarrow`
: (two adjoint arrows)

`\bisline`
: (two straight lines)

The width between double arrows is controlled by the parameter `\channelwidth`. There are also the parameters `\hchannel` and `\vchannel` which, if nonzero, override `\channelwidth` by controlling the horizontal and vertical shifting from the first arrow to the second.

There are no adornments on these arrows to distinguish inclusions from epimorphisms from monomorphisms. Many texts, such as Lang's book Algebra, use as a tasteful alternative the symbol `inc' (in roman) next to an arrow to denote inclusion.

Future enhancements will include a mechanism to draw curved arrows found in, e.g., the Snake Lemma, by employing a version of the `\path` macros of Appendix D of The TeXbook.

Construction of commutative diagrams

There are two approaches to the construction of commutative diagrams described here. The first approach, and the simplest, treats commutative diagrams like fancy matrices, as Knuth does in Exercise 18.46 of The

TeXbook. This case is covered by the macro `\commdiag`, which is an altered version of the Plain TeX macro `\matrix`. An example suffices to demonstrate this macro. The following commutative diagram (illustrating the covering homotopy property; Bott and Tu, Differential Forms in Algebraic Topology) is produced with the code

```
$$\commdiag{Y&\mapright^f&E\cr \mapdown&\arrow(3,2)\lft{f_t}&\mapdown\cr
Y\times I&\mapright^{\bar f_t}&X}$$
```

Of course, the parameters may be changed to produce a different effect. The following commutative diagram (illustrating the universal mapping property; Warner, Foundations of Differentiable Manifolds and Lie Groups) is produced with the code

```
$$\varrowlength=20pt
\commdiag{V\otimes W\cr \mapup\lft\phi&\arrow(3,-1)\rt{\tilde l}\cr
V\times W&\mapright^l&U\cr}$$
```

A diagram containing isosceles triangles is achieved by placing the apex of the triangle in the center column, as shown in the example (illustrating all constant minimal realizations of a linear system; Brockett, Finite Dimensional Linear Systems) which is produced with the code

```
$$\sarrowlength=.42\harrowlength
\commdiag{&R^m\cr &\arrow(-1,-1)\lft{\bf B}\quad \arrow(1,-1)\rt{\bf G}\cr
R^n&\mapright^{\bf P}&R^n\cr
\mapdown\lft{e^{{\bf A}t}}&&\mapdown\rt{e^{{\bf F}t}}\cr
R^n&\mapright^{\bf P}&R^n\cr
&\arrow(1,-1)\lft{\bf C}\quad \arrow(-1,-1)\rt{\bf H}\cr
&R^q\cr}$$
```

Other commutative diagram examples appear in the file `commdiags.tex`, which is distributed with this package.

In these examples the arrow lengths and line slopes were carefully chosen to blend with each other. In the first example, the default settings for the arrow lengths are used, but a direction for the arrow must be chosen. The ratio of the default horizontal and vertical arrow lengths is approximately the golden mean the arrow direction closest to this mean is `(3,2)`. In the second example, a slope of is desired and the default horizontal arrow length is 60 pt; therefore, choose a vertical arrow length of 20 pt. You may affect the interline glue settings of `\commdiag` by redefining the macro `\commdiagbaselines`. (cf. Exercise 18.46 of The TeXbook and the section on parameters below.)

The width, height, and depth of all morphisms are hidden so that the morphisms' size do not affect arrow positions. This can cause a large morphism at the top or bottom of a diagram to impinge upon the text surrounding the diagram. To overcome this problem, use TeX's `\noalign` primitive to insert a `\vskip` immediately above or below the offending line, e.g., `$$\commdiag{\noalign{\vskip6pt}X&\mapright^\int&Y\cr ...}'.

The macro `\commdiag` is too simple to be used for more complicated diagrams, which may have intersecting or overlapping arrows. A second approach, borrowed from Francis Borceux's Diagram macros for LaTeX, treats the commutative diagram like a grid of identically shaped boxes. To compose the commutative diagram, first draw an equally spaced grid, e.g., on a piece of scratch paper. Then draw each element (vertices and

arrows) of the commutative diagram on this grid, centered at each grid point. Finally, use the macro `\gridcommdiag` to implement your design as a TeX alignment. For example, the cubic diagram that appears in Francis Borceux's documentation can be implemented on a 7 by 7 grid, and is achieved with the code

```
$$\harrowlength=48pt \varrowlength=48pt \sarrowlength=20pt
\def\cross#1#2{\setbox0=\hbox{$#1$}%
  \hbox to\wd0{\hss\hbox{$#2$}\hss}\llap{\unhbox0}}
\gridcommdiag{&&B&&\mapright^b&&D\cr
&\arrow(1,1)\lft a&&&&\arrow(1,1)\lft d\cr
A&&\cross{\hmorphposn=12pt\mapright^c}{\vmorphposn=-12pt\mapdown\lft f}
&&C&&\mapdown\rt h\cr\cr
\mapdown\lft e&&F&&\cross{\hmorphposn=-12pt\mapright_j}
{\vmorphposn=12pt\mapdown\rt g}&&H\cr
&\arrow(1,1)\lft i&&&&\arrow(1,1)\rt l\cr
E&&\mapright_k&&G\cr}$$
```

The dimensions `\hgrid` and `\vgrid` control the horizontal and vertical spacing of the grid used by `\gridcommdiag`. The default setting for both of these dimensions is 15 pt. Note that in the example of the cube the arrow lengths must be adjusted so that the arrows overlap into neighboring boxes by the desired amount. Hence, the `\gridcommdiag` method, albeit more powerful, is less automatic than the simpler `\commdiag` method. Furthermore, the ad hoc macro `\cross` is introduced to allow the effect of overlapping arrows. Finally, note that the positions of four of the morphisms are adjusted by setting `\hmorphposn` and `\vmorphposn`.

One is not restricted to a square grid. For example, the proof of Zassenhaus's Butterfly Lemma can be illustrated by the diagram (appearing in Lang's book Algebra) This diagram may be implemented on a 9 by 12 grid with an aspect ratio of 1/2, and is set with the code

```
$$\hgrid=16pt \vgrid=8pt \sarrowlength=32pt
\def\cross#1#2{\setbox0=\hbox{$#1$}%
  \hbox to\wd0{\hss\hbox{$#2$}\hss}\llap{\unhbox0}}
\def\l#1{\llap{$#1$\hskip.5em}}
\def\r#1{\rlap{\hskip.5em$#1$}}
\gridcommdiag{&&U&&&&V\cr &&\bullet&&&&\bullet\cr
&&\sarrowlength=16pt\sline(0,1)&&&&\sarrowlength=16pt\sline(0,1)\cr
&&\l{u(U\cap V)}\bullet&&&&\bullet\r{(U\cap V)v}\cr
&&&\sline(2,-1)&&\sline(2,1)\cr
&&\cross{=}{\sline(0,1)}&&\bullet&&\cross{=}{\sline(0,1)}\cr\cr
&&\l{^{\textstyle u(U\cap v)}}\bullet&&\cross{=}{\sline(0,1)}&&
 \bullet\r{^{\textstyle(u\cap V)v}}\cr
&\sline(2,1)&&\sline(2,-1)&&\sline(2,1)&&\sline(2,-1)\cr
\l{u}\bullet&&&&\bullet&&&&\bullet\r{v}\cr
&\sline(2,-1)&&\sline(2,1)&&\sline(2,-1)&&\sline(2,1)\cr
&&\bullet&&&&\bullet\cr &&u\cap V&&&&U\cap v\cr}$$
```

Again, the construction of this diagram requires careful choices for the arrow lengths and is facilitated by the introduction of the ad hoc macros `\cross`, `\r`, and `\l`. Note also that superscripts were used to adjust the

position of the vertices Many diagrams may be typeset with the predefined macros that appear here; howe ingenuity is often required to handle special cases.

Commutative diagram parameters

The following is a list describing the parameters used in the commutative diagram macros. These dimensions may be changed globally or locally.

`\harrowlength`

(Default: 60 pt) The length of right or left arrows.

`\varrowlength`

(Default: 0.618`\harrowlength`) The length of up or down arrows.

`\sarrowlength`

(Default: 60 pt) The horizontal length of slanted arrows.

`\hmorphposn`

(Default: 0 pt) The horizontal position of the morphism with respect to its default position. There are also the dimensions `\hmorphposnup`, `\hmorphposndn`, `\hmorphposnrt`, and `\hmorphposnlft` for ^/_ or `\lft`/`\rt` constructions.

`\vmorphposn`

(Default: 0 pt) The vertical position of the morphism with respect to its default position. There are also the dimensions `\vmorphposnup`, `\vmorphposndn`, `\vmorphposnrt`, and `\vmorphposnlft` for ^/_ or `\lft`/`\rt` constructions.

`\morphdist`

(Default: 4 pt) The distance of morphisms from slanted lines or arrows.

`\channelwidth`

(Default: 3 pt) The distance between double lines or arrows.

`\hchannel, \vchannel`

(Defaults: 0 pt) Overrides `\channelwidth`. The horizontal and vertical shifts between double lines or arrows.

`\commdiagbaselines`

(Default: `\baselineskip=15pt \lineskip=3pt \lineskiplimit=3pt`) The parameters used by `\commdiag` for setting interline glue.

`\hgrid`

(Default: 15 pt) The horizontal spacing of the grid used by `\gridcommdiag`.

`\vgrid`

(Default: 15 pt) The vertical spacing of the grid used by `\gridcommdiag`.

Go to the previous, next section.

Go to the previous, next section.

Programming definitions

The definitions in this section are only likely to be useful when you are writing nontrivial macros, not for controlling the appearance of your document.

Category codes

Plain TeX defines `\active` (as the number 13) for use in changing category codes. Although the author of The TeXbook has "intentionally kept the category codes numeric", two other categories are commonly used: letters (category code 11) and others (12). Therefore, Eplain defines `\letter` and `\other`.

Sometimes it is cleaner to make a character active without actually writing a `\catcode` command. The `\makeactive` command takes a character as an argument to make active (and ignores following spaces). For example, here are two commands which both make \ active:

```
\makeactive`\\   \makeactive92
```

Usually, when you give a definition to an active character, you have to do so inside a group where you temporarily make the character active, and then give it a global definition (cf. the definition of `\obeyspaces` in The TeXbook). This is inconvenient if you are writing a long macro, or if the character already has a global definition you do not wish to transcend. Eplain provides `\letreturn`, which defines the usual end-of-line character to be the argument. For example:

```
\def\mymacro{... \letreturn\myreturn ... }
\mymacro hello
there
```

The end-of-line between `hello' and `there' causes `\myreturn` to be expanded.

The TeXbook describes `\uncatcodespecials`, which makes all characters which are normally "special" into "other" characters, but the definition never made it into plain TeX. Eplain therefore defines it.

Finally, `\percentchar` expands into a literal `%' character. This is useful when you `\write` TeX output to a file, and want to avoid spurious spaces. For example, Eplain writes a `\percentchar` after the definition of cross-references. The macros `\lbracechar` and `\rbracechar` expand similarly.

Allocation macros

Plain TeX provides macros that allocate registers of each primitive type in TeX, to prevent different sets of macros from using the same register for two different things. The macros are all named starting with `new', e.g., `\newcount` allocates a new "count" (integer) register. Such allocations are usually needed

only at the top level of some macro definition file; therefore, plain TeX makes the allocation register `\outer`, to help find errors. (The error this helps to find is a missing right brace in some macro definition.)

Sometimes, however, it is useful to allocate a register as part of some macro. An outer control sequence cannot be used as part of a macro definition (or in a few other contexts: the parameter text of a definition, an argument to a definition, the preamble of an alignment, or in conditional text that is being skipped). Therefore, Eplain defines "inner" versions of all the allocation macros, named with the prefix `inner': `\innernewbox`, `\innernewcount`, `\innernewdimen`, `\innernewfam`, `\innernewhelp`, `\innernewif`, `\innernewinsert`, `\innernewlanguage`, `\innernewread`, `\innernewskip`, `\innernewtoks`, `\innernewwrite`.

You can also define non-outer versions of other macros in the same way that Eplain defines the above. The basic macro is called `\innerdef`:

```
\innerdef \innername {outername}
```

The first argument (\innername) to `\innerdef` is the control sequence that you want to define. Any previous definition of \innername is replaced. The second argument (outername) is the *characters* in the name of the outer control sequence. (You can't use the actual control sequence name, since it's outer!)

If the outer control sequence is named \cs, and you want to define `innercs` as the inner one, you can use `\innerinnerdef`, which is just an abbreviation for a call to `\innerdef`. For example, these two calls are equivalent:

```
\innerdef\innerproclaim{proclaim}
\innerinnerdef{proclaim}
```

Iteration

You can iterate through a comma-separated list of items with `\for`. Here is an example:

```
\for\name:=karl,kathy\do{%
   \message{\name}%
}%
```

This writes `karl' and `kathy' to the terminal. Spaces before or after the commas in the list, or after the `:=`, are *not* ignored.

`\for` expands the iterated values fully (with `\edef`), so this is equivalent to the above:

```
\def\namelist{karl,kathy}%
\for\name:=\namelist\do ...
```

Macro arguments

It is occasionally useful to redefine a macro that takes arguments to do nothing. Eplain defines `\gobble`, `\gobbletwo`, and `\gobblethree` to swallow one, two, and three arguments, respectively.

For example, if you want to produce a "short" table of contents--one that includes only chapters, say--the easiest thing to do is read the entire `.toc` file (see section Contents), and just ignore the commands that produce section or subsection entries. To be specific:

```
\let\tocchapterentry = \shorttocchapter
\let\tocsectionentry = \gobbletwo
\let\tocsubsectionentry = \gobbletwo
\readtocfile
```

(Of course, this assumes you only have chapters, sections, and subsections in your document.)

In addition, Eplain defines `\eattoken` to swallow the single following token, using `\let`. Thus, `\gobble` followed by `{...}' ignores the entire brace-enclosed text. `\eattoken` followed by the same ignores only the opening left brace.

Eplain defines a macro `\identity` which takes one argument and expands to that argument. This may be useful if you want to provide a function for the user to redefine, but don't need to do anything by default. (For example, the default definition of `\eqconstruct` (see section Formatting equation references) is `\identity`.)

You may also want to read an optional argument. The established convention is that optional arguments are put in square brackets, so that is the syntax Eplain recognizes. Eplain ignores space tokens before an optional argument, via `\futurenonspacelet`.

You test for an optional argument by using `\@getoptionalarg`. It takes one argument, a control sequence to expand after reading the argument, if present. If an optional argument is present, the control sequence `\@optionalarg` expands to it; otherwise, `\@optionalarg` is `\empty`. You must therefore have the category code of @ set to 11 (letter). Here is an example:

```
\catcode`@=\letter
\def\cmd{\@getoptionalarg\finishcmd}
\def\finishcmd{%
  \ifx\@optionalarg\empty
    % No optional argument present.
  \else
    % One was present.
  \fi
}
```

If an optional argument contains another optional argument, the inner one will need to be enclosed in braces, so TeX does not mistake the end of the first for the end of the second.

Converting to characters

Eplain defines `\xrlabel`. to produce control sequence names for cross-reference labels, et al. This macro expands to its argument with an `_' appended. (It does this because the usual use of `\xrlabel` is to generate a control sequence name, and conflicts between control sequence names would lead to obscure bugs.)

Because `\xrlabel` is fully expandable, to make a control sequence name out of the result you need only do

```
\csname \xrlabel{label}\endcsname
```

The `\csname` primitive makes a control sequence name out of any sequence of character tokens, regardless of category code. Labels can therefore include any characters except for `\', `{', `}', and `#', all of which are used in macro definitions themselves.

`\sanitize` takes a control sequence as an argument and converts the expansion of the control sequence into a list of character tokens. This is the behavior you want when writing information like chapter titles to an output file. For example, here is part of the definition of `\writenumberedtocentry`; `#2` is the title that the user has given.

```
...
\def\temp{#2}%
...
  \write\tocfile{%
    ...
    \sanitize\temp
    ...
  }%
```

Expansion

This section describes some miscellanous macros for expansion, etc.

\csn and \ece

`\csn`{name} simply abbreviates `\csname` name `\encsname`, thus saving some typing. The extra level of expansion does take some time, though, so I don't recommend it for an inner loop.

`\ece`{token}{name} abbreviates

```
\expandafter token \csname name \endcsname
```

For example,


```
\def\fontabbrevdef#1#2{\ece\def{@#1font}{#2}}
\fontabbrevdef{normal}{ptmr}
```

defines a control sequence `\@normalfont` to expand to `ptmr`.

\edefappend

`\edefappend` is a way of adding on to an existing definition. It takes two arguments: the first is the control sequence name, the second the new tokens to append to the definition. The second argument is fully expanded (in the `\edef` that redefines the control sequence).

For example:

```
\def\foo{abc}
\def\bar{xyz}
\edefappend\foo{\bar karl}
```

results in `\foo` being defined as `abcxyzkarl'.

Hooks

A hook is simply a name for a group of actions which is executed in certain places--presumably when it is most useful to allow customization or modification. TeX already provides many builtin hooks; for example, the `\every ...` token lists are all examples of hooks.

Eplain provides several macros for adding actions to hooks. They all take two arguments: the name of the hook and the new actions.

`hookaction name actions`

`hookappend name actions`

`hookprepend name actions`

Each of these adds actions to the hook name. (Any previously-defined actions are retained.) name is not a control sequence, but rather the characters of the name.

`hookactiononce name \cs`

`\hookactiononce` adds cs to name, like the macros above, but first it adds

```
\global\let \cs \relax
```

to the definition of \cs. (This implies \cs must be a true expandable macro, not a control sequence `\let` to a primitive or some other such thing.) Thus, \cs is expanded the next time the hook name is run, but it will disappear after that.

The `\global` is useful because `\hookactiononce` is most useful when the grouping structure of the TeX code could be anything. Neither this nor the other hook macros do global assignments to the hook variable itself, so TeX's usual grouping rules apply.

The companion macro to defining hook actions is `\hookrun`, for running them. This takes a single argument, the name of the hook. If no actions for the hook are defined, no error ensues.

Here is a skeleton of general `\begin` and `\end` macros that run hooks, and a couple of calls to define actions. The use of `\hookprepend` for the begin action and `\hookappend` for the end action ensures that the actions are executed in proper sequence with other actions (as long as the other actions use `\hookprepend` and `\hookappend` also).

```
\def\begin#1{ ... \hookrun{begin} ... }
\def\end#1{ ... \hookrun{end} ... }
\hookprepend{begin}\start_underline
\hookappend{end}\finish_underline
```

Properties

A property is a name/value pair associated with another symbol, traditionally called an atom. Both atom and property names are control sequence names.

Eplain provides two macros for dealing with property lists: `\setproperty` and `\getproperty`.

`\setproperty atom propname value`

> `\setproperty` defines the property property on the atom atom to be value. atom and propname can be anything acceptable to `\csname`. value can be anything.

`\getproperty atom propname`

> `\getproperty` expands to the value stored for propname on atom. If propname is undefined, it expands to nothing (i.e., `\empty`).

The idea of properties originated in Lisp (I believe). There, the implementation truly does associate properties with atoms. In TeX, where we have no builtin support for properties, the association is only conceptual.

The following example typesets `xyz'.

```
\setproperty{a}{pr}{xyz}
\getproperty{a}{pr}
```

\expandonce

`\expandonce` is defined as `\expandafter\noexpand`. Thus, `\expandonce token` expands token once, instead of to TeX primitives. This is most useful in an `\edef`.

For example, the following defines `\temp` to be `\foo`, not `abc'.

```
\def\foo{abc}
\def\bar{\foo}
\edef\temp{\expandonce\bar}
```

\ifundefined

`\ifundefined{cs} t \else f \fi` expands the t text if the control sequence `\cs` is undefined or has been `\let` to `\relax`, and the f text otherwise.

Since `\ifundefined` is not a primitive conditional, it cannot be used in places where TeX might skip tokens "at high speed", e.g., within another conditional---TeX can't match up the `\if`'s and `\fi`'s.

This macro was taken directly from The TeXbook, page 308.

\futurenonspacelet

The `\futurelet` primitive allows you to look at the next token from the input. Sometimes, though, you want to look ahead ignoring any spaces. This is what `\futurenonspacelet` does. It is otherwise the same as `\futurelet`: you give it two control sequences as arguments, and it assigns the next nonspace token to the first, and then expands the second. For example:

```
\futurenonspacelet\temp\finishup
\def\finishup{\ifx\temp ...}
```

Obeying spaces

`\obeywhitespace` makes both end-of-lines and space characters in the input be respected in the output. Unlike plain TeX's `\obeyspaces`, even spaces at the beginnings of lines turn into blank space.

By default, the size of the space that is produced by a space character is the natural space of the current font, i.e., what `\ ` produces.

Ordinarily, a blank line in the input produces as much blank vertical space as a line of text would occupy. You can adjust this by assigning to the parameter `\blanklineskipamount`: if you set this negative, the space produced by a blank line will be smaller; if positive, larger.

Tabs are not affected by this routine. In particular, if tabs occur at the beginning of a line, they will disappear. (If you are trying to make TeX do the "right thing" with tabs, don't. Use a utility program like *expand* instead.)

Writing out numbers

`\numbername` produces the written-out form of its argument, i.e., `zero' through `ten' for the numbers 0--10, and numerals for all others.

Mode-specific penalties

TeX's built-in `\penalty` command simply appends to the current list, no matter what kind of list it is. You might intend a particular penalty to always be a "vertical" penalty, however, i.e., appended to a vertical list. Therefore, Eplain provides `\vpenalty` and `\hpenalty` which first leave the other mode and then do `\penalty`.

More precisely, `\vpenalty` inserts `\par` if the current mode is horizontal, and `\hpenalty` inserts `\leavevmode` if the current mode is vertical. (Thus, `\vpenalty` cannot be used in math mode.)

Auxiliary files

It is common to write some information out to a file to be used on a subsequent run. But when it is time to read the file again, you only want to do so if the file actually exists. `\testfileexistence` is given an argument which is appended to `\jobname`, and sets the conditional `\iffileexists` appropriately.

For example:

```
\testfileexistence{toc}%
\iffileexists
   \input \jobname.toc
\fi
```

Go to the previous, next section.

Go to the previous, next section.

TOC

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

TOC

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on

the Program, the distribution of the whole must be on the terms of this License, whose permission other licensees extend to the entire whole, and thus to each and every part regardless of who wrote

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
    1. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
    2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
    3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

    The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

    If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on

it.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
```


```
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
`Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Go to the previous, next section.

Go to the previous, next section.

Regain your programming freedom

Until a few years ago, programmers in the United States could write any program they wished. This freedom has now been taken away by two developments: software patents, which grant the patent holder an absolute monopoly on some programming technique, and user interface copyright, which forbid compatible implementations of an existing user interface.

In Europe, especially through the GATT treaty, things are rapidly approaching the same pass.

Software patents

The U.S. Patent and Trademark Office has granted numerous software patents on software techniques. Patents are an absolute monopoly--independent reinvention is precluded. This monopoly lasts for seventeen years, i.e., forever (with respect to computer science).

One patent relevant to TeX is patent 4,956,809, issued to the Mark Williams company on September 11, 1990, applied for in 1982, which covers (among other things)

> representing in a standardized order consisting of a standard binary structure file stored on auxiliary memory or transported on a communications means, said standardized order being different from a different order used on at least one of the different computers;
>
> Converting in each of the different computers binary data read from an auxiliary data storage or communications means from the standardized order to the natural order of the respective host computer after said binary data are read from said auxiliary data storage or communications means and before said binary data are used by the respective host computer; and
>
> Converting in each of the different computers binary data written into auxiliary data storage or communications means from the natural order of the respective host computer to the standardized order prior to said writing.

... in other words, storing data on disk in a machine-independent order, as the DVI, TFM, GF, and PK file formats specify. Even though TeX is "prior art" in this respect, the patent was granted (the patent examiners not being computer scientists, even less computer typographers). Since there is a strong presumption in the courts of a patent's validity once it has been granted, there is a good chance that users or implementors of TeX could be successfully sued on the issue.

As another example, the X window system, which was intended to be able to be used freely by everyone, is now being threatened by two patents: 4,197,590 on the use of exclusive-or to redraw cursors, held by Cadtrak, a litigation company (this has been upheld twice in court); and 4,555,775, held by AT&T, on the use of backing store to redraw windows quickly.

Here is one excerpt from a recent mailing by the League for Programming Freedom (see section What to

do?) which I feel sums up the situation rather well. It comes from an article in Think magazine, issue 1990. The comments after the quote were written by Richard Stallman.

> "You get value from patents in two ways," says Roger Smith, IBM Assistant General Counsel, intellectual property law. "Through fees, and through licensing negotiations that give IBM access to other patents.
>
> "The IBM patent portfolio gains us the freedom to do what we need to do through cross-licensing--it gives us access to the inventions of others that are the key to rapid innovation. Access is far more valuable to IBM than the fees it receives from its 9,000 active patents. There's no direct calculation of this value, but it's many times larger than the fee income, perhaps an order of magnitude larger."

This information should dispel the belief that the patent system will "protect" a small software developer from competition from IBM. IBM can always find patents in its collection which the small developer is infringing, and thus obtain a cross-license.

However, the patent system does cause trouble for the smaller companies which, like IBM, need access to patented techniques in order to do useful work in software. Unlike IBM, the smaller companies do not have 9,000 patents and cannot usually get a cross-license. No matter how hard they try, they cannot have enough patents to do this.

Only the elimination of patents from the software field can enable most software developers to continue with their work.

The value IBM gets from cross-licensing is a measure of the amount of harm that the patent system would do to IBM if IBM could not avoid it. IBM's estimate is that the trouble could easily be ten times the good one can expect from one's own patents--even for a company with 9,000 of them.

User interface copyright

(This section is copied from the GCC manual, by Richard Stallman.)

> *This section is a political message from the League for Programming Freedom to the users of Eplain. It is included here as an expression of support for the League on my part.*

Apple, Lotus and Xerox are trying to create a new form of legal monopoly: a copyright on a class of user interfaces. These monopolies would cause serious problems for users and developers of computer software and systems.

Until a few years ago, the law seemed clear: no one could restrict others from using a user interface; programmers were free to implement any interface they chose. Imitating interfaces, sometimes with changes, was standard practice in the computer field. The interfaces we know evolved gradually in this way; for example, the Macintosh user interface drew ideas from the Xerox interface, which in turn drew on work done at Stanford and SRI. 1-2-3 imitated VisiCalc, and dBase imitated a database program from JPL.

Most computer companies, and nearly all computer users, were happy with this state of affairs. The companies that are suing say it does not offer "enough incentive" to develop their products, but they must

have considered it "enough" when they made their decision to do so. It seems they are not satisfied w the opportunity to continue to compete in the marketplace--not even with a head start.

If Xerox, Lotus, and Apple are permitted to make law through the courts, the precedent will hobble the software industry:

- Gratuitous incompatibilities will burden users. Imagine if each car manufacturer had to arrange the pedals in a different order.
- Software will become and remain more expensive. Users will be "locked in" to proprietary interfaces, for which there is no real competition.
- Large companies have an unfair advantage wherever lawsuits become commonplace. Since they can easily afford to sue, they can intimidate small companies with threats even when they don't really have a case.
- User interface improvements will come slower, since incremental evolution through creative imitation will no longer be permitted.
- Even Apple, etc., will find it harder to make improvements if they can no longer adapt the good ideas that others introduce, for fear of weakening their own legal positions. Some users suggest that this stagnation may already have started.
- If you use GNU software, you might find it of some concern that user interface copyright will make it hard for the Free Software Foundation to develop programs compatible with the interfaces that you already know.

What to do?

(This section is copied from the GCC manual, by Richard Stallman.)

To protect our freedom from lawsuits like these, a group of programmers and users have formed a new grass-roots political organization, the League for Programming Freedom.

The purpose of the League is to oppose new monopolistic practices such as user-interface copyright and software patents; it calls for a return to the legal policies of the recent past, in which these practices were not allowed. The League is not concerned with free software as an issue, and not affiliated with the Free Software Foundation.

The League's membership rolls include John McCarthy, inventor of Lisp, Marvin Minsky, founder of the Artificial Intelligence lab, Guy L. Steele, Jr., author of well-known books on Lisp and C, as well as Richard Stallman, the developer of GNU CC. Please join and add your name to the list. Membership dues in the League are $42 per year for programmers, managers and professionals; $10.50 for students; $21 for others.

The League needs both activist members and members who only pay their dues.

To join, or for more information, phone (617) 492-0023 or write to:

```
League for Programming Freedom
1 Kendall Square #143
```


```
P.O. Box 9171
Cambridge, MA 02139
```

You can also send electronic mail to `league@prep.ai.mit.edu`.

Here are some suggestions from the League for things you can do to protect your freedom to write programs:

- Don't buy from Xerox, Lotus or Apple. Buy from their competitors or from the defendants they are suing.
- Don't develop software to work with the systems made by these companies.
- Port your existing software to competing systems, so that you encourage users to switch.
- Write letters to company presidents to let them know their conduct is unacceptable.
- Tell your friends and colleagues about this issue and how it threatens to ruin the computer industry.
- Above all, don't work for the look-and-feel plaintiffs, and don't accept contracts from them.
- Write to Congress to explain the importance of this issue.

```
House Subcommittee on Intellectual Property
2137 Rayburn Bldg
Washington, DC 20515

Senate Subcommittee on Patents, Trademarks and Copyrights
United States Senate
Washington, DC 20510
```

(These committees have received lots of mail already; let's give them even more.)

Express your opinion! You can make a difference.

Go to the previous, next section.

Go to the previous, next section.

TOC

Macro index

.

- .aux file
- .bbl file
- .bib file
- .bst files
- .fmt file
- .toc file

a

- abovecolumnspenalty
- abovedoublecolumnskip
- abovelistpenalty
- abovelistskip
- abovelistskipamount
- adjarrow
- adjmapdown
- adjmapleft
- adjmapright
- adjmapup
- advancebottommargin
- advanceleftmargin
- advancerightmargin
- advancetopmargin
- AMSLaTeX
- AMSTeX
- arrow

TOC

b

c

d

e

f

g

h

i

j

l

m

n

o

p

r

s

- sanitize
- sarrowlength
- setproperty
- setuplistinghook
- singlecolumn
- sline
- SLiTeX
- specialpathdelimiters (conditional)
- subeqnumber

t

- testfileexistence
- TeX
- timestamp
- timestring
- toc...entry
- tocfilebasename
- today
- topmargin
- tracingall
- tracingboxes
- tracingoff

u

- uncatcodespecials
- unorderedlist
- unorderedlistdepth
- unorderedmarker

v

w

x

Go to the previous, next section.

Go to the previous section.

Concept index

a

b

c

d

e

f

g

h

i

j

k

l

m

TOC

n

o

p

r

s

t

u

v

W

Z

Go to the previous section.

Forms Mode User's Manual

Forms Mode User's Manual

Forms Mode User's Manual

Forms-Mode version 2.3

September 1993

Johan Vromans *jv@nl.net*

Forms mode is an Emacs major mode for working with simple textual data bases in a forms-oriented manner. In Forms mode, the information in these files is presented in an Emacs window in a user-defined format, one record at a time. The user can view records or modify their contents.

Forms mode is not a simple major mode, but requires two files to do its job: a control file and a data file. The data file holds the actual data to be presented. The control file describes how to present it.

Forms Example

Let's illustrate Forms mode with an example. Suppose you are looking at the `/etc/passwd' file, and the screen looks like this:

```
====== /etc/passwd ======

User : root   Uid: 0   Gid: 1

Name : Super User

Home : /

Shell: /bin/sh
```

As you can see, the familiar fields from the entry for the super user are all there, but instead of being colon-separated on one single line, they make up a forms.

The contents of the forms consists of the contents of the fields of the record (e.g. `root', `0', `1', `Super User') interspersed with normal text (e.g `User : ', `Uid: ').

If you modify the contents of the fields, Forms mode will analyze your changes and update the file appropriately. You cannot modify the interspersed explanatory text (unless you go to some trouble about it), because that is marked read-only (see section 'Text Properties' in The Emacs Lisp Reference Manual).

The Forms mode control file specifies the relationship between the format of `/etc/passwd' and what appears on the screen in Forms mode. See section Control File Format.

Entering and Exiting Forms Mode

M-x forms-find-file RET control-file RET

Visit a database using Forms mode. Specify the name of the **control file**, not the data file!

M-x forms-find-file-other-window RET control-file RET

Similar, but displays the file in another window.

The command `forms-find-file` evaluates the file control-file, and also visits it in Forms mode. What you see in its buffer is not the contents of this file, but rather a single record of the corresponding data file that is visited in its own buffer. So there are two buffers involved in Forms mode: the forms buffer that is initially used to visit the control file and that shows the records being browsed, and the data buffer that holds the data file being visited. The latter buffer is normally not visible.

Initially, the first record is displayed in the forms buffer. The mode line displays the major mode name `Forms', followed by the minor mode `View' if the data base is read-only. The number of the current record (n) and the total number of records in the file(t) are shown in the mode line as `n/t'. For example:

```
--%%-Emacs: passwd-demo          (Forms View 1/54)----All-------
```

If the buffer is not read-only, you may change the buffer to modify the fields in the record. When you move to a different record, the contents of the buffer are parsed using the specifications in `forms-format-list`, and the data file is updated. If the record has fields that aren't included in the display, they are not changed.

Entering Forms mode runs the normal hook `forms-mode-hooks` to perform user-defined customization.

To save any modified data, you can use C-x C-s (`save-buffer`). This does not save the forms buffer (which would be rather useless), but instead saves the buffer visiting the data file.

To terminate Forms mode, you can use C-x C-s (`save-buffer`) and then kill the forms buffer. However, the data buffer will still remain. If this is not desired, you have to kill this buffer too.

Forms Commands

The commands of Forms mode belong to the C-c prefix, with one exception: TAB, which moves to the next field. Forms mode uses different key maps for normal mode and read-only mode. In read-only Forms mode, you can access most of the commands without the C-c prefix, but you must type ordinary letters instead of control characters; for example, type n instead of C-c C-n.

C-c C-n

Show the next record (`forms-next-record`). With a prefix argument n, show the nth next record.

C-c C-p

Show the previous record (`forms-prev-record`). With a prefix argument n, show the nth previous record.

C-c C-l

Jump to a record by number (`forms-jump-record`). Specify the record number with a prefix argument.

C-c <

Jump to the first record (`forms-first-record`).

C-c >

Jump to the last record (`forms-last-record`). This command also recalculates the number of records in the data file.

TAB

C-c TAB

Jump to the next field in the current record (`forms-next-field`). With a numeric argument n, jump forward n fields. If this command would move past the last field, it wraps around to the first field.

C-c C-q

Toggles read-only mode (`forms-toggle-read-only`). In read-only Forms mode, you cannot edit the fields; most Forms mode commands can be accessed without the prefix C-c if you use the normal letter instead (for example, type n instead of C-c C-n). In edit mode, you can edit the fields and thus change the contents of the data base; you must begin Forms mode commands with `C-c`. Switching to edit mode is allowed only if you have write access to the data file.

C-c C-o

Create a new record and insert it before the current record (`forms-insert-record`). It starts out with empty (or default) contents for its fields; you can then edit the fields. With a prefix argument, the new record is created *after* the current one. See also `forms-modified-record-filter` in section Modifying The Forms Contents.

C-c C-k

Delete the current record (`forms-delete-record`). You are prompted for confirmation before the record is deleted unless a prefix argument has been provided.

C-c C-s regexp RET

Search for regexp in all records following this one (`forms-search`). If found, this record is shown. If you give an empty argument, the previous regexp is used again.

M-x forms-prev-field

Similar to `forms-next-field` but moves backwards.

In addition, commands such as C-x C-s (`save-buffer`) and M-x revert-buffer are useful in Forms mode just as in other modes.

The following function key definitions are set up in Forms mode (whether read-only or not):

next

forms-next-record

prior

forms-prev-record

begin

forms-first-record

end

forms-last-record

S-Tab

forms-prev-field

Data File Format

Files for use with Forms mode are very simple--each record (usually one line) forms the contents of one form. Each record consists of a number of fields, which are separated by the value of the string `forms-field-sep`, which is `"\t"` (a tab) by default.

Fields may contain text which shows up in the forms in multiple lines. These lines are separated in the field using a "pseudo-newline" character which is defined by the value of the string `forms-multi-line`. Its default value is `"\^k"`. If it is set to `nil`, multiple line fields are prohibited.

Control File Format

The Forms mode control file serves two purposes. First, it names the data file to use, and defines its format and properties. Second, the Emacs buffer it occupies is used by Forms mode to display the forms.

The contents of the control file are evaluated as a Lisp program. It should set the following Lisp variables to suitable values:

`forms-file`

This variable specifies the name of the data file. Example:


```
(setq forms-file "my/data-file")
```

`forms-format-list`

This variable describes the way the fields of the record are formatted on the screen. For details, see section The Format Description.

`forms-number-of-fields`

This variable holds the number of fields in each record of the data file. Example:

```
(setq forms-number-of-fields 10)
```

If the control file doesn't set all of these variables, Forms mode reports an error.

The control file can optionally set the following additional Forms mode variables. Most of them have default values that are good for most applications.

`forms-field-sep`

This variable may be used to designate the string which separates the fields in the records of the data file. If not set, it defaults to the string `"\t"` (a tab character). Example:

```
(setq forms-field-sep "\t")
```

`forms-read-only`

If the value is non-`nil`, the data file is treated read-only. (Forms mode also treats the data file as read-only if you don't have access to write it.) Example:

```
(set forms-read-only t)
```

`forms-multi-line`

This variable specifies the pseudo newline separator that allows multi-line fields. This separator goes between the "lines" within a field--thus, the field doesn't really contain multiple lines, but it appears that way when displayed in Forms mode. If the value is `nil`, multi-line text fields are prohibited. The pseudo newline must not be a character contained in `forms-field-sep`.

The default value is `"\^k"`, so the default pseudo newline is the character control-k. Example:

```
(setq forms-multi-line "\^k")
```

`forms-new-record-filter`

This variable holds a function to be called whenever a new record is created to supply default values for fields. If it is `nil`, no function is called. See section Modifying The Forms Contents, for details.

`forms-modified-record-filter`

This variable holds a function to be called whenever a record is modified, just before updating the Forms data file. If it is `nil`, no function is called. See section Modifying The Forms Contents, for details.

The Format Description

The variable `forms-format-list` specifies the format of the data in the data file, and how to convert the data for display in Forms mode. Its value must be a list of Forms mode formatting elements, each of which can be a string, a number, a Lisp list, or a Lisp symbol that evaluates to one of those. The formatting elements are processed in the order they appear in the list.

string

A string formatting element is inserted in the forms "as is," as text that the user cannot alter.

number

A number element selects a field of the record. The contents of this field are inserted in the display at this point. Field numbers count starting from 1 (one).

list

A formatting element that is a list specifies a function call. This function is called every time a record is displayed, and its result, which must be a string, is inserted in the display text. The function should do nothing but returning a string.

The function you call can access the fields of the record as a list in the variable `forms-fields`.

symbol

A symbol used as a formatting element should evaluate to a string, number, or list; the value is interpreted as a formatting element, as described above.

If a record does not contain the number of fields as specified in `forms-number-of-fields`, a warning message will be printed. Excess fields are ignored, missing fields are set to empty.

The control file which displays `` `/etc/passwd' `` file as demonstrated in the beginning of this manual might look as follows:

```
;; This demo visits `/etc/passwd'.

(setq forms-file "/etc/passwd")
(setq forms-number-of-fields 7)
(setq forms-read-only t)                ; to make sure
(setq forms-field-sep ":")
;; Don't allow multi-line fields.
(setq forms-multi-line nil)

(setq forms-format-list
      (list
       "====== /etc/passwd ======\n\n"
       "User : "    1
       "   Uid: "   3
       "   Gid: "   4
       "\n\n"
```


```
        "Name : "    5
        "\n\n"
        "Home : "    6
        "\n\n"
        "Shell: "    7
        "\n"))
```

When you construct the value of `forms-format-list`, you should usually either quote the whole value, like this,

```
(setq forms-format-list
      '(
        "====== " forms-file " ======\n\n"
        "User : "    1
        (make-string 20 ?-)
        ...
       ))
```

or quote the elements which are lists, like this:

```
(setq forms-format-list
       (list
        "====== " forms-file " ======\n\n"
        "User : "    1
        '(make-string 20 ?-)
        ...
       ))
```

Forms mode validates the contents of `forms-format-list` when you visit a database. If there are errors, processing is aborted with an error message which includes a descriptive text. See section Error Messages, for a detailed list of error messages.

Modifying The Forms Contents

If `forms-read-only` is `nil`, the user can modify the fields and records of the database.

All normal editing commands are available for editing the contents of the displayed record. You cannot delete or modify the fixed, explanatory text that comes from string formatting elements, but you can modify the actual field contents.

If the variable `forms-modified-record-filter` is non-`nil`, it is called as a function before the new data is written to the data file. The function receives one argument, a vector that contains the contents of the fields of the record.

The function can refer to fields with `aref` and modify them with `aset`. The first field has number 1 (one); thus, element 0 of the vector is not used. The function should return the same vector it was passed;

the (possibly modified) contents of the vector determine what is actually written in the file. Here is an example:

```
(defun my-modified-record-filter (record)
  ;; Modify second field.
  (aset record 2 (current-time-string))
  ;; Return the field vector.
  record)

(setq forms-modified-record-filter 'my-modified-record-filter)
```

If the variable `forms-new-record-filter` is non-`nil`, its value is a function to be called to fill in default values for the fields of a new record. The function is passed a vector of empty strings, one for each field; it should return the same vector, with the desired field values stored in it. Fields are numbered starting from 1 (one). Example:

```
(defun my-new-record-filter (fields)
  (aset fields 5 (login-name))
  (aset fields 1 (current-time-string))
  fields)

(setq forms-new-record-filter 'my-new-record-filter)
```

Miscellaneous

The global variable `forms-version` holds the version information of the Forms mode software.

It is very convenient to use symbolic names for the fields in a record. The function `forms-enumerate` provides an elegant means to define a series of variables whose values are consecutive integers. The function returns the highest number used, so it can be used to set `forms-number-of-fields` also. For example:

```
(setq forms-number-of-fields
      (forms-enumerate
       '(field1 field2 field3 ...)))
```

This sets `field1` to 1, `field2` to 2, and so on.

Care has been taken to keep the Forms mode variables buffer-local, so it is possible to visit multiple files in Forms mode simultaneously, even if they have different properties.

If you have visited the control file in normal fashion with `find-file` or a like command, you can switch to Forms mode with the command `M-x forms-mode`. If you put `-*- forms -*-' in the first line of the control file, then visiting it enables Forms mode automatically. But this makes it hard to edit the control file itself, so you'd better think twice before using this.

The default format for the data file, using `"\t"` to separate fields and `"\^k"` to separate lines within a field, matches the file format of some popular database programss, e.g. FileMaker. So `forms-mode` can decrease the need to use proprietary software.

Error Messages

This section describes all error messages which can be generated by forms mode. Error messages that result from parsing the control file all start with the text `Forms control file error'. Messages generated while analyzing the definition of `forms-format-list` start with `Forms format error'.

`Forms control file error: 'forms-file' has not been set`

The variable `forms-file` was not set by the control file.

`Forms control file error: 'forms-number-of-fields' has not been set`

The variable `forms-number-of-fields` was not set by the control file.

`Forms control file error: 'forms-number-of-fields' must be a number > 0`

The variable `forms-number-of-fields` did not contain a positive number.

`Forms control file error: 'forms-field-sep' is not a string`

`Forms control file error: 'forms-multi-line' must be nil or a one-character string`

The variable `forms-multi-line` was set to something other than `nil` or a single-character string.

`Forms control file error: 'forms-multi-line' is equal to 'forms-field-sep'`

The variable `forms-multi-line` may not be equal to `forms-field-sep` for this would make it impossible to distinguish fields and the lines in the fields.

`Forms control file error: 'forms-new-record-filter' is not a function`

`Forms control file error: 'forms-modified-record-filter' is not a function`

The variable has been set to something else than a function.

`Forms control file error: 'forms-format-list' has not been set`

`Forms control file error: 'forms-format-list' is not a list`

The variable `forms-format-list` was not set to a Lisp list by the control file.

`Forms format error: field number xx out of range 1..nn`

A field number was supplied in `forms-format-list` with a value of xx, which was not greater than zero and smaller than or equal to the number of fields in the forms, nn.

`Forms format error: not a function fun`

The first element of a list which is an element of `forms-format-list` was not a valid Lisp function.

`Forms format error: invalid element xx`

A list element was supplied in `forms-format-list` which was not a string, number or list.

`Warning: this record has xx fields instead of yy`

The number of fields in this record in the data file did not match `forms-number-of-fields`. Missing fields will be made empty.

`Multi-line fields in this record - update refused!`

The current record contains newline characters, hence can not be written back to the data file, for it would corrupt it. Probably you inserted a newline in a field, while `forms-multi-line` was `nil`.

`Record number xx out of range 1..yy`

A jump was made to non-existing record xx. yy denotes the number of records in the file.

`Stuck at record xx`

An internal error prevented a specific record from being retrieved.

`No write access to "file"`

An attempt was made to enable edit mode on a file that has been write protected.

`"regexp" not found`

The regexp could not be found in the data file, starting at the current record location.

`Warning: number of records changed to nn`

Forms mode's idea of the number of records has been adjusted to the number of records actually present in the data file.

`Problem saving buffers?`

An error occurred while saving the data file buffer. Most likely, Emacs did ask to confirm deleting the buffer because it had been modified, and you said `no'.

Long Example

The following example exploits most of the features of Forms mode. This example is included in the distribution as file `forms-d2.el'.

```
;; demo2 -- demo forms-mode     -*- emacs-lisp -*-

;; SCCS Status     : demo2      1.1.2
;; Author          : Johan Vromans
;; Created On      : 1989
;; Last Modified By: Johan Vromans
;; Last Modified On: Mon Jul  1 13:56:31 1991
;; Update Count    : 2
;; Status          : OK
;;
;; This sample forms exploit most of the features of forms mode.
```


```
;; Set the name of the data file.
(setq forms-file "forms-d2.dat")

;; Use forms-enumerate to set field names and number thereof.
(setq forms-number-of-fields
      (forms-enumerate
       '(arch-newsgroup                 ; 1
         arch-volume                    ; 2
         arch-issue                     ; and ...
         arch-article                   ; ... so
         arch-shortname                 ; ... ... on
         arch-parts
         arch-from
         arch-longname
         arch-keywords
         arch-date
         arch-remarks)))

;; The following functions are used by this form for layout purposes.
;;
(defun arch-tocol (target &optional fill)
  "Produces a string to skip to column TARGET.
Prepends newline if needed.
The optional FILL should be a character, used to fill to the column."
  (if (null fill)
      (setq fill ? ))
  (if (< target (current-column))
      (concat "\n" (make-string target fill))
    (make-string (- target (current-column)) fill)))
;;
(defun arch-rj (target field &optional fill)
  "Produces a string to skip to column TARGET\
 minus the width of field FIELD.
Prepends newline if needed.
The optional FILL should be a character,
used to fill to the column."
  (arch-tocol (- target (length (nth field forms-fields))) fill))

;; Record filters.
;;
(defun new-record-filter (the-record)
  "Form a new record with some defaults."
  (aset the-record arch-from (user-full-name))
  (aset the-record arch-date (current-time-string))
  the-record)                           ; return it
```


```
(setq forms-new-record-filter 'new-record-filter)

;; The format list.
(setq forms-format-list
     (list
       "====== Public Domain Software Archive ======\n\n"
       arch-shortname
       " - "                    arch-longname
       "\n\n"
       "Article: "              arch-newsgroup
       "/"                      arch-article
       " "
       '(arch-tocol 40)
       "Issue: "                arch-issue
       " "
       '(arch-rj 73 10)
       "Date: "                 arch-date
       "\n\n"
       "Submitted by: "         arch-from
       "\n"
       '(arch-tocol 79 ?-)
       "\n"
       "Keywords: "             arch-keywords
       "\n\n"
       "Parts: "                arch-parts
       "\n\n====== Remarks ======\n\n"
       arch-remarks
     ))

;; That's all, folks!
```

Credits

Forms mode was developed by Johan Vromans while working at Multihouse Research in the Netherlands.

Bug fixes and other useful suggestions were supplied by Richard Stallman (`rms@gnu.ai.mit.edu`), Harald Hanche-Olsen (`hanche@imf.unit.no`), `cwitty@portia.stanford.edu`, Jonathan I. Kamens, Per Cederqvist (`ceder@signum.se`), and Ignatios Souvatzis.

This documentation was slightly inspired by the documentation of "rolo mode" by Paul Davis at Schlumberger Cambridge Research (`davis%scrsu1%sdr.slb.com@relay.cs.net`).

None of this would have been possible without GNU Emacs of the Free Software Foundation. Thanks, Richard!

Index

b

c

e

f

n

p

r

s

- S-Tab

t

- TAB

Flex, version 2.5

A fast scanner generator

Edition 2.5, March 1995

Vern Paxson

Flex, version 2.5

Name

flex - fast lexical analyzer generator

Synopsis

```
flex [-bcdfhilnpstvwBFILTV78+? -C[aefFmr] -ooutput -Pprefix -Sskeleton]
[--help --version] [filename ...]
```

Overview

This manual describes `flex`, a tool for generating programs that perform pattern-matching on text. The manual includes both tutorial and reference sections:

Description
: a brief overview of the tool

Some Simple Examples

Format Of The Input File

Patterns
: the extended regular expressions used by flex

How The Input Is Matched

Description

`flex` is a tool for generating scanners: programs which recognized lexical patterns in text. `flex` reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. `flex` generates as output a C source file, `` `lex.yy.c' ``, which defines a routine `yylex()'. This file is compiled and linked with the `-lfl' library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

Some simple examples

First some simple examples to get the flavor of how one uses `flex`. The following `flex` input specifies a scanner which whenever it encounters the string "username" will replace it with the user's login name:

```
%%
username    printf( "%s", getlogin() );
```

By default, any text not matched by a `flex` scanner is copied to the output, so the net effect of this scanner is to copy its input file to its output with each occurrence of "username" expanded. In this input, there is just one rule. "username" is the pattern and the "printf" is the action. The "%%" marks the beginning of the rules.

Here's another simple example:

```
        int num_lines = 0, num_chars = 0;

%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;

%%
main()
        {
        yylex();
        printf( "# of lines = %d, # of chars = %d\n",
                num_lines, num_chars );
        }
```

This scanner counts the number of characters and the number of lines in its input (it produces no output other than the final report on the counts). The first line declares two globals, "num_lines" and "num_chars", which are accessible both inside `yylex()' and in the `main()' routine declared after the second "%%". There are two rules, one which matches a newline ("\n") and increments both the line count and the character count, and one which matches any character other than a newline (indicated by the "." regular expression).

A somewhat more complicated example:

```
/* scanner for a toy Pascal-like language */

%{
/* need this for the call to atof() below */
#include <math.h>
%}

DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%

{DIGIT}+    {
            printf( "An integer: %s (%d)\n", yytext,
                    atoi( yytext ) );
            }

{DIGIT}+"."{DIGIT}*         {
            printf( "A float: %s (%g)\n", yytext,
                    atof( yytext ) );
            }

if|then|begin|end|procedure|function        {
            printf( "A keyword: %s\n", yytext );
            }

{ID}        printf( "An identifier: %s\n", yytext );

"+"|"-"|"*"|"/"   printf( "An operator: %s\n", yytext );

"{"[^}\n]*"}"     /* eat up one-line comments */

[ \t\n]+          /* eat up whitespace */

.           printf( "Unrecognized character: %s\n", yytext );

%%

main( argc, argv )
int argc;
char **argv;
    {
    ++argv, --argc;  /* skip over program name */
    if ( argc > 0 )
```


```
            yyin = fopen( argv[0], "r" );
    else
            yyin = stdin;

    yylex();
    }
```

This is the beginnings of a simple scanner for a language like Pascal. It identifies different types of tokens and reports on what it has seen.

The details of this example will be explained in the following sections.

Format of the input file

The `flex` input file consists of three sections, separated by a line with just `%%' in it:

```
definitions
%%
rules
%%
user code
```

The definitions section contains declarations of simple name definitions to simplify the scanner specification, and declarations of start conditions, which are explained in a later section. Name definitions have the form:

```
name definition
```

The "name" is a word beginning with a letter or an underscore ('_') followed by zero or more letters, digits, '_', or '-' (dash). The definition is taken to begin at the first non-white-space character following the name and continuing to the end of the line. The definition can subsequently be referred to using "{name}", which will expand to "(definition)". For example,

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

defines "DIGIT" to be a regular expression which matches a single digit, and "ID" to be a regular expression which matches a letter followed by zero-or-more letters-or-digits. A subsequent reference to

```
{DIGIT}+"."{DIGIT}*
```

is identical to

```
([0-9])+"."([0-9])*
```

and matches one-or-more digits followed by a '.' followed by zero-or-more digits.

The rules section of the `flex` input contains a series of rules of the form:

```
pattern   action
```

where the pattern must be unindented and the action must begin on the same line.

See below for a further description of patterns and actions.

Finally, the user code section is simply copied to `` `lex.yy.c' `` verbatim. It is used for companion routines which call or are called by the scanner. The presence of this section is optional; if it is missing, the second `%%' in the input file may be skipped, too.

In the definitions and rules sections, any *indented* text or text enclosed in `%{' and `%}' is copied verbatim to the output (with the `%{}"s removed). The `%{}"s must appear unindented on lines by themselves.

In the rules section, any indented or %{} text appearing before the first rule may be used to declare variables which are local to the scanning routine and (after the declarations) code which is to be executed whenever the scanning routine is entered. Other indented or %{} text in the rule section is still copied to the output, but its meaning is not well-defined and it may well cause compile-time errors (this feature is present for `POSIX` compliance; see below for other such features).

In the definitions section (but not in the rules section), an unindented comment (i.e., a line beginning with "/*") is also copied verbatim to the output up to the next "*/".

Patterns

The patterns in the input are written using an extended set of regular expressions. These are:

`x'
: match the character `x'

`.'
: any character (byte) except newline

`[xyz]'
: a "character class"; in this case, the pattern matches either an `x', a `y', or a `z'

`[abj-oZ]'
: a "character class" with a range in it; matches an `a', a `b', any letter from `j' through `o', or a `Z'

`[^A-Z]'
: a "negated character class", i.e., any character but those in the class. In this case, any character EXCEPT an uppercase letter.

`[^A-Z\n]'
: any character EXCEPT an uppercase letter or a newline

`r*'
: zero or more r's, where r is any regular expression

`r+'
: one or more r's

`r?'
: zero or one r's (that is, "an optional r")

`r{2,5}'
: anywhere from two to five r's

`r{2,}'
: two or more r's

`r{4}'
: exactly 4 r's

`{name}'
: the expansion of the "name" definition (see above)

`"[xyz]\"foo"'
: the literal string: `[xyz]"foo'

`\x'
: if x is an `a', `b', `f', `n', `r', `t', or `v', then the ANSI-C interpretation of \x. Otherwise, a literal `x' (used to escape operators such as `*')

`\0'
: a NUL character (ASCII code 0)

`\123'
: the character with octal value 123

`\x2a'
: the character with hexadecimal value `2a`

`(r)'
: match an r; parentheses are used to override precedence (see below)

`rs'
: the regular expression r followed by the regular expression s; called "concatenation"

`r|s'
: either an r or an s

`r/s'
: an r but only if it is followed by an s. The text matched by s is included when determining whether this rule is the longest match, but is then returned to the input before the action is executed. So the action only sees the text matched by r. This type of pattern is called trailing context. (There are some combinations of `r/s' that `flex` cannot match correctly; see notes in the Deficiencies / Bugs section below regarding "dangerous trailing context".)

`^r'
: an r, but only at the beginning of a line (i.e., which just starting to scan, or right after a newline has been scanned).

`r$'

an r, but only at the end of a line (i.e., just before a newline). Equivalent to "r/\n".

Note that flex's notion of "newline" is exactly whatever the C compiler used to compile flex interprets '\n' as; in particular, on some DOS systems you must either filter out \r's in the input yourself, or explicitly use r/\r\n for "r$".

`<s>r'

an r, but only in start condition s (see below for discussion of start conditions) <s1,s2,s3>r same, but in any of start conditions s1, s2, or s3

`<*>r'

an r in any start condition, even an exclusive one.

`<<EOF>>'

an end-of-file <s1,s2><<EOF>> an end-of-file when in start condition s1 or s2

Note that inside of a character class, all regular expression operators lose their special meaning except escape ('\') and the character class operators, '-', ']', and, at the beginning of the class, '^'.

The regular expressions listed above are grouped according to precedence, from highest precedence at the top to lowest at the bottom. Those grouped together have equal precedence. For example,

```
foo|bar*
```

is the same as

```
(foo)|(ba(r*))
```

since the '*' operator has higher precedence than concatenation, and concatenation higher than alternation ('|'). This pattern therefore matches *either* the string "foo" *or* the string "ba" followed by zero-or-more r's. To match "foo" or zero-or-more "bar"'s, use:

```
foo|(bar)*
```

and to match zero-or-more "foo"'s-or-"bar"'s:

```
(foo|bar)*
```

In addition to characters and ranges of characters, character classes can also contain character class expressions. These are expressions enclosed inside `[': and `:'] delimiters (which themselves must appear between the '[' and ']' of the character class; other elements may occur inside the character class, too). The valid expressions are:

```
[:alnum:] [:alpha:] [:blank:]
[:cntrl:] [:digit:] [:graph:]
[:lower:] [:print:] [:punct:]
[:space:] [:upper:] [:xdigit:]
```

These expressions all designate a set of characters equivalent to the corresponding standard C `isXXX' function. For example, `[:alnum:]' designates those characters for which `isalnum()' returns true - i.e., alphabetic or numeric. Some systems don't provide `isblank()', so flex defines `[:blank:]' as a blank or a tab.

For example, the following character classes are all equivalent:

```
[[:alnum:]]
[[:alpha:][:digit:]
[[:alpha:]0-9]
[a-zA-Z0-9]
```

If your scanner is case-insensitive (the `-i' flag), then `[:upper:]' and `[:lower:]' are equivalent to `[:alpha:]'.

Some notes on patterns:

- A negated character class such as the example "[^A-Z]" above *will match a newline* unless "\n" (or an equivalent escape sequence) is one of the characters explicitly present in the negated character class (e.g., "[^A-Z\n]"). This is unlike how many other regular expression tools treat negated character classes, but unfortunately the inconsistency is historically entrenched. Matching newlines means that a pattern like [^"]* can match the entire input unless there's another quote in the input.
- A rule can have at most one instance of trailing context (the '/' operator or the '$' operator). The start condition, '^', and "<<EOF>>" patterns can only occur at the beginning of a pattern, and, as well as with '/' and '$', cannot be grouped inside parentheses. A '^' which does not occur at the beginning of a rule or a '$' which does not occur at the end of a rule loses its special properties and is treated as a normal character.

  The following are illegal:

  ```
  foo/bar$
  <sc1>foo<sc2>bar
  ```

  Note that the first of these, can be written "foo/bar\n".

  The following will result in '$' or '^' being treated as a normal character:

  ```
  foo|(bar$)
  foo|^bar
  ```

  If what's wanted is a "foo" or a bar-followed-by-a-newline, the following could be used (the special '|' action is explained below):

  ```
  foo      |
  bar$     /* action goes here */
  ```

  A similar trick will work for matching a foo or a bar-at-the-beginning-of-a-line.

How the input is matched

When the generated scanner is run, it analyzes its input looking for strings which match any of its patterns. If it finds more than one match, it takes the one matching the most text (for trailing context rules, this includes the length of the trailing part, even though it will then be returned to the input). If it finds two or more matches of the same length, the rule listed first in the `flex` input file is chosen.

Once the match is determined, the text corresponding to the match (called the token) is made available in the global character pointer `yytext`, and its length in the global integer `yyleng`. The action corresponding to the matched pattern is then executed (a more detailed description of actions follows), and then the remaining input is scanned for another match.

If no match is found, then the default rule is executed: the next character in the input is considered matched and copied to the standard output. Thus, the simplest legal `flex` input is:

```
%%
```

which generates a scanner that simply copies its input (one character at a time) to its output.

Note that `yytext` can be defined in two different ways: either as a character *pointer* or as a character *array*. You can control which definition `flex` uses by including one of the special directives `%pointer' or `%array' in the first (definitions) section of your flex input. The default is `%pointer', unless you use the `-l' lex compatibility option, in which case `yytext` will be an array. The advantage of using `%pointer' is substantially faster scanning and no buffer overflow when matching very large tokens (unless you run out of dynamic memory). The disadvantage is that you are restricted in how your actions can modify `yytext` (see the next section), and calls to the `unput()' function destroys the present contents of `yytext`, which can be a considerable porting headache when moving between different `lex` versions.

The advantage of `%array' is that you can then modify `yytext` to your heart's content, and calls to `unput()' do not destroy `yytext` (see below). Furthermore, existing `lex` programs sometimes access `yytext` externally using declarations of the form:

```
extern char yytext[];
```

This definition is erroneous when used with `%pointer', but correct for `%array'.

`%array' defines `yytext` to be an array of `YYLMAX` characters, which defaults to a fairly large value. You can change the size by simply #define'ing `YYLMAX` to a different value in the first section of your `flex` input. As mentioned above, with `%pointer' yytext grows dynamically to accommodate large tokens. While this means your `%pointer' scanner can accommodate very large tokens (such as matching entire blocks of comments), bear in mind that each time the scanner must resize `yytext` it also must rescan the entire token from the beginning, so matching such tokens can prove slow. `yytext` presently does *not* dynamically grow if a call to `unput()' results in too much text being pushed back; instead, a run-time error results.

Also note that you cannot use `%array' with C++ scanner classes (the `c++` option; see below).

Actions

Each pattern in a rule has a corresponding action, which can be any arbitrary C statement. The pattern ends at the first non-escaped whitespace character; the remainder of the line is its action. If the action is empty, then when the pattern is matched the input token is simply discarded. For example, here is the specification for a program which deletes all occurrences of "zap me" from its input:

```
%%
"zap me"
```

(It will copy all other characters in the input to the output since they will be matched by the default rule.)

Here is a program which compresses multiple blanks and tabs down to a single blank, and throws away whitespace found at the end of a line:

```
%%
[ \t]+        putchar( ' ' );
[ \t]+$       /* ignore this token */
```

If the action contains a '{', then the action spans till the balancing '}' is found, and the action may cross multiple lines. `flex` knows about C strings and comments and won't be fooled by braces found within them, but also allows actions to begin with `%{' and will consider the action to be all the text up to the next `%}' (regardless of ordinary braces inside the action).

An action consisting solely of a vertical bar ('|') means "same as the action for the next rule." See below for an illustration.

Actions can include arbitrary C code, including `return` statements to return a value to whatever routine called `yylex()'. Each time `yylex()' is called it continues processing tokens from where it last left off until it either reaches the end of the file or executes a return.

Actions are free to modify `yytext` except for lengthening it (adding characters to its end--these will overwrite later characters in the input stream). This however does not apply when using `%array' (see above); in that case, `yytext` may be freely modified in any way.

Actions are free to modify `yyleng` except they should not do so if the action also includes use of `yymore()' (see below).

There are a number of special directives which can be included within an action:

- `ECHO' copies yytext to the scanner's output.
- `BEGIN` followed by the name of a start condition places the scanner in the corresponding start condition (see below).
- `REJECT` directs the scanner to proceed on to the "second best" rule which matched the input (or a prefix of the input). The rule is chosen as described above in "How the Input is Matched", and `yytext` and `yyleng` set up appropriately. It may either be one which matched as much text as the originally chosen rule but came later in the `flex` input file, or one which matched less text. For example, the following will both count the words in the input and call the routine special()

whenever "frob" is seen:

```
        int word_count = 0;
%%

frob        special(); REJECT;
[^ \t\n]+   ++word_count;
```

Without the `REJECT`, any "frob"'s in the input would not be counted as words, since the scanner normally executes only one action per token. Multiple `REJECT's` are allowed, each one finding the next best choice to the currently active rule. For example, when the following scanner scans the token "abcd", it will write "abcdabcaba" to the output:

```
%%
a           |
ab          |
abc         |
abcd        ECHO; REJECT;
.|\n        /* eat up any unmatched character */
```

(The first three rules share the fourth's action since they use the special '|' action.) `REJECT` is a particularly expensive feature in terms of scanner performance; if it is used in *any* of the scanner's actions it will slow down *all* of the scanner's matching. Furthermore, `REJECT` cannot be used with the `-Cf' or `-CF' options (see below).

Note also that unlike the other special actions, `REJECT` is a *branch*; code immediately following it in the action will *not* be executed.

- `yymore()' tells the scanner that the next time it matches a rule, the corresponding token should be *appended* onto the current value of `yytext` rather than replacing it. For example, given the input "mega-kludge" the following will write "mega-mega-kludge" to the output:

  ```
  %%
  mega-    ECHO; yymore();
  kludge   ECHO;
  ```

  First "mega-" is matched and echoed to the output. Then "kludge" is matched, but the previous "mega-" is still hanging around at the beginning of `yytext` so the `ECHO' for the "kludge" rule will actually write "mega-kludge".

Two notes regarding use of `yymore()'. First, `yymore()' depends on the value of `yyleng` correctly reflecting the size of the current token, so you must not modify `yyleng` if you are using `yymore()'. Second, the presence of `yymore()' in the scanner's action entails a minor performance penalty in the scanner's matching speed.

- `yyless(n)' returns all but the first n characters of the current token back to the input stream, where they will be rescanned when the scanner looks for the next match. `yytext` and `yyleng` are adjusted appropriately (e.g., `yyleng` will now be equal to n). For example, on the input "foobar"

the following will write out "foobarbar":

```
%%
foobar    ECHO; yyless(3);
[a-z]+    ECHO;
```

An argument of 0 to `yyless` will cause the entire current input string to be scanned again. Unless you've changed how the scanner will subsequently process its input (using `BEGIN`, for example), this will result in an endless loop.

Note that `yyless` is a macro and can only be used in the flex input file, not from other source files.

- `unput(c)' puts the character `c` back onto the input stream. It will be the next character scanned. The following action will take the current token and cause it to be rescanned enclosed in parentheses.

```
{
int i;
/* Copy yytext because unput() trashes yytext */
char *yycopy = strdup( yytext );
unput( ')' );
for ( i = yyleng - 1; i >= 0; --i )
    unput( yycopy[i] );
unput( '(' );
free( yycopy );
}
```

Note that since each `unput()' puts the given character back at the *beginning* of the input stream, pushing back strings must be done back-to-front. An important potential problem when using `unput()' is that if you are using `%pointer' (the default), a call to `unput()' *destroys* the contents of `yytext`, starting with its rightmost character and devouring one character to the left with each call. If you need the value of yytext preserved after a call to `unput()' (as in the above example), you must either first copy it elsewhere, or build your scanner using `%array' instead (see How The Input Is Matched).

Finally, note that you cannot put back `EOF` to attempt to mark the input stream with an end-of-file.

- `input()' reads the next character from the input stream. For example, the following is one way to eat up C comments:

```
%%
"/*"        {
            register int c;

            for ( ; ; )
                {
                while ( (c = input()) != '*' &&
                        c != EOF )
```


```
            ;    /* eat up text of comment */

        if ( c == '*' )
            {
            while ( (c = input()) == '*' )
                ;
            if ( c == '/' )
                break;    /* found the end */
            }

        if ( c == EOF )
            {
            error( "EOF in comment" );
            break;
            }
        }
    }
```

(Note that if the scanner is compiled using `C++', then `input()' is instead referred to as `yyinput()', in order to avoid a name clash with the `C++' stream by the name of `input'.)

- YY_FLUSH_BUFFER flushes the scanner's internal buffer so that the next time the scanner attempts to match a token, it will first refill the buffer using `YY_INPUT' (see The Generated Scanner, below). This action is a special case of the more general `yy_flush_buffer()' function, described below in the section Multiple Input Buffers.
- `yyterminate()' can be used in lieu of a return statement in an action. It terminates the scanner and returns a 0 to the scanner's caller, indicating "all done". By default, `yyterminate()' is also called when an end-of-file is encountered. It is a macro and may be redefined.

The generated scanner

The output of `flex' is the file `lex.yy.c', which contains the scanning routine `yylex()', a number of tables used by it for matching tokens, and a number of auxiliary routines and macros. By default, `yylex()' is declared as follows:

```
int yylex()
    {
    ... various definitions and the actions in here ...
    }
```

(If your environment supports function prototypes, then it will be "int yylex(void)".) This definition may be changed by defining the "YY_DECL" macro. For example, you could use:

```
#define YY_DECL float lexscan( a, b ) float a, b;
```

to give the scanning routine the name `lexscan', returning a float, and taking two floats as arguments.

Note that if you give arguments to the scanning routine using a K&R-style/non-prototyped function declaration, you must terminate the definition with a semi-colon (`;').

Whenever `yylex()' is called, it scans tokens from the global input file `yyin` (which defaults to stdin). It continues until it either reaches an end-of-file (at which point it returns the value 0) or one of its actions executes a `return` statement.

If the scanner reaches an end-of-file, subsequent calls are undefined unless either `yyin` is pointed at a new input file (in which case scanning continues from that file), or `yyrestart()' is called. `yyrestart()' takes one argument, a `FILE *' pointer (which can be nil, if you've set up `YY_INPUT` to scan from a source other than `yyin`), and initializes `yyin` for scanning from that file. Essentially there is no difference between just assigning `yyin` to a new input file or using `yyrestart()' to do so; the latter is available for compatibility with previous versions of `flex`, and because it can be used to switch input files in the middle of scanning. It can also be used to throw away the current input buffer, by calling it with an argument of `yyin`; but better is to use `YY_FLUSH_BUFFER` (see above). Note that `yyrestart()' does *not* reset the start condition to `INITIAL` (see Start Conditions, below).

If `yylex()' stops scanning due to executing a `return` statement in one of the actions, the scanner may then be called again and it will resume scanning where it left off.

By default (and for purposes of efficiency), the scanner uses block-reads rather than simple `getc()' calls to read characters from `yyin`. The nature of how it gets its input can be controlled by defining the `YY_INPUT` macro. YY_INPUT's calling sequence is "YY_INPUT(buf,result,max_size)". Its action is to place up to max_size characters in the character array buf and return in the integer variable result either the number of characters read or the constant YY_NULL (0 on Unix systems) to indicate EOF. The default YY_INPUT reads from the global file-pointer "yyin".

A sample definition of YY_INPUT (in the definitions section of the input file):

```
%{
#define YY_INPUT(buf,result,max_size) \
    { \
    int c = getchar(); \
    result = (c == EOF) ? YY_NULL : (buf[0] = c, 1); \
    }
%}
```

This definition will change the input processing to occur one character at a time.

When the scanner receives an end-of-file indication from YY_INPUT, it then checks the `yywrap()' function. If `yywrap()' returns false (zero), then it is assumed that the function has gone ahead and set up `yyin` to point to another input file, and scanning continues. If it returns true (non-zero), then the scanner terminates, returning 0 to its caller. Note that in either case, the start condition remains unchanged; it does *not* revert to `INITIAL`.

If you do not supply your own version of `yywrap()', then you must either use `%option noyywrap' (in which case the scanner behaves as though `yywrap()' returned 1), or you must link with `-lfl' to obtain the default version of the routine, which always returns 1.

Three routines are available for scanning from in-memory buffers rather than files: `yy_scan_string()', `yy_scan_bytes()', and `yy_scan_buffer()'. See the discussion of them below in the section Multiple Input Buffers.

The scanner writes its `ECHO' output to the `yyout` global (default, stdout), which may be redefined by the user simply by assigning it to some other `FILE` pointer.

Start conditions

`flex` provides a mechanism for conditionally activating rules. Any rule whose pattern is prefixed with "<sc>" will only be active when the scanner is in the start condition named "sc". For example,

```
<STRING>[^"]*        { /* eat up the string body ... */
             ...
             }
```

will be active only when the scanner is in the "STRING" start condition, and

```
<INITIAL,STRING,QUOTE>\.        { /* handle an escape ... */
             ...
             }
```

will be active only when the current start condition is either "INITIAL", "STRING", or "QUOTE".

Start conditions are declared in the definitions (first) section of the input using unindented lines beginning with either `%s' or `%x' followed by a list of names. The former declares *inclusive* start conditions, the latter *exclusive* start conditions. A start condition is activated using the `BEGIN` action. Until the next `BEGIN` action is executed, rules with the given start condition will be active and rules with other start conditions will be inactive. If the start condition is *inclusive*, then rules with no start conditions at all will also be active. If it is *exclusive*, then *only* rules qualified with the start condition will be active. A set of rules contingent on the same exclusive start condition describe a scanner which is independent of any of the other rules in the `flex` input. Because of this, exclusive start conditions make it easy to specify "mini-scanners" which scan portions of the input that are syntactically different from the rest (e.g., comments).

If the distinction between inclusive and exclusive start conditions is still a little vague, here's a simple example illustrating the connection between the two. The set of rules:

```
%s example
%%

<example>foo   do_something();

bar            something_else();
```

is equivalent to


```
%x example
%%

<example>foo   do_something();

<INITIAL,example>bar    something_else();
```

Without the `<INITIAL,example>' qualifier, the `bar' pattern in the second example wouldn't be active (i.e., couldn't match) when in start condition `example'. If we just used `<example>' to qualify `bar', though, then it would only be active in `example' and not in `INITIAL', while in the first example it's active in both, because in the first example the `example' starting condition is an *inclusive* (`%s') start condition.

Also note that the special start-condition specifier `<*>' matches every start condition. Thus, the above example could also have been written;

```
%x example
%%

<example>foo   do_something();

<*>bar    something_else();
```

The default rule (to `ECHO' any unmatched character) remains active in start conditions. It is equivalent to:

```
<*>.|\\n     ECHO;
```

`BEGIN(0)' returns to the original state where only the rules with no start conditions are active. This state can also be referred to as the start-condition "INITIAL", so `BEGIN(INITIAL)' is equivalent to `BEGIN(0)'. (The parentheses around the start condition name are not required but are considered good style.)

`BEGIN` actions can also be given as indented code at the beginning of the rules section. For example, the following will cause the scanner to enter the "SPECIAL" start condition whenever `yylex()' is called and the global variable `enter_special` is true:

```
        int enter_special;

%x SPECIAL
%%
        if ( enter_special )
            BEGIN(SPECIAL);

<SPECIAL>blahblahblah
...more rules follow...
```

To illustrate the uses of start conditions, here is a scanner which provides two different interpretations of a string like "123.456". By default it will treat it as as three tokens, the integer "123", a dot ('.'), and the integer "456". But if the string is preceded earlier in the line by the string "expect-floats" it will treat it as a single token, the floating-point number 123.456:

```
%{
#include <math.h>
%}
%s expect

%%
expect-floats        BEGIN(expect);

<expect>[0-9]+"."[0-9]+      {
            printf( "found a float, = %f\n",
                    atof( yytext ) );
            }
<expect>\n           {
            /* that's the end of the line, so
             * we need another "expect-number"
             * before we'll recognize any more
             * numbers
             */
            BEGIN(INITIAL);
            }

[0-9]+       {

Version 2.5                 December 1994                          18

            printf( "found an integer, = %d\n",
                    atoi( yytext ) );
            }

"."          printf( "found a dot\n" );
```

Here is a scanner which recognizes (and discards) C comments while maintaining a count of the current input line.

```
%x comment
%%
        int line_num = 1;

"/*"         BEGIN(comment);
```


```
<comment>[^*\n]*        /* eat anything that's not a '*' */
<comment>"*"+[^*/\n]*   /* eat up '*'s not followed by '/'s */
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(INITIAL);
```

This scanner goes to a bit of trouble to match as much text as possible with each rule. In general, when attempting to write a high-speed scanner try to match as much possible in each rule, as it's a big win.

Note that start-conditions names are really integer values and can be stored as such. Thus, the above could be extended in the following fashion:

```
%x comment foo
%%
        int line_num = 1;
        int comment_caller;

"/*"         {
             comment_caller = INITIAL;
             BEGIN(comment);
             }

...

<foo>"/*"    {
             comment_caller = foo;
             BEGIN(comment);
             }

<comment>[^*\n]*        /* eat anything that's not a '*' */
<comment>"*"+[^*/\n]*   /* eat up '*'s not followed by '/'s */
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(comment_caller);
```

Furthermore, you can access the current start condition using the integer-valued `YY_START` macro. For example, the above assignments to `comment_caller` could instead be written

```
comment_caller = YY_START;
```

Flex provides `YYSTATE` as an alias for `YY_START` (since that is what's used by AT&T `lex`).

Note that start conditions do not have their own name-space; %s's and %x's declare names in the same fashion as #define's.

Finally, here's an example of how to match C-style quoted strings using exclusive start conditions, including expanded escape sequences (but not including checking for a string that's too long):

```
%x str
```


```
%%
        char string_buf[MAX_STR_CONST];
        char *string_buf_ptr;

\"      string_buf_ptr = string_buf; BEGIN(str);

<str>\"        { /* saw closing quote - all done */
        BEGIN(INITIAL);
        *string_buf_ptr = '\0';
        /* return string constant token type and
         * value to parser
         */
        }

<str>\n        {
        /* error - unterminated string constant */
        /* generate error message */
        }

<str>\\[0-7]{1,3} {
        /* octal escape sequence */
        int result;

        (void) sscanf( yytext + 1, "%o", &result );

        if ( result > 0xff )
                /* error, constant is out-of-bounds */

        *string_buf_ptr++ = result;
        }

<str>\\[0-9]+ {
        /* generate error - bad escape sequence; something
         * like '\48' or '\0777777'
         */
        }

<str>\\n  *string_buf_ptr++ = '\n';
<str>\\t  *string_buf_ptr++ = '\t';
<str>\\r  *string_buf_ptr++ = '\r';
<str>\\b  *string_buf_ptr++ = '\b';
<str>\\f  *string_buf_ptr++ = '\f';

<str>\\(.|\n)  *string_buf_ptr++ = yytext[1];
```


```
<str>[^\\\n\"]+        {
        char *yptr = yytext;

        while ( *yptr )
                *string_buf_ptr++ = *yptr++;
        }
```

Often, such as in some of the examples above, you wind up writing a whole bunch of rules all preceded by the same start condition(s). Flex makes this a little easier and cleaner by introducing a notion of start condition scope. A start condition scope is begun with:

```
<SCs>{
```

where SCs is a list of one or more start conditions. Inside the start condition scope, every rule automatically has the prefix `<SCs>' applied to it, until a `}' which matches the initial `{'. So, for example,

```
<ESC>{
    "\\n"   return '\n';
    "\\r"   return '\r';
    "\\f"   return '\f';
    "\\0"   return '\0';
}
```

is equivalent to:

```
<ESC>"\\n"  return '\n';
<ESC>"\\r"  return '\r';
<ESC>"\\f"  return '\f';
<ESC>"\\0"  return '\0';
```

Start condition scopes may be nested.

Three routines are available for manipulating stacks of start conditions:

`void yy_push_state(int new_state)'

pushes the current start condition onto the top of the start condition stack and switches to new_state as though you had used `BEGIN new_state' (recall that start condition names are also integers).

`void yy_pop_state()'

pops the top of the stack and switches to it via `BEGIN`.

`int yy_top_state()'

returns the top of the stack without altering the stack's contents.

The start condition stack grows dynamically and so has no built-in size limitation. If memory is exhausted, program execution aborts.

To use start condition stacks, your scanner must include a `%option stack' directive (see Options below).

Multiple input buffers

Some scanners (such as those which support "include" files) require reading from several input streams. As `flex` scanners do a large amount of buffering, one cannot control where the next input will be read from by simply writing a `YY_INPUT` which is sensitive to the scanning context. `YY_INPUT` is only called when the scanner reaches the end of its buffer, which may be a long time after scanning a statement such as an "include" which requires switching the input source.

To negotiate these sorts of problems, `flex` provides a mechanism for creating and switching between multiple input buffers. An input buffer is created by using:

```
YY_BUFFER_STATE yy_create_buffer( FILE *file, int size )
```

which takes a `FILE` pointer and a size and creates a buffer associated with the given file and large enough to hold size characters (when in doubt, use `YY_BUF_SIZE` for the size). It returns a `YY_BUFFER_STATE` handle, which may then be passed to other routines (see below). The `YY_BUFFER_STATE` type is a pointer to an opaque `struct yy_buffer_state` structure, so you may safely initialize YY_BUFFER_STATE variables to `((YY_BUFFER_STATE) 0)' if you wish, and also refer to the opaque structure in order to correctly declare input buffers in source files other than that of your scanner. Note that the `FILE` pointer in the call to `yy_create_buffer` is only used as the value of `yyin` seen by `YY_INPUT`; if you redefine `YY_INPUT` so it no longer uses `yyin`, then you can safely pass a nil `FILE` pointer to `yy_create_buffer`. You select a particular buffer to scan from using:

```
void yy_switch_to_buffer( YY_BUFFER_STATE new_buffer )
```

switches the scanner's input buffer so subsequent tokens will come from new_buffer. Note that `yy_switch_to_buffer()' may be used by `yywrap()' to set things up for continued scanning, instead of opening a new file and pointing `yyin` at it. Note also that switching input sources via either `yy_switch_to_buffer()' or `yywrap()' does *not* change the start condition.

```
void yy_delete_buffer( YY_BUFFER_STATE buffer )
```

is used to reclaim the storage associated with a buffer. You can also clear the current contents of a buffer using:

```
void yy_flush_buffer( YY_BUFFER_STATE buffer )
```

This function discards the buffer's contents, so the next time the scanner attempts to match a token from the buffer, it will first fill the buffer anew using `YY_INPUT`.

`yy_new_buffer()' is an alias for `yy_create_buffer()', provided for compatibility with the C++ use of `new` and `delete` for creating and destroying dynamic objects.

Finally, the `YY_CURRENT_BUFFER` macro returns a `YY_BUFFER_STATE` handle to the current buffer.

Here is an example of using these features for writing a scanner which expands include files (the

`<<EOF>>' feature is discussed below):

```
/* the "incl" state is used for picking up the name
 * of an include file
 */
%x incl

%{
#define MAX_INCLUDE_DEPTH 10
YY_BUFFER_STATE include_stack[MAX_INCLUDE_DEPTH];
int include_stack_ptr = 0;
%}

%%
include             BEGIN(incl);

[a-z]+              ECHO;
[^a-z\n]*\n?        ECHO;

<incl>[ \t]*      /* eat the whitespace */
<incl>[^ \t\n]+   { /* got the include file name */
        if ( include_stack_ptr >= MAX_INCLUDE_DEPTH )
            {
            fprintf( stderr, "Includes nested too deeply" );
            exit( 1 );
            }

        include_stack[include_stack_ptr++] =
            YY_CURRENT_BUFFER;

        yyin = fopen( yytext, "r" );

        if ( ! yyin )
            error( ... );

        yy_switch_to_buffer(
            yy_create_buffer( yyin, YY_BUF_SIZE ) );

        BEGIN(INITIAL);
        }

<<EOF>> {
        if ( --include_stack_ptr < 0 )
            {
            yyterminate();
            }
```


```
        else
            {
            yy_delete_buffer( YY_CURRENT_BUFFER );
            yy_switch_to_buffer(
                 include_stack[include_stack_ptr] );
            }
        }
```

Three routines are available for setting up input buffers for scanning in-memory strings instead of files. All of them create a new input buffer for scanning the string, and return a corresponding YY_BUFFER_STATE handle (which you should delete with `yy_delete_buffer()' when done with it). They also switch to the new buffer using `yy_switch_to_buffer()', so the next call to `yylex()' will start scanning the string.

`yy_scan_string(const char *str)'

scans a NUL-terminated string.

`yy_scan_bytes(const char *bytes, int len)'

scans `len` bytes (including possibly NUL's) starting at location bytes.

Note that both of these functions create and scan a *copy* of the string or bytes. (This may be desirable, since `yylex()' modifies the contents of the buffer it is scanning.) You can avoid the copy by using:

`yy_scan_buffer(char *base, yy_size_t size)'

which scans in place the buffer starting at base, consisting of size bytes, the last two bytes of which *must* be YY_END_OF_BUFFER_CHAR (ASCII NUL). These last two bytes are not scanned; thus, scanning consists of `base[0]' through `base[size-2]', inclusive.

If you fail to set up base in this manner (i.e., forget the final two YY_END_OF_BUFFER_CHAR bytes), then `yy_scan_buffer()' returns a nil pointer instead of creating a new input buffer.

The type `yy_size_t` is an integral type to which you can cast an integer expression reflecting the size of the buffer.

End-of-file rules

The special rule "<<EOF>>" indicates actions which are to be taken when an end-of-file is encountered and yywrap() returns non-zero (i.e., indicates no further files to process). The action must finish by doing one of four things:

- assigning `yyin` to a new input file (in previous versions of flex, after doing the assignment you had to call the special action YY_NEW_FILE; this is no longer necessary);
- executing a `return` statement;
- executing the special `yyterminate()' action;
- or, switching to a new buffer using `yy_switch_to_buffer()' as shown in the example above.

<<EOF>> rules may not be used with other patterns; they may only be qualified with a list of start

conditions. If an unqualified <<EOF>> rule is given, it applies to *all* start conditions which do not already have <<EOF>> actions. To specify an <<EOF>> rule for only the initial start condition, use

```
<INITIAL><<EOF>>
```

These rules are useful for catching things like unclosed comments. An example:

```
%x quote
%%

...other rules for dealing with quotes...

<quote><<EOF>>   {
         error( "unterminated quote" );
         yyterminate();
         }
<<EOF>>  {
         if ( *++filelist )
             yyin = fopen( *filelist, "r" );
         else
            yyterminate();
         }
```

Miscellaneous macros

The macro `YY_USER_ACTION` can be defined to provide an action which is always executed prior to the matched rule's action. For example, it could be #define'd to call a routine to convert yytext to lower-case. When `YY_USER_ACTION` is invoked, the variable `yy_act` gives the number of the matched rule (rules are numbered starting with 1). Suppose you want to profile how often each of your rules is matched. The following would do the trick:

```
#define YY_USER_ACTION ++ctr[yy_act]
```

where `ctr` is an array to hold the counts for the different rules. Note that the macro `YY_NUM_RULES` gives the total number of rules (including the default rule, even if you use `-s', so a correct declaration for `ctr` is:

```
int ctr[YY_NUM_RULES];
```

The macro `YY_USER_INIT` may be defined to provide an action which is always executed before the first scan (and before the scanner's internal initializations are done). For example, it could be used to call a routine to read in a data table or open a logging file.

The macro `yy_set_interactive(is_interactive)' can be used to control whether the current buffer is considered *interactive*. An interactive buffer is processed more slowly, but must be used when the scanner's input source is indeed interactive to avoid problems due to waiting to fill buffers (see the

discussion of the `-I' flag below). A non-zero value in the macro invocation marks the buffer as interactive, a zero value as non-interactive. Note that use of this macro overrides `%option always-interactive' or `%option never-interactive' (see Options below). `yy_set_interactive()' must be invoked prior to beginning to scan the buffer that is (or is not) to be considered interactive.

The macro `yy_set_bol(at_bol)' can be used to control whether the current buffer's scanning context for the next token match is done as though at the beginning of a line. A non-zero macro argument makes rules anchored with

The macro `YY_AT_BOL()' returns true if the next token scanned from the current buffer will have '^' rules active, false otherwise.

In the generated scanner, the actions are all gathered in one large switch statement and separated using YY_BREAK, which may be redefined. By default, it is simply a "break", to separate each rule's action from the following rule's. Redefining YY_BREAK allows, for example, C++ users to #define YY_BREAK to do nothing (while being very careful that every rule ends with a "break" or a "return"!) to avoid suffering from unreachable statement warnings where because a rule's action ends with "return", the YY_BREAK is inaccessible.

Values available to the user

This section summarizes the various values available to the user in the rule actions.

- `char *yytext' holds the text of the current token. It may be modified but not lengthened (you cannot append characters to the end).

  If the special directive `%array' appears in the first section of the scanner description, then `yytext` is instead declared `char yytext[YYLMAX]', where YYLMAX is a macro definition that you can redefine in the first section if you don't like the default value (generally 8KB). Using `%array' results in somewhat slower scanners, but the value of `yytext` becomes immune to calls to `input()' and `unput()', which potentially destroy its value when `yytext` is a character pointer. The opposite of `%array' is `%pointer', which is the default.

  You cannot use `%array' when generating C++ scanner classes (the `-+' flag).
- `int yyleng' holds the length of the current token.
- `FILE *yyin' is the file which by default `flex` reads from. It may be redefined but doing so only makes sense before scanning begins or after an EOF has been encountered. Changing it in the midst of scanning will have unexpected results since `flex` buffers its input; use `yyrestart()' instead. Once scanning terminates because an end-of-file has been seen, you can assign `yyin` at the new input file and then call the scanner again to continue scanning.
- `void yyrestart( FILE *new_file )' may be called to point `yyin` at the new input file. The switch-over to the new file is immediate (any previously buffered-up input is lost). Note that calling `yyrestart()' with `yyin` as an argument thus throws away the current input buffer and continues scanning the same input file.
- `FILE *yyout' is the file to which `ECHO' actions are done. It can be reassigned by the user.
- YY_CURRENT_BUFFER returns a YY_BUFFER_STATE handle to the current buffer.

- YY_START returns an integer value corresponding to the current start condition. You can subsequently use this value with BEGIN to return to that start condition.

Interfacing with yacc

One of the main uses of `flex` is as a companion to the `yacc` parser-generator. `yacc` parsers expect to call a routine named `yylex()' to find the next input token. The routine is supposed to return the type of the next token as well as putting any associated value in the global `yylval`. To use `flex` with `yacc`, one specifies the `-d' option to `yacc` to instruct it to generate the file `` `y.tab.h' `` containing definitions of all the `%tokens' appearing in the `yacc` input. This file is then included in the `flex` scanner. For example, if one of the tokens is "TOK_NUMBER", part of the scanner might look like:

```
%{
#include "y.tab.h"
%}

%%

[0-9]+        yylval = atoi( yytext ); return TOK_NUMBER;
```

Options

`flex` has the following options:

`-b'

Generate backing-up information to `` `lex.backup' ``. This is a list of scanner states which require backing up and the input characters on which they do so. By adding rules one can remove backing-up states. If *all* backing-up states are eliminated and `-Cf' or `-CF' is used, the generated scanner will run faster (see the `-p' flag). Only users who wish to squeeze every last cycle out of their scanners need worry about this option. (See the section on Performance Considerations below.)

`-c'

is a do-nothing, deprecated option included for POSIX compliance.

`-d'

makes the generated scanner run in debug mode. Whenever a pattern is recognized and the global `yy_flex_debug` is non-zero (which is the default), the scanner will write to `stderr` a line of the form:

```
--accepting rule at line 53 ("the matched text")
```

The line number refers to the location of the rule in the file defining the scanner (i.e., the file that was fed to flex). Messages are also generated when the scanner backs up, accepts the default rule, reaches the end of its input buffer (or encounters a NUL; at this point, the two look the same as far as the scanner's concerned), or reaches an end-of-file.

`-f'

specifies fast scanner. No table compression is done and stdio is bypassed. The result is large but fast. This option is equivalent to `-Cfr' (see below).

`-h'

generates a "help" summary of `flex's options to `stdout` and then exits. `-?' and `--help' are synonyms for `-h'.

`-i'

instructs `flex` to generate a *case-insensitive* scanner. The case of letters given in the `flex` input patterns will be ignored, and tokens in the input will be matched regardless of case. The matched text given in `yytext` will have the preserved case (i.e., it will not be folded).

`-l'

turns on maximum compatibility with the original AT&T `lex` implementation. Note that this does not mean *full* compatibility. Use of this option costs a considerable amount of performance, and it cannot be used with the `-+, -f, -F, -Cf', or `-CF' options. For details on the compatibilities it provides, see the section "Incompatibilities With Lex And POSIX" below. This option also results in the name `YY_FLEX_LEX_COMPAT` being #define'd in the generated scanner.

`-n'

is another do-nothing, deprecated option included only for POSIX compliance.

`-p'

generates a performance report to stderr. The report consists of comments regarding features of the `flex` input file which will cause a serious loss of performance in the resulting scanner. If you give the flag twice, you will also get comments regarding features that lead to minor performance losses.

Note that the use of `REJECT`, `%option yylineno' and variable trailing context (see the Deficiencies / Bugs section below) entails a substantial performance penalty; use of `yymore()', the `^' operator, and the `-I' flag entail minor performance penalties.

`-s'

causes the default rule (that unmatched scanner input is echoed to `stdout`) to be suppressed. If the scanner encounters input that does not match any of its rules, it aborts with an error. This option is useful for finding holes in a scanner's rule set.

`-t'

instructs `flex` to write the scanner it generates to standard output instead of `lex.yy.c'.

`-v'

specifies that `flex` should write to `stderr` a summary of statistics regarding the scanner it generates. Most of the statistics are meaningless to the casual `flex` user, but the first line identifies the version of `flex` (same as reported by `-V'), and the next line the flags used when generating the scanner, including those that are on by default.

`-w'

suppresses warning messages.

`-B'

instructs `flex` to generate a *batch* scanner, the opposite of *interactive* scanners generated by `-I'

(see below). In general, you use `-B' when you are *certain* that your scanner will never be used interactively, and you want to squeeze a *little* more performance out of it. If your goal is instead to squeeze out a *lot* more performance, you should be using the `-Cf' or `-CF' options (discussed below), which turn on `-B' automatically anyway.

`-F'

specifies that the fast scanner table representation should be used (and stdio bypassed). This representation is about as fast as the full table representation `(-f)', and for some sets of patterns will be considerably smaller (and for others, larger). In general, if the pattern set contains both "keywords" and a catch-all, "identifier" rule, such as in the set:

```
"case"    return TOK_CASE;
"switch"  return TOK_SWITCH;
...
"default" return TOK_DEFAULT;
[a-z]+    return TOK_ID;
```

then you're better off using the full table representation. If only the "identifier" rule is present and you then use a hash table or some such to detect the keywords, you're better off using `-F'.

This option is equivalent to `-CFr' (see below). It cannot be used with `-+'.

`-I'

instructs `flex` to generate an *interactive* scanner. An interactive scanner is one that only looks ahead to decide what token has been matched if it absolutely must. It turns out that always looking one extra character ahead, even if the scanner has already seen enough text to disambiguate the current token, is a bit faster than only looking ahead when necessary. But scanners that always look ahead give dreadful interactive performance; for example, when a user types a newline, it is not recognized as a newline token until they enter *another* token, which often means typing in another whole line.

`Flex` scanners default to *interactive* unless you use the `-Cf' or `-CF' table-compression options (see below). That's because if you're looking for high-performance you should be using one of these options, so if you didn't, `flex` assumes you'd rather trade off a bit of run-time performance for intuitive interactive behavior. Note also that you *cannot* use `-I' in conjunction with `-Cf' or `-CF'. Thus, this option is not really needed; it is on by default for all those cases in which it is allowed.

You can force a scanner to *not* be interactive by using `-B' (see above).

`-L'

instructs `flex` not to generate `#line' directives. Without this option, `flex` peppers the generated scanner with #line directives so error messages in the actions will be correctly located with respect to either the original `flex` input file (if the errors are due to code in the input file), or `` `lex.yy.c' `` (if the errors are `flex's` fault -- you should report these sorts of errors to the email address given below).

`-T'

makes `flex` run in `trace` mode. It will generate a lot of messages to `stderr` concerning the form of the input and the resultant non-deterministic and deterministic finite automata. This option

is mostly for use in maintaining `flex`.

`-V'

prints the version number to `stdout` and exits. `--version' is a synonym for `-V'.

`-7'

instructs `flex` to generate a 7-bit scanner, i.e., one which can only recognized 7-bit characters in its input. The advantage of using `-7' is that the scanner's tables can be up to half the size of those generated using the `-8' option (see below). The disadvantage is that such scanners often hang or crash if their input contains an 8-bit character.

Note, however, that unless you generate your scanner using the `-Cf' or `-CF' table compression options, use of `-7' will save only a small amount of table space, and make your scanner considerably less portable. `Flex's` default behavior is to generate an 8-bit scanner unless you use the `-Cf' or `-CF', in which case `flex` defaults to generating 7-bit scanners unless your site was always configured to generate 8-bit scanners (as will often be the case with non-USA sites). You can tell whether flex generated a 7-bit or an 8-bit scanner by inspecting the flag summary in the `-v' output as described above.

Note that if you use `-Cfe' or `-CFe' (those table compression options, but also using equivalence classes as discussed see below), flex still defaults to generating an 8-bit scanner, since usually with these compression options full 8-bit tables are not much more expensive than 7-bit tables.

`-8'

instructs `flex` to generate an 8-bit scanner, i.e., one which can recognize 8-bit characters. This flag is only needed for scanners generated using `-Cf' or `-CF', as otherwise flex defaults to generating an 8-bit scanner anyway.

See the discussion of `-7' above for flex's default behavior and the tradeoffs between 7-bit and 8-bit scanners.

`-+'

specifies that you want flex to generate a C++ scanner class. See the section on Generating C++ Scanners below for details.

`-C[aefFmr]'

controls the degree of table compression and, more generally, trade-offs between small scanners and fast scanners.

`-Ca' ("align") instructs flex to trade off larger tables in the generated scanner for faster performance because the elements of the tables are better aligned for memory access and computation. On some RISC architectures, fetching and manipulating long-words is more efficient than with smaller-sized units such as shortwords. This option can double the size of the tables used by your scanner.

`-Ce' directs `flex` to construct equivalence classes, i.e., sets of characters which have identical lexical properties (for example, if the only appearance of digits in the `flex` input is in the character class "[0-9]" then the digits '0', '1', ..., '9' will all be put in the same equivalence class). Equivalence classes usually give dramatic reductions in the final table/object file sizes (typically a factor of 2-5) and are pretty cheap performance-wise (one array look-up per character scanned).

`-Cf' specifies that the *full* scanner tables should be generated - `flex` should not compress the tables by taking advantages of similar transition functions for different states.

`-CF' specifies that the alternate fast scanner representation (described above under the `-F' flag) should be used. This option cannot be used with `-+'.

`-Cm' directs `flex` to construct meta-equivalence classes, which are sets of equivalence classes (or characters, if equivalence classes are not being used) that are commonly used together. Meta-equivalence classes are often a big win when using compressed tables, but they have a moderate performance impact (one or two "if" tests and one array look-up per character scanned).

`-Cr' causes the generated scanner to *bypass* use of the standard I/O library (stdio) for input. Instead of calling `fread()' or `getc()', the scanner will use the `read()' system call, resulting in a performance gain which varies from system to system, but in general is probably negligible unless you are also using `-Cf' or `-CF'. Using `-Cr' can cause strange behavior if, for example, you read from `yyin` using stdio prior to calling the scanner (because the scanner will miss whatever text your previous reads left in the stdio input buffer).

`-Cr' has no effect if you define `YY_INPUT` (see The Generated Scanner above).

A lone `-C' specifies that the scanner tables should be compressed but neither equivalence classes nor meta-equivalence classes should be used.

The options `-Cf' or `-CF' and `-Cm' do not make sense together - there is no opportunity for meta-equivalence classes if the table is not being compressed. Otherwise the options may be freely mixed, and are cumulative.

The default setting is `-Cem', which specifies that `flex` should generate equivalence classes and meta-equivalence classes. This setting provides the highest degree of table compression. You can trade off faster-executing scanners at the cost of larger tables with the following generally being true:

```
slowest & smallest
      -Cem
      -Cm
      -Ce
      -C
      -C{f,F}e
      -C{f,F}
      -C{f,F}a
fastest & largest
```

Note that scanners with the smallest tables are usually generated and compiled the quickest, so during development you will usually want to use the default, maximal compression.

`-Cfe' is often a good compromise between speed and size for production scanners.

`-ooutput'

directs flex to write the scanner to the file `out-' `put` instead of `` `lex.yy.c' ``. If you combine `-o'

with the `-t' option, then the scanner is written to `stdout' but its `#line' directives (see the `-L' option above) refer to the file `output'.

`-Pprefix'

changes the default `yy' prefix used by `flex' for all globally-visible variable and function names to instead be prefix. For example, `-Pfoo' changes the name of `yytext' to `footext'. It also changes the name of the default output file from `lex.yy.c' to `lex.foo.c'. Here are all of the names affected:

```
yy_create_buffer
yy_delete_buffer
yy_flex_debug
yy_init_buffer
yy_flush_buffer
yy_load_buffer_state
yy_switch_to_buffer
yyin
yyleng
yylex
yylineno
yyout
yyrestart
yytext
yywrap
```

(If you are using a C++ scanner, then only `yywrap' and `yyFlexLexer' are affected.) Within your scanner itself, you can still refer to the global variables and functions using either version of their name; but externally, they have the modified name.

This option lets you easily link together multiple `flex' programs into the same executable. Note, though, that using this option also renames `yywrap()', so you now *must* either provide your own (appropriately-named) version of the routine for your scanner, or use `%option noyywrap', as linking with `-lfl' no longer provides one for you by default.

`-Sskeleton_file'

overrides the default skeleton file from which `flex' constructs its scanners. You'll never need this option unless you are doing `flex' maintenance or development.

`flex' also provides a mechanism for controlling options within the scanner specification itself, rather than from the flex command-line. This is done by including `%option' directives in the first section of the scanner specification. You can specify multiple options with a single `%option' directive, and multiple directives in the first section of your flex input file. Most options are given simply as names, optionally preceded by the word "no" (with no intervening whitespace) to negate their meaning. A number are equivalent to flex flags or their negation:

```
7bit            -7 option
8bit            -8 option
```


```
align           -Ca option
backup          -b option
batch           -B option
c++             -+ option

caseful or
case-sensitive  opposite of -i (default)

case-insensitive or
caseless        -i option

debug           -d option
default         opposite of -s option
ecs             -Ce option
fast            -F option
full            -f option
interactive     -I option
lex-compat      -l option
meta-ecs        -Cm option
perf-report     -p option
read            -Cr option
stdout          -t option
verbose         -v option
warn            opposite of -w option
                (use "%option nowarn" for -w)

array           equivalent to "%array"
pointer         equivalent to "%pointer" (default)
```

Some `%option's' provide features otherwise not available:

`always-interactive'

instructs flex to generate a scanner which always considers its input "interactive". Normally, on each new input file the scanner calls `isatty()' in an attempt to determine whether the scanner's input source is interactive and thus should be read a character at a time. When this option is used, however, then no such call is made.

`main'

directs flex to provide a default `main()' program for the scanner, which simply calls `yylex()'. This option implies `noyywrap` (see below).

`never-interactive'

instructs flex to generate a scanner which never considers its input "interactive" (again, no call made to `isatty())'. This is the opposite of `always-' *interactive*.

`stack'

enables the use of start condition stacks (see Start Conditions above).

`stdinit'

if unset (i.e., `%option nostdinit') initializes `yyin` and `yyout` to nil `FILE` pointers, instead of `stdin` and `stdout`.

`yylineno'

directs `flex` to generate a scanner that maintains the number of the current line read from its input in the global variable `yylineno`. This option is implied by `%option lex-compat'.

`yywrap'

if unset (i.e., `%option noyywrap'), makes the scanner not call `yywrap()' upon an end-of-file, but simply assume that there are no more files to scan (until the user points `yyin` at a new file and calls `yylex()' again).

`flex` scans your rule actions to determine whether you use the `REJECT` or `yymore()' features. The `reject` and `yymore` options are available to override its decision as to whether you use the options, either by setting them (e.g., `%option reject') to indicate the feature is indeed used, or unsetting them to indicate it actually is not used (e.g., `%option noyymore').

Three options take string-delimited values, offset with '=':

```
%option outfile="ABC"
```

is equivalent to `-oABC', and

```
%option prefix="XYZ"
```

is equivalent to `-PXYZ'.

Finally,

```
%option yyclass="foo"
```

only applies when generating a C++ scanner (`-+' option). It informs `flex` that you have derived `foo' as a subclass of `yyFlexLexer` so `flex` will place your actions in the member function `foo::yylex()' instead of `yyFlexLexer::yylex()'. It also generates a `yyFlexLexer::yylex()' member function that emits a run-time error (by invoking `yyFlexLexer::LexerError()') if called. See Generating C++ Scanners, below, for additional information.

A number of options are available for lint purists who want to suppress the appearance of unneeded routines in the generated scanner. Each of the following, if unset, results in the corresponding routine not appearing in the generated scanner:

```
input, unput
yy_push_state, yy_pop_state, yy_top_state
yy_scan_buffer, yy_scan_bytes, yy_scan_string
```

(though `yy_push_state()' and friends won't appear anyway unless you use `%option stack').

Performance considerations

The main design goal of `flex` is that it generate high-performance scanners. It has been optimized for dealing well with large sets of rules. Aside from the effects on scanner speed of the table compression `-C' options outlined above, there are a number of options/actions which degrade performance. These are, from most expensive to least:

```
REJECT
%option yylineno
arbitrary trailing context

pattern sets that require backing up
%array
%option interactive
%option always-interactive

'^' beginning-of-line operator
yymore()
```

with the first three all being quite expensive and the last two being quite cheap. Note also that `unput()' is implemented as a routine call that potentially does quite a bit of work, while `yyless()' is a quite-cheap macro; so if just putting back some excess text you scanned, use `yyless()'.

`REJECT` should be avoided at all costs when performance is important. It is a particularly expensive option.

Getting rid of backing up is messy and often may be an enormous amount of work for a complicated scanner. In principal, one begins by using the `-b' flag to generate a `` `lex.backup' `` file. For example, on the input

```
%%
foo         return TOK_KEYWORD;
foobar      return TOK_KEYWORD;
```

the file looks like:

```
State #6 is non-accepting -
 associated rule line numbers:
       2       3
 out-transitions: [ o ]
 jam-transitions: EOF [ \001-n  p-\177 ]

State #8 is non-accepting -
 associated rule line numbers:
       3
 out-transitions: [ a ]
```


```
 jam-transitions: EOF [ \001-`  b-\177 ]

State #9 is non-accepting -
 associated rule line numbers:
       3
 out-transitions: [ r ]
 jam-transitions: EOF [ \001-q  s-\177 ]

Compressed tables always back up.
```

The first few lines tell us that there's a scanner state in which it can make a transition on an 'o' but not on any other character, and that in that state the currently scanned text does not match any rule. The state occurs when trying to match the rules found at lines 2 and 3 in the input file. If the scanner is in that state and then reads something other than an 'o', it will have to back up to find a rule which is matched. With a bit of head-scratching one can see that this must be the state it's in when it has seen "fo". When this has happened, if anything other than another 'o' is seen, the scanner will have to back up to simply match the 'f' (by the default rule).

The comment regarding State #8 indicates there's a problem when "foob" has been scanned. Indeed, on any character other than an 'a', the scanner will have to back up to accept "foo". Similarly, the comment for State #9 concerns when "fooba" has been scanned and an 'r' does not follow.

The final comment reminds us that there's no point going to all the trouble of removing backing up from the rules unless we're using `-Cf' or `-CF', since there's no performance gain doing so with compressed scanners.

The way to remove the backing up is to add "error" rules:

```
%%
foo         return TOK_KEYWORD;
foobar      return TOK_KEYWORD;

fooba       |
foob        |
fo          {
            /* false alarm, not really a keyword */
            return TOK_ID;
            }
```

Eliminating backing up among a list of keywords can also be done using a "catch-all" rule:

```
%%
foo         return TOK_KEYWORD;
foobar      return TOK_KEYWORD;

[a-z]+      return TOK_ID;
```

This is usually the best solution when appropriate.

Backing up messages tend to cascade. With a complicated set of rules it's not uncommon to get hundr of messages. If one can decipher them, though, it often only takes a dozen or so rules to eliminate the backing up (though it's easy to make a mistake and have an error rule accidentally match a valid token. A possible future `flex` feature will be to automatically add rules to eliminate backing up).

It's important to keep in mind that you gain the benefits of eliminating backing up only if you eliminate *every* instance of backing up. Leaving just one means you gain nothing.

Variable trailing context (where both the leading and trailing parts do not have a fixed length) entails almost the same performance loss as `REJECT` (i.e., substantial). So when possible a rule like:

```
%%
mouse|rat/(cat|dog)   run();
```

is better written:

```
%%
mouse/cat|dog         run();
rat/cat|dog           run();
```

or as

```
%%
mouse|rat/cat         run();
mouse|rat/dog         run();
```

Note that here the special '|' action does *not* provide any savings, and can even make things worse (see Deficiencies / Bugs below).

Another area where the user can increase a scanner's performance (and one that's easier to implement) arises from the fact that the longer the tokens matched, the faster the scanner will run. This is because with long tokens the processing of most input characters takes place in the (short) inner scanning loop, and does not often have to go through the additional work of setting up the scanning environment (e.g., `yytext`) for the action. Recall the scanner for C comments:

```
%x comment
%%
        int line_num = 1;

"/*"         BEGIN(comment);

<comment>[^*\n]*
<comment>"*"+[^*/\n]*
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(INITIAL);
```

This could be sped up by writing it as:


```
%x comment
%%
        int line_num = 1;

"/*"         BEGIN(comment);

<comment>[^*\n]*
<comment>[^*\n]*\n      ++line_num;
<comment>"*"+[^*/\n]*
<comment>"*"+[^*/\n]*\n ++line_num;
<comment>"*"+"/"        BEGIN(INITIAL);
```

Now instead of each newline requiring the processing of another action, recognizing the newlines is "distributed" over the other rules to keep the matched text as long as possible. Note that *adding* rules does *not* slow down the scanner! The speed of the scanner is independent of the number of rules or (modulo the considerations given at the beginning of this section) how complicated the rules are with regard to operators such as '*' and '|'.

A final example in speeding up a scanner: suppose you want to scan through a file containing identifiers and keywords, one per line and with no other extraneous characters, and recognize all the keywords. A natural first approach is:

```
%%
asm      |
auto     |
break    |
... etc ...
volatile |
while    /* it's a keyword */

.|\n     /* it's not a keyword */
```

To eliminate the back-tracking, introduce a catch-all rule:

```
%%
asm      |
auto     |
break    |
... etc ...
volatile |
while    /* it's a keyword */

[a-z]+   |
.|\n     /* it's not a keyword */
```

Now, if it's guaranteed that there's exactly one word per line, then we can reduce the total number of

matches by a half by merging in the recognition of newlines with that of the other tokens:

```
%%
asm\n    |
auto\n   |
break\n  |
... etc ...
volatile\n |
while\n  /* it's a keyword */

[a-z]+\n |
.|\n     /* it's not a keyword */
```

One has to be careful here, as we have now reintroduced backing up into the scanner. In particular, while *we* know that there will never be any characters in the input stream other than letters or newlines, `flex` can't figure this out, and it will plan for possibly needing to back up when it has scanned a token like "auto" and then the next character is something other than a newline or a letter. Previously it would then just match the "auto" rule and be done, but now it has no "auto" rule, only a "auto\n" rule. To eliminate the possibility of backing up, we could either duplicate all rules but without final newlines, or, since we never expect to encounter such an input and therefore don't how it's classified, we can introduce one more catch-all rule, this one which doesn't include a newline:

```
%%
asm\n    |
auto\n   |
break\n  |
... etc ...
volatile\n |
while\n  /* it's a keyword */

[a-z]+\n |
[a-z]+   |
.|\n     /* it's not a keyword */
```

Compiled with `-Cf', this is about as fast as one can get a `flex` scanner to go for this particular problem.

A final note: `flex` is slow when matching NUL's, particularly when a token contains multiple NUL's. It's best to write rules which match *short* amounts of text if it's anticipated that the text will often include NUL's.

Another final note regarding performance: as mentioned above in the section How the Input is Matched, dynamically resizing `yytext` to accommodate huge tokens is a slow process because it presently requires that the (huge) token be rescanned from the beginning. Thus if performance is vital, you should attempt to match "large" quantities of text but not "huge" quantities, where the cutoff between the two is at about 8K characters/token.

Generating C++ scanners

`flex` provides two different ways to generate scanners for use with C++. The first way is to simply compile a scanner generated by `flex` using a C++ compiler instead of a C compiler. You should not encounter any compilations errors (please report any you find to the email address given in the Author section below). You can then use C++ code in your rule actions instead of C code. Note that the default input source for your scanner remains `yyin`, and default echoing is still done to `yyout`. Both of these remain `FILE *' variables and not C++ `streams`.

You can also use `flex` to generate a C++ scanner class, using the `-+' option, (or, equivalently, `%option c++'), which is automatically specified if the name of the flex executable ends in a `+', such as `flex++`. When using this option, flex defaults to generating the scanner to the file `` `lex.yy.cc' `` instead of `` `lex.yy.c' ``. The generated scanner includes the header file `` `FlexLexer.h' ``, which defines the interface to two C++ classes.

The first class, `FlexLexer`, provides an abstract base class defining the general scanner class interface. It provides the following member functions:

`const char* YYText()'

> returns the text of the most recently matched token, the equivalent of `yytext`.

`int YYLeng()'

> returns the length of the most recently matched token, the equivalent of `yyleng`.

`int lineno() const'

> returns the current input line number (see `%option yylineno'), or 1 if `%option yylineno' was not used.

`void set_debug(int flag)'

> sets the debugging flag for the scanner, equivalent to assigning to `yy_flex_debug` (see the Options section above). Note that you must build the scanner using `%option debug' to include debugging information in it.

`int debug() const'

> returns the current setting of the debugging flag.

Also provided are member functions equivalent to `yy_switch_to_buffer(), yy_create_buffer()' (though the first argument is an `istream*' object pointer and not a `FILE*', `yy_flush_buffer()', `yy_delete_buffer()', and `yyrestart()' (again, the first argument is a `istream*' object pointer).

The second class defined in `` `FlexLexer.h' `` is `yyFlexLexer`, which is derived from `FlexLexer`. It defines the following additional member functions:

`yyFlexLexer(istream* arg_yyin = 0, ostream* arg_yyout = 0)'

> constructs a `yyFlexLexer` object using the given streams for input and output. If not specified, the streams default to `cin` and `cout`, respectively.

`virtual int yylex()'

> performs the same role is `yylex()' does for ordinary flex scanners: it scans the input stream, consuming tokens, until a rule's action returns a value. If you derive a subclass S from

`yyFlexLexer` and want to access the member functions and variables of S inside `yylex()', then you need to use `%option yyclass="S"' to inform `flex` that you will be using that subclass instead of `yyFlexLexer`. In this case, rather than generating `yyFlexLexer::yylex()', `flex` generates `S::yylex()' (and also generates a dummy `yyFlexLexer::yylex()' that calls `yyFlexLexer::LexerError()' if called).

`virtual void switch_streams(istream* new_in = 0, ostream* new_out = 0)'

reassigns `yyin` to `new_in` (if non-nil) and `yyout` to `new_out` (ditto), deleting the previous input buffer if `yyin` is reassigned.

`int yylex(istream* new_in = 0, ostream* new_out = 0)'

first switches the input streams via `switch_streams( new_in, new_out )' and then returns the value of `yylex()'.

In addition, `yyFlexLexer` defines the following protected virtual functions which you can redefine in derived classes to tailor the scanner:

`virtual int LexerInput(char* buf, int max_size)'

reads up to `max_size' characters into buf and returns the number of characters read. To indicate end-of-input, return 0 characters. Note that "interactive" scanners (see the `-B' and `-I' flags) define the macro `YY_INTERACTIVE`. If you redefine `LexerInput()` and need to take different actions depending on whether or not the scanner might be scanning an interactive input source, you can test for the presence of this name via `#ifdef'.

`virtual void LexerOutput(const char* buf, int size)'

writes out size characters from the buffer buf, which, while NUL-terminated, may also contain "internal" NUL's if the scanner's rules can match text with NUL's in them.

`virtual void LexerError(const char* msg)'

reports a fatal error message. The default version of this function writes the message to the stream `cerr` and exits.

Note that a `yyFlexLexer` object contains its *entire* scanning state. Thus you can use such objects to create reentrant scanners. You can instantiate multiple instances of the same `yyFlexLexer` class, and you can also combine multiple C++ scanner classes together in the same program using the `-P' option discussed above. Finally, note that the `%array' feature is not available to C++ scanner classes; you must use `%pointer' (the default).

Here is an example of a simple C++ scanner:

```
    // An example of using the flex C++ scanner class.

%{
int mylineno = 0;
%}

string  \"[^\n"]+\"

ws      [ \t]+
```


```
alpha   [A-Za-z]
dig     [0-9]
name    ({alpha}|{dig}|\$)({alpha}|{dig}|[_.\-/$])*
num1    [-+]?{dig}+\.?([eE][-+]?{dig}+)?
num2    [-+]?{dig}*\.{dig}+([eE][-+]?{dig}+)?
number  {num1}|{num2}

%%

{ws}    /* skip blanks and tabs */

"/*"    {
        int c;

        while((c = yyinput()) != 0)
            {
            if(c == '\n')
                ++mylineno;

            else if(c == '*')
                {
                if((c = yyinput()) == '/')
                    break;
                else
                    unput(c);
                }
            }
        }

{number}  cout << "number " << YYText() << '\n';

\n        mylineno++;

{name}    cout << "name " << YYText() << '\n';

{string}  cout << "string " << YYText() << '\n';

%%

Version 2.5                December 1994                        44

int main( int /* argc */, char** /* argv */ )
    {
    FlexLexer* lexer = new yyFlexLexer;
    while(lexer->yylex() != 0)
```


```
    ;
return 0;
}
```

If you want to create multiple (different) lexer classes, you use the `-P' flag (or the `prefix=' option) to rename each `yyFlexLexer` to some other `xxFlexLexer`. You then can include `<FlexLexer.h>' in your other sources once per lexer class, first renaming `yyFlexLexer` as follows:

```
#undef yyFlexLexer
#define yyFlexLexer xxFlexLexer
#include <FlexLexer.h>

#undef yyFlexLexer
#define yyFlexLexer zzFlexLexer
#include <FlexLexer.h>
```

if, for example, you used `%option prefix="xx"' for one of your scanners and `%option prefix="zz"' for the other.

IMPORTANT: the present form of the scanning class is *experimental* and may change considerably between major releases.

Incompatibilities with `lex` and POSIX

`flex` is a rewrite of the AT&T Unix `lex` tool (the two implementations do not share any code, though), with some extensions and incompatibilities, both of which are of concern to those who wish to write scanners acceptable to either implementation. Flex is fully compliant with the POSIX `lex` specification, except that when using `%pointer' (the default), a call to `unput()' destroys the contents of `yytext`, which is counter to the POSIX specification.

In this section we discuss all of the known areas of incompatibility between flex, AT&T lex, and the POSIX specification.

`flex's` `-l' option turns on maximum compatibility with the original AT&T `lex` implementation, at the cost of a major loss in the generated scanner's performance. We note below which incompatibilities can be overcome using the `-l' option.

`flex` is fully compatible with `lex` with the following exceptions:

- The undocumented `lex` scanner internal variable `yylineno` is not supported unless `-l' or `%option yylineno' is used. `yylineno` should be maintained on a per-buffer basis, rather than a per-scanner (single global variable) basis. `yylineno` is not part of the POSIX specification.
- The `input()' routine is not redefinable, though it may be called to read characters following whatever has been matched by a rule. If `input()' encounters an end-of-file the normal `yywrap()' processing is done. A "real" end-of-file is returned by `input()' as `EOF`.

  Input is instead controlled by defining the `YY_INPUT` macro.

The `flex` restriction that `input()' cannot be redefined is in accordance with the POSIX specification, which simply does not specify any way of controlling the scanner's input other th by making an initial assignment to `yyin`.

- The `unput()' routine is not redefinable. This restriction is in accordance with POSIX.
- `flex` scanners are not as reentrant as `lex` scanners. In particular, if you have an interactive scanner and an interrupt handler which long-jumps out of the scanner, and the scanner is subsequently called again, you may get the following message:

  ```
  fatal flex scanner internal error--end of buffer missed
  ```

  To reenter the scanner, first use

  ```
  yyrestart( yyin );
  ```

  Note that this call will throw away any buffered input; usually this isn't a problem with an interactive scanner.

  Also note that flex C++ scanner classes *are* reentrant, so if using C++ is an option for you, you should use them instead. See "Generating C++ Scanners" above for details.
- `output()' is not supported. Output from the `ECHO' macro is done to the file-pointer `yyout` (default `stdout`).

  `output()' is not part of the POSIX specification.
- `lex` does not support exclusive start conditions (%x), though they are in the POSIX specification.
- When definitions are expanded, `flex` encloses them in parentheses. With lex, the following:

  ```
  NAME    [A-Z][A-Z0-9]*
  %%
  foo{NAME}?      printf( "Found it\n" );
  %%
  ```

  will not match the string "foo" because when the macro is expanded the rule is equivalent to "foo[A-Z][A-Z0-9]*?" and the precedence is such that the '?' is associated with "[A-Z0-9]*". With `flex`, the rule will be expanded to "foo([A-Z][A-Z0-9]*)?" and so the string "foo" will match.

  Note that if the definition begins with `^' or ends with `$' then it is *not* expanded with parentheses, to allow these operators to appear in definitions without losing their special meanings. But the `<s>, /', and `<<EOF>>' operators cannot be used in a `flex` definition.

  Using `-l' results in the `lex` behavior of no parentheses around the definition.

  The POSIX specification is that the definition be enclosed in parentheses.
- Some implementations of `lex` allow a rule's action to begin on a separate line, if the rule's pattern has trailing whitespace:

  ```
  %%
  ```


```
foo|bar<space here>
  { foobar_action(); }
```

`flex` does not support this feature.

- The `lex` `%r' (generate a Ratfor scanner) option is not supported. It is not part of the POSIX specification.
- After a call to `unput()', `yytext` is undefined until the next token is matched, unless the scanner was built using `%array'. This is not the case with `lex` or the POSIX specification. The `-l' option does away with this incompatibility.
- The precedence of the `{}' (numeric range) operator is different. `lex` interprets "abc{1,3}" as "match one, two, or three occurrences of 'abc'", whereas `flex` interprets it as "match 'ab' followed by one, two, or three occurrences of 'c'". The latter is in agreement with the POSIX specification.
- The precedence of the `^' operator is different. `lex` interprets "^foo|bar" as "match either 'foo' at the beginning of a line, or 'bar' anywhere", whereas `flex` interprets it as "match either 'foo' or 'bar' if they come at the beginning of a line". The latter is in agreement with the POSIX specification.
- The special table-size declarations such as `%a' supported by `lex` are not required by `flex` scanners; `flex` ignores them.
- The name FLEX_SCANNER is #define'd so scanners may be written for use with either `flex` or `lex`. Scanners also include `YY_FLEX_MAJOR_VERSION` and `YY_FLEX_MINOR_VERSION` indicating which version of `flex` generated the scanner (for example, for the 2.5 release, these defines would be 2 and 5 respectively).

The following `flex` features are not included in `lex` or the POSIX specification:

```
C++ scanners
%option
start condition scopes
start condition stacks
interactive/non-interactive scanners
yy_scan_string() and friends
yyterminate()
yy_set_interactive()
yy_set_bol()
YY_AT_BOL()
<<EOF>>
<*>
YY_DECL
YY_START
YY_USER_ACTION
YY_USER_INIT
#line directives
%{}'s around actions
multiple actions on a line
```

plus almost all of the flex flags. The last feature in the list refers to the fact that with `flex` you can put

multiple actions on the same line, separated with semicolons, while with `lex`, the following

```
foo     handle_foo(); ++num_foos_seen;
```

is (rather surprisingly) truncated to

```
foo     handle_foo();
```

`flex` does not truncate the action. Actions that are not enclosed in braces are simply terminated at the end of the line.

Diagnostics

`warning, rule cannot be matched'

indicates that the given rule cannot be matched because it follows other rules that will always match the same text as it. For example, in the following "foo" cannot be matched because it comes after an identifier "catch-all" rule:

```
[a-z]+    got_identifier();
foo       got_foo();
```

Using `REJECT` in a scanner suppresses this warning.

`warning, -s option given but default rule can be matched'

means that it is possible (perhaps only in a particular start condition) that the default rule (match any single character) is the only one that will match a particular input. Since `-s' was given, presumably this is not intended.

`reject_used_but_not_detected undefined'

`yymore_used_but_not_detected undefined'

These errors can occur at compile time. They indicate that the scanner uses `REJECT` or `yymore()' but that `flex` failed to notice the fact, meaning that `flex` scanned the first two sections looking for occurrences of these actions and failed to find any, but somehow you snuck some in (via a #include file, for example). Use `%option reject' or `%option yymore' to indicate to flex that you really do use these features.

`flex scanner jammed'

a scanner compiled with `-s' has encountered an input string which wasn't matched by any of its rules. This error can also occur due to internal problems.

`token too large, exceeds YYLMAX'

your scanner uses `%array' and one of its rules matched a string longer than the `YYL-' `MAX` constant (8K bytes by default). You can increase the value by #define'ing `YYLMAX` in the definitions section of your `flex` input.

`scanner requires -8 flag to use the character 'x"

Your scanner specification includes recognizing the 8-bit character x and you did not specify the -8

flag, and your scanner defaulted to 7-bit because you used the `-Cf' or `-CF' table compression options. See the discussion of the `-7' flag for details.

`flex scanner push-back overflow'

you used `unput()' to push back so much text that the scanner's buffer could not hold both the pushed-back text and the current token in `yytext'. Ideally the scanner should dynamically resize the buffer in this case, but at present it does not.

`input buffer overflow, can't enlarge buffer because scanner uses REJECT'

the scanner was working on matching an extremely large token and needed to expand the input buffer. This doesn't work with scanners that use `REJECT'.

`fatal flex scanner internal error--end of buffer missed'

This can occur in an scanner which is reentered after a long-jump has jumped out (or over) the scanner's activation frame. Before reentering the scanner, use:

```
yyrestart( yyin );
```

or, as noted above, switch to using the C++ scanner class.

`too many start conditions in <> construct!'

you listed more start conditions in a <> construct than exist (so you must have listed at least one of them twice).

Files

`-lfl'

library with which scanners must be linked.

`lex.yy.c'

generated scanner (called `lexyy.c' on some systems).

`lex.yy.cc'

generated C++ scanner class, when using `-+'.

`<FlexLexer.h>'

header file defining the C++ scanner base class, `FlexLexer', and its derived class, `yyFlexLexer'.

`flex.skl'

skeleton scanner. This file is only used when building flex, not when flex executes.

`lex.backup'

backing-up information for `-b' flag (called `lex.bck' on some systems).

Deficiencies / Bugs

Some trailing context patterns cannot be properly matched and generate warning messages ("dangerous trailing context"). These are patterns where the ending of the first part of the rule matches the beginning of the second part, such as "zx*/xy*", where the 'x*' matches the 'x' at the beginning of the trailing context. (Note that the POSIX draft states that the text matched by such patterns is undefined.)

For some trailing context rules, parts which are actually fixed-length are not recognized as such, leading to the abovementioned performance loss. In particular, parts using '|' or {n} (such as "foo{3}") are always considered variable-length.

Combining trailing context with the special '|' action can result in *fixed* trailing context being turned into the more expensive variable trailing context. For example, in the following:

```
%%
abc      |
xyz/def
```

Use of `unput()' invalidates yytext and yyleng, unless the `%array' directive or the `-l' option has been used.

Pattern-matching of NUL's is substantially slower than matching other characters.

Dynamic resizing of the input buffer is slow, as it entails rescanning all the text matched so far by the current (generally huge) token.

Due to both buffering of input and read-ahead, you cannot intermix calls to <stdio.h> routines, such as, for example, `getchar()', with `flex` rules and expect it to work. Call `input()' instead.

The total table entries listed by the `-v' flag excludes the number of table entries needed to determine what rule has been matched. The number of entries is equal to the number of DFA states if the scanner does not use `REJECT`, and somewhat greater than the number of states if it does.

`REJECT` cannot be used with the `-f' or `-F' options.

The `flex` internal algorithms need documentation.

See also

`lex`(1), `yacc`(1), `sed`(1), `awk`(1).

John Levine, Tony Mason, and Doug Brown: Lex & Yacc; O'Reilly and Associates. Be sure to get the 2nd edition.

M. E. Lesk and E. Schmidt, LEX - Lexical Analyzer Generator.

Alfred Aho, Ravi Sethi and Jeffrey Ullman: Compilers: Principles, Techniques and Tools; Addison-Wesley (1986). Describes the pattern-matching techniques used by `flex` (deterministic finite automata).

Author

Vern Paxson, with the help of many ideas and much inspiration from Van Jacobson. Original version by Jef Poskanzer. The fast table representation is a partial implementation of a design done by Van Jacobson. The implementation was done by Kevin Gong and Vern Paxson.

Thanks to the many `flex` beta-testers, feedbackers, and contributors, especially Francois Pinard, Casey Leedom, Stan Adermann, Terry Allen, David Barker-Plummer, John Basrai, Nelson H.F. Beebe, `benson@odi.com', Karl Berry, Peter A. Bigot, Simon Blanchard, Keith Bostic, Frederic Brehm, Ian Brockbank, Kin Cho, Nick Christopher, Brian Clapper, J.T. Conklin, Jason Coughlin, Bill Cox, Nick Cropper, Dave Curtis, Scott David Daniels, Chris G. Demetriou, Theo Deraadt, Mike Donahue, Chuck Doucette, Tom Epperly, Leo Eskin, Chris Faylor, Chris Flatters, Jon Forrest, Joe Gayda, Kaveh R. Ghazi, Eric Goldman, Christopher M. Gould, Ulrich Grepel, Peer Griebel, Jan Hajic, Charles Hemphill, NORO Hideo, Jarkko Hietaniemi, Scott Hofmann, Jeff Honig, Dana Hudes, Eric Hughes, John Interrante, Ceriel Jacobs, Michal Jaegermann, Sakari Jalovaara, Jeffrey R. Jones, Henry Juengst, Klaus Kaempf, Jonathan I. Kamens, Terrence O Kane, Amir Katz, `ken@ken.hilco.com', Kevin B. Kenny, Steve Kirsch, Winfried Koenig, Marq Kole, Ronald Lamprecht, Greg Lee, Rohan Lenard, Craig Leres, John Levine, Steve Liddle, Mike Long, Mohamed el Lozy, Brian Madsen, Malte, Joe Marshall, Bengt Martensson, Chris Metcalf, Luke Mewburn, Jim Meyering, R. Alexander Milowski, Erik Naggum, G.T. Nicol, Landon Noll, James Nordby, Marc Nozell, Richard Ohnemus, Karsten Pahnke, Sven Panne, Roland Pesch, Walter Pelissero, Gaumond Pierre, Esmond Pitt, Jef Poskanzer, Joe Rahmeh, Jarmo Raiha, Frederic Raimbault, Pat Rankin, Rick Richardson, Kevin Rodgers, Kai Uwe Rommel, Jim Roskind, Alberto Santini, Andreas Scherer, Darrell Schiebel, Raf Schietekat, Doug Schmidt, Philippe Schnoebelen, Andreas Schwab, Alex Siegel, Eckehard Stolz, Jan-Erik Strvmquist, Mike Stump, Paul Stuart, Dave Tallman, Ian Lance Taylor, Chris Thewalt, Richard M. Timoney, Jodi Tsai, Paul Tuinenga, Gary Weik, Frank Whaley, Gerhard Wilhelms, Kent Williams, Ken Yap, Ron Zellar, Nathan Zelle, David Zuhn, and those whose names have slipped my marginal mail-archiving skills but whose contributions are appreciated all the same.

Thanks to Keith Bostic, Jon Forrest, Noah Friedman, John Gilmore, Craig Leres, John Levine, Bob Mulcahy, G.T. Nicol, Francois Pinard, Rich Salz, and Richard Stallman for help with various distribution headaches.

Thanks to Esmond Pitt and Earle Horton for 8-bit character support; to Benson Margulies and Fred Burke for C++ support; to Kent Williams and Tom Epperly for C++ class support; to Ove Ewerlid for support of NUL's; and to Eric Hughes for support of multiple buffers.

This work was primarily done when I was with the Real Time Systems Group at the Lawrence Berkeley Laboratory in Berkeley, CA. Many thanks to all there for the support I received.

Send comments to `vern@ee.lbl.gov'.

Using

The GNU Assembler

for the family

January 1994

Dean Elsner, Jay Fenlason & friends

Using

The Free Software Foundation Inc. thanks The Nice Computer Company of Australia for loaning Dean Elsner to write the first (Vax) version of `as` for Project GNU. The proprietors, management and staff of TNCCA thank FSF for distracting the boss while they got some work done.

Overview

This manual is a user guide to the GNU assembler . This version of the manual describes configured to generate code for architectures.

Here is a brief summary of how to invoke . For details, see section Command-Line Options.

```
 [ -a[cdhlns][=file] ] [ -D ]  [ --defsym sym=val ]
 [ -f ] [ --help ] [ -I dir ] [ -J ] [ -K ] [ -L ]
 [ -o objfile ] [ -R ] [ --statistics ] [ -v ] [ -version ]
 [ --version ] [ -W ] [ -w ] [ -x ] [ -Z ]

 [ -- | files ... ]
```

`-a[dhlns]`

Turn on listings, in any of a variety of ways:

`-ad`

omit debugging directives

`-ah`

include high-level source

`-al`

include assembly

`-an`

omit forms processing

`-as`

include symbols

`=file`

set the name of the listing file

You may combine these options; for example, use `-aln' for assembly listing without forms processing. The `=file' option, if used, must be the last one. By itself, `-a' defaults to `-ahls'---that is, all listings turned on.

- -D Ignored. This option is accepted for script compatibility with calls to other assemblers.
- --defsym sym=value Define the symbol sym to be value before assembling the input file. value must be an integer constant. As in C, a leading `0x' indicates a hexadecimal value, and a leading `0' indicates an octal value.
- -f "fast"---skip whitespace and comment preprocessing (assume source is compiler output).
- --help Print a summary of the command line options and exit.
- -I dir Add directory dir to the search list for `.include` directives.
- -J Don't warn about signed overflow.
- -K This option is accepted but has no effect on the family.
- -L Keep (in the symbol table) local symbols, starting with `L'.
- -o objfile Name the object-file output from objfile.
- -R Fold the data section into the text section.
- --statistics Print the maximum space (in bytes) and total time (in seconds) used by assembly.
- -v
- -version Print the `as` version.
- --version Print the `as` version and exit.
- -W Suppress warning messages.
- -w Ignored.
- -x Ignored.
- -Z Generate an object file even after errors.
- -- | files ... Standard input, or source files to assemble.

Structure of this Manual

This manual is intended to describe what you need to know to use GNU . We cover the syntax expected in source files, including notation for symbols, constants, and expressions; the directives that understands; and of course how to invoke .

We also cover special features in the configuration of , including assembler directives.

On the other hand, this manual is *not* intended as an introduction to programming in assembly language--let alone programming in general! In a similar vein, we make no attempt to introduce the machine architecture; we do *not* describe the instruction set, standard mnemonics, registers or addressing

modes that are standard to a particular architecture.

The GNU Assembler

GNU `as` is really a family of assemblers. This manual describes , a member of that family which is configured for the architectures. If you use (or have used) the GNU assembler on one architecture, you should find a fairly similar environment when you use it on another architecture. Each version has much in common with the others, including object file formats, most assembler directives (often called pseudo-ops) and assembler syntax.

is primarily intended to assemble the output of the GNU C compiler for use by the linker . Nevertheless, we've tried to make assemble correctly everything that other assemblers for the same machine would assemble.

Unlike older assemblers, is designed to assemble a source program in one pass of the source file. This has a subtle impact on the .org directive (see section `.org new-lc , fill`).

Object File Formats

The GNU assembler can be configured to produce several alternative object file formats. For the most part, this does not affect how you write assembly language programs; but directives for debugging symbols are typically different in different file formats. See section Symbol Attributes. On the , is configured to produce format object files.

Command Line

After the program name , the command line may contain options and file names. Options may appear in any order, and may be before, after, or between file names. The order of file names is significant.

`` `--' `` (two hyphens) by itself names the standard input file explicitly, as one of the files for to assemble.

Except for `--' any command line argument that begins with a hyphen (`-') is an option. Each option changes the behavior of . No option changes the way another option works. An option is a `-' followed by one or more letters; the case of the letter is important. All options are optional.

Some options expect exactly one file name to follow them. The file name may either immediately follow the option's letter (compatible with older assemblers) or it may be the next command argument (GNU standard). These two command lines are equivalent:

```
-o my-object-file.o mumble.s
-omy-object-file.o mumble.s
```

Input Files

We use the phrase source program, abbreviated source, to describe the program input to one run of . The program may be in one or more files; how the source is partitioned into files doesn't change the meaning of the source.

The source program is a concatenation of the text in all the files, in the order specified.

Each time you run it assembles exactly one source program. The source program is made up of one or more files. (The standard input is also a file.)

You give a command line that has zero or more input file names. The input files are read (from left file name to right). A command line argument (in any position) that has no special meaning is taken to be an input file name.

If you give no file names it attempts to read one input file from the standard input, which is normally your terminal. You may have to type ctl-D to tell there is no more program to assemble.

Use `--' if you need to explicitly name the standard input file in your command line.

If the source is empty, produces a small, empty object file.

Filenames and Line-numbers

There are two ways of locating a line in the input file (or files) and either may be used in reporting error messages. One way refers to a line number in a physical file; the other refers to a line number in a "logical" file. See section Error and Warning Messages.

Physical files are those files named in the command line given to .

Logical files are simply names declared explicitly by assembler directives; they bear no relation to physical files. Logical file names help error messages reflect the original source file, when source is itself synthesized from other files. See section `.app-file string`.

Output (Object) File

Every time you run it produces an output file, which is your assembly language program translated into numbers. This file is the object file. Its default name is `a.out`. `b.out` when is configured for the Intel 80960. You can give it another name by using the `-o` option. Conventionally, object file names end with `.o'. The default name is used for historical reasons: older assemblers were capable of assembling self-contained programs directly into a runnable program. (For some formats, this isn't currently possible, but it can be done for the `a.out` format.)

The object file is meant for input to the linker . It contains assembled program code, information to help integrate the assembled program into a runnable file, and (optionally) symbolic information for the debugger.

Error and Warning Messages

may write warnings and error messages to the standard error file (usually your terminal). This should not happen when a compiler runs automatically. Warnings report an assumption made so that could keep assembling a flawed program; errors report a grave problem that stops the assembly.

Warning messages have the format

```
file_name:NNN:Warning Message Text
```

(where **NNN** is a line number). If a logical file name has been given (see section `.app-file string`) it is used for the filename, otherwise the name of the current input file is used. If a logical line number was given (see section `.line line-number`) then it is used to calculate the number printed, otherwise the actual line in the current source file is printed. The message text is intended to be self explanatory (in the grand Unix tradition).

Error messages have the format

```
file_name:NNN:FATAL:Error Message Text
```

The file name and line number are derived as for warning messages. The actual message text may be rather less explanatory because many of them aren't supposed to happen.

Command-Line Options

This chapter describes command-line options available in *all* versions of the GNU assembler; @xref{Machine Dependencies}, for options specific to the .

If you are invoking via the GNU C compiler (version 2), you can use the `-Wa' option to pass arguments through to the assembler. The assembler arguments must be separated from each other (and the `-Wa') by commas. For example:

```
gcc -c -g -O -Wa,-alh,-L file.c
```

emits a listing to standard output with high-level and assembly source.

Usually you do not need to use this `-Wa' mechanism, since many compiler command-line options are automatically passed to the assembler by the compiler. (You can call the GNU compiler driver with the `-v' option to see precisely what options it passes to each compilation pass, including the assembler.)

Enable Listings: `-a[cdhlns]`

These options enable listing output from the assembler. By itself, `-a' requests high-level, assembly, and symbols listing. You can use other letters to select specific options for the list: `-ah' requests a high-level

language listing, `-al' requests an output-program assembly listing, and `-as' requests a symbol table listing. High-level listings require that a compiler debugging option like `-g' be used, and that assembly listings (`-al') be requested also.

Use the `-ac' option to omit false conditionals from a listing. Any lines which are not assembled because of a false `.if` (or `.ifdef`, or any other conditional), or a true `.if` followed by an `.else`, will be omitted from the listing.

Use the `-ad' option to omit debugging directives from the listing.

Once you have specified one of these options, you can further control listing output and its appearance using the directives `.list`, `.nolist`, `.psize`, `.eject`, `.title`, and `.sbttl`. The `-an' option turns off all forms processing. If you do not request listing output with one of the `-a' options, the listing-control directives have no effect.

The letters after `-a' may be combined into one option, *e.g.*, `-aln'.

`-D`

This option has no effect whatsoever, but it is accepted to make it more likely that scripts written for other assemblers also work with .

Work Faster: `-f`

`-f' should only be used when assembling programs written by a (trusted) compiler. `-f' stops the assembler from doing whitespace and comment preprocessing on the input file(s) before assembling them. See section Preprocessing.

> *Warning:* if you use `-f' when the files actually need to be preprocessed (if they contain comments, for example), does not work correctly.

`.include` search path: `-I` path

Use this option to add a path to the list of directories searches for files specified in `.include` directives (see section `.include "file"`). You may use `-I` as many times as necessary to include a variety of paths. The current working directory is always searched first; after that, searches any `-I' directories in the same order as they were specified (left to right) on the command line.

Difference Tables: `-K`

On the family, this option is allowed, but has no effect. It is permitted for compatibility with the GNU assembler on other platforms, where it can be used to warn when the assembler alters the machine code generated for `.word' directives in difference tables. The family does not have the addressing limitations that sometimes lead to this alteration on other platforms.

Include Local Labels: `-L`

Labels beginning with `L' (upper case only) are called local labels. See section Symbol Names. Normally you do not see such labels when debugging, because they are intended for the use of programs (like compilers) that compose assembler programs, not for your notice. Normally both and discard such labels, so you do not normally debug with them.

This option tells to retain those `L...' symbols in the object file. Usually if you do this you also tell the linker to preserve symbols whose names begin with `L'.

By default, a local label is any label beginning with `L', but each target is allowed to redefine the local label prefix.

Assemble in MRI Compatibility Mode: `-M`

The `-M` or `--mri` option selects MRI compatibility mode. This changes the syntax and pseudo-op handling of to make it compatible with the `ASM68K` or the `ASM960` (depending upon the configured target) assembler from Microtec Research. The exact nature of the MRI syntax will not be documented here; see the MRI manuals for more information. Note in particular that the handling of macros and macro arguments is somewhat different. The purpose of this option is to permit assembling existing MRI assembler code using .

The MRI compatibility is not complete. Certain operations of the MRI assembler depend upon its object file format, and can not be supported using other object file formats. Supporting these would require enhancing each object file format individually. These are:

- global symbols in common section

  The m68k MRI assembler supports common sections which are merged by the linker. Other object file formats do not support this. handles common sections by treating them as a single common symbol. It permits local symbols to be defined within a common section, but it can not support global symbols, since it has no way to describe them.

- complex relocations

  The MRI assemblers support relocations against a negated section address, and relocations which combine the start addresses of two or more sections. These are not support by other object file formats.

- `END` pseudo-op specifying start address

  The MRI `END` pseudo-op permits the specification of a start address. This is not supported by other object file formats. The start address may instead be specified using the `-e` option to the linker, or in a linker script.

- `IDNT`, `.ident` and `NAME` pseudo-ops

  The MRI `IDNT`, `.ident` and `NAME` pseudo-ops assign a module name to the output file. This is not supported by other object file formats.

- `ORG` pseudo-op

  The m68k MRI `ORG` pseudo-op begins an absolute section at a given address. This differs from the usual `.org` pseudo-op, which changes the location within the current section. Absolute sections are not supported by other object file formats. The address of a section may be assigned within a linker script.

There are some other features of the MRI assembler which are not supported by , typically either because they are difficult or because they seem of little consequence. Some of these may be supported in future releases.

- EBCDIC strings

  EBCDIC strings are not supported.

- packed binary coded decimal

  Packed binary coded decimal is not supported. This means that the `DC.P` and `DCB.P` pseudo-ops are not supported.

- `FEQU` pseudo-op

  The m68k `FEQU` pseudo-op is not supported.

- `NOOBJ` pseudo-op

  The m68k `NOOBJ` pseudo-op is not supported.

- `OPT` branch control options

  The m68k `OPT` branch control options---`B`, `BRS`, `BRB`, `BRL`, and `BRW`---are ignored. automatically relaxes all branches, whether forward or backward, to an appropriate size, so these options serve no purpose.

- `OPT` list control options

  The following m68k `OPT` list control options are ignored: `C`, `CEX`, `CL`, `CRE`, `E`, `G`, `I`, `M`, `MEX`, `MC`, `MD`, `X`.

- other `OPT` options

  The following m68k `OPT` options are ignored: `NEST`, `O`, `OLD`, `OP`, `P`, `PCO`, `PCR`, `PCS`, `R`.

- `OPT D` option is default

  The m68k `OPT D` option is the default, unlike the MRI assembler. `OPT NOD` may be used to turn it off.

- `XREF` pseudo-op.

  The m68k `XREF` pseudo-op is ignored.

- `.debug` pseudo-op

  The i960 `.debug` pseudo-op is not supported.

- `.extended` pseudo-op

  The i960 `.extended` pseudo-op is not supported.

- `.list` pseudo-op.

  The various options of the i960 `.list` pseudo-op are not supported.
- `.optimize` pseudo-op

  The i960 `.optimize` pseudo-op is not supported.
- `.output` pseudo-op

  The i960 `.output` pseudo-op is not supported.
- `.setreal` pseudo-op

  The i960 `.setreal` pseudo-op is not supported.

Name the Object File: `-o`

There is always one object file output when you run . By default it has the name `` `a.out' ``. `` `a.out' ``. You use this option (which takes exactly one filename) to give the object file a different name.

Whatever the object file is called, overwrites any existing file of the same name.

Join Data and Text Sections: `-R`

`-R` tells to write the object file as if all data-section data lives in the text section. This is only done at the very last moment: your binary data are the same, but data section parts are relocated differently. The data section part of your object file is zero bytes long because all its bytes are appended to the text section. (See section Sections and Relocation.)

When you specify `-R` it would be possible to generate shorter address displacements (because we do not have to cross between text and data section). We refrain from doing this simply for compatibility with older versions of . In future, `-R` may work this way.

Display Assembly Statistics: `--statistics`

Use `--statistics' to display two statistics about the resources used by : the maximum amount of space allocated during the assembly (in bytes), and the total execution time taken for the assembly (in CPU seconds).

Announce Version: `-v`

You can find out what version of as is running by including the option `-v' (which you can also spell as `-version') on the command line.

Suppress Warnings: -W

should never give a warning or error message when assembling compiler output. But programs written by people often cause to give a warning that a particular assumption was made. All such warnings are directed to the standard error file. If you use this option, no warnings are issued. This option only affects the warning messages: it does not change any particular of how assembles your file. Errors, which stop the assembly, are still reported.

Generate Object File in Spite of Errors: -Z

After an error message, normally produces no output. If for some reason you are interested in object file output even after gives an error message on your program, use the `-Z' option. If there are any errors, continues anyways, and writes an object file after a final warning message of the form `n errors, m warnings, generating bad object file.'

Syntax

This chapter describes the machine-independent syntax allowed in a source file. syntax is similar to what many other assemblers use; it is inspired by the BSD 4.2 assembler.

Preprocessing

The internal preprocessor:

- adjusts and removes extra whitespace. It leaves one space or tab before the keywords on a line, and turns any other whitespace on the line into a single space.
- removes all comments, replacing them with a single space, or an appropriate number of newlines.
- converts character constants into the appropriate numeric values.

It does not do macro processing, include file handling, or anything else you may get from your C compiler's preprocessor. You can do include file processing with the `.include` directive (see section `.include "file"`). You can use the GNU C compiler driver to get other "CPP" style preprocessing, by giving the input file a `.S' suffix. See section 'Options Controlling the Kind of Output' in Using GNU CC.

Excess whitespace, comments, and character constants cannot be used in the portions of the input text that are not preprocessed.

If the first line of an input file is `#NO_APP` or if you use the `-f' option, whitespace and comments are not removed from the input file. Within an input file, you can ask for whitespace and comment removal in specific portions of the by putting a line that says `#APP` before the text that may contain whitespace or comments, and putting a line that says `#NO_APP` after this text. This feature is mainly intend to support `asm` statements in compilers whose output is otherwise free of comments and whitespace.

Whitespace

Whitespace is one or more blanks or tabs, in any order. Whitespace is used to separate symbols, and to make programs neater for people to read. Unless within character constants (see section Character Constants), any whitespace means the same as exactly one space.

Comments

There are two ways of rendering comments to . In both cases the comment is equivalent to one space.

Anything from `/*' through the next `*/' is a comment. This means you may not nest these comments.

```
/*
  The only way to include a newline ('\n') in a comment
  is to use this sort of comment.
*/

/* This sort of comment does not nest. */
```

Anything from the line comment character to the next newline is considered a comment and is ignored. The line comment character is see @xref{Machine Dependencies}.

To be compatible with past assemblers, lines that begin with `#' have a special interpretation. Following the `#' should be an absolute expression (see section Expressions): the logical line number of the *next* line. Then a string (see section Strings) is allowed: if present it is a new logical file name. The rest of the line, if any, should be whitespace.

If the first non-whitespace characters on the line are not numeric, the line is ignored. (Just like a comment.)

```
                            # This is an ordinary comment.
# 42-6 "new_file_name"      # New logical file name
                            # This is logical line # 36.
```

This feature is deprecated, and may disappear from future versions of .

Symbols

A symbol is one or more characters chosen from the set of all letters (both upper and lower case), digits and the three characters `_.$'. No symbol may begin with a digit. Case is significant. There is no length limit: all characters are significant. Symbols are delimited by characters not in that set, or by the beginning of a file (since the source program must end with a newline, the end of a file is not a possible symbol delimiter). See section Symbols.

Statements

A statement ends at a newline character (`\n') or at a semicolon (`;'). The newline or semicolon is considered part of the preceding statement. Newlines and semicolons within character constants are an exception: they do not end statements.

It is an error to end any statement with end-of-file: the last character of any input file should be a newline.

You may write a statement on more than one line if you put a backslash (\) immediately in front of any newlines within the statement. When reads a backslashed newline both characters are ignored. You can even put backslashed newlines in the middle of symbol names without changing the meaning of your source program.

An empty statement is allowed, and may include whitespace. It is ignored.

A statement begins with zero or more labels, optionally followed by a key symbol which determines what kind of statement it is. The key symbol determines the syntax of the rest of the statement. If the symbol begins with a dot `.' then the statement is an assembler directive: typically valid for any computer. If the symbol begins with a letter the statement is an assembly language instruction: it assembles into a machine language instruction.

A label is a symbol immediately followed by a colon (`:`). Whitespace before a label or after a colon is permitted, but you may not have whitespace between a label's symbol and its colon. See section Labels.

```
label:     .directive    followed by something
another_label:           # This is an empty statement.
           instruction   operand_1, operand_2, ...
```

Constants

A constant is a number, written so that its value is known by inspection, without knowing any context. Like this:

```
.byte  74, 0112, 092, 0x4A, 0X4a, 'J, '\J # All the same value.
.ascii "Ring the bell\7"                  # A string constant.
.octa  0x123456789abcdef0123456789ABCDEF0 # A bignum.
.float 0f-314159265358979323846264338327\
95028841971.693993751E-40                 # - pi, a flonum.
```

Character Constants

There are two kinds of character constants. A character stands for one character in one byte and its value may be used in numeric expressions. String constants (properly called string *literals*) are potentially many bytes and their values may not be used in arithmetic expressions.

Strings

A string is written between double-quotes. It may contain double-quotes or null characters. The way to get special characters into a string is to escape these characters: precede them with a backslash `\' character. For example `\\' represents one backslash: the first \ is an escape which tells to interpret the second character literally as a backslash (which prevents from recognizing the second \ as an escape character). The complete list of escapes follows.

\b
: Mnemonic for backspace; for ASCII this is octal code 010.

\f
: Mnemonic for FormFeed; for ASCII this is octal code 014.

\n
: Mnemonic for newline; for ASCII this is octal code 012.

\r
: Mnemonic for carriage-Return; for ASCII this is octal code 015.

\t
: Mnemonic for horizontal Tab; for ASCII this is octal code 011.

\ digit digit digit
: An octal character code. The numeric code is 3 octal digits. For compatibility with other Unix systems, 8 and 9 are accepted as digits: for example, `\008` has the value 010, and `\009` the value 011.

\x hex-digits...
: A hex character code. All trailing hex digits are combined. Either upper or lower case `x` works.

\\
: Represents one `\' character.

\"
: Represents one `"' character. Needed in strings to represent this character, because an unescaped `"' would end the string.

\ anything-else
: Any other character when escaped by \ gives a warning, but assembles as if the `\' was not present. The idea is that if you used an escape sequence you clearly didn't want the literal interpretation of the following character. However has no other interpretation, so knows it is giving you the wrong code and warns you of the fact.

Which characters are escapable, and what those escapes represent, varies widely among assemblers. The current set is what we think the BSD 4.2 assembler recognizes, and is a subset of what most C compilers recognize. If you are in doubt, do not use an escape sequence.

Characters

A single character may be written as a single quote immediately followed by that character. The same

escapes apply to characters as to strings. So if you want to write the character backslash, you must w
'\\ where the first \ escapes the second \. As you can see, the quote is an acute accent, not a grave a
A newline (or semicolon `;') immediately following an acute accent is taken as a literal character and
does not count as the end of a statement. The value of a character constant in a numeric expression is the machine's byte-wide code for that character. assumes your character code is ASCII: 'A means 65, 'B means 66, and so on.

Number Constants

distinguishes three kinds of numbers according to how they are stored in the target machine. *Integers* are numbers that would fit into an `int` in the C language. *Bignums* are integers, but they are stored in more than 32 bits. *Flonums* are floating point numbers, described below.

Integers

A binary integer is `0b' or `0B' followed by zero or more of the binary digits `01'.

An octal integer is `0' followed by zero or more of the octal digits (`01234567').

A decimal integer starts with a non-zero digit followed by zero or more digits (`0123456789').

A hexadecimal integer is `0x' or `0X' followed by one or more hexadecimal digits chosen from `0123456789abcdefABCDEF'.

Integers have the usual values. To denote a negative integer, use the prefix operator `-' discussed under expressions (see section Prefix Operator).

Bignums

A bignum has the same syntax and semantics as an integer except that the number (or its negative) takes more than 32 bits to represent in binary. The distinction is made because in some places integers are permitted while bignums are not.

Flonums

A flonum represents a floating point number. The translation is indirect: a decimal floating point number from the text is converted by to a generic binary floating point number of more than sufficient precision. This generic floating point number is converted to a particular computer's floating point format (or formats) by a portion of specialized to that computer.

A flonum is written by writing (in order)

- The digit `0'.
- A letter, to tell the rest of the number is a flonum.
- An optional sign: either `+' or `-'.
- An optional integer part: zero or more decimal digits.
- An optional fractional part: `.' followed by zero or more decimal digits.

- An optional exponent, consisting of:
  - An `E' or `e'.
  - Optional sign: either `+' or `-'.
  - One or more decimal digits.

At least one of the integer part or the fractional part must be present. The floating point number has the usual base-10 value.

does all processing using integers. Flonums are computed independently of any floating point hardware in the computer running .

into a field whose width depends on which assembler directive has the bit-field as its argument. Overflow (a result from the bitwise and requiring more binary digits to represent) is not an error; instead, more constants are generated, of the specified width, beginning with the least significant digits.

The directives `.byte`, `.hword`, `.int`, `.long`, `.short`, and `.word` accept bit-field arguments.

Sections and Relocation

Background

Roughly, a section is a range of addresses, with no gaps; all data "in" those addresses is treated the same for some particular purpose. For example there may be a "read only" section.

The linker reads many object files (partial programs) and combines their contents to form a runnable program. When emits an object file, the partial program is assumed to start at address 0. assigns the final addresses for the partial program, so that different partial programs do not overlap. This is actually an oversimplification, but it suffices to explain how uses sections.

moves blocks of bytes of your program to their run-time addresses. These blocks slide to their run-time addresses as rigid units; their length does not change and neither does the order of bytes within them. Such a rigid unit is called a *section*. Assigning run-time addresses to sections is called relocation. It includes the task of adjusting mentions of object-file addresses so they refer to the proper run-time addresses.

An object file written by has at least three sections, any of which may be empty. These are named text, data and bss sections.

can also generate whatever other named sections you specify using the `.section' directive (see section `.section name`). If you do not use any directives that place output in the `.text' or `.data' sections, these sections still exist, but are empty.

can also generate whatever other named sections you specify using the `.space' and `.subspace' directives. See HP9000 Series 800 Assembly Language Reference Manual (HP 92432-90001) for details on the `.space' and `.subspace' assembler directives.

Within the object file, the text section starts at address `0`, the data section follows, and the bss section follows the data section.

To let know which data changes when the sections are relocated, and how to change that data, also writes to the object file details of the relocation needed. To perform relocation must know, each time an address in the object file is mentioned:

- Where in the object file is the beginning of this reference to an address?
- How long (in bytes) is this reference?
- Which section does the address refer to? What is the numeric value of

  ```
  (address) - (start-address of section)?
  ```
- Is the reference to an address "Program-Counter relative"?

In fact, every address ever uses is expressed as

```
(section) + (offset into section)
```

Further, most expressions computes have this section-relative nature.

In this manual we use the notation {secname N} to mean "offset N into section secname."

Apart from text, data and bss sections you need to know about the absolute section. When mixes partial programs, addresses in the absolute section remain unchanged. For example, address `{absolute 0}` is "relocated" to run-time address 0 by . Although the linker never arranges two partial programs' data sections with overlapping addresses after linking, *by definition* their absolute sections must overlap. Address `{absolute 239}` in one part of a program is always the same address when the program is running as address `{absolute 239}` in any other part of the program.

The idea of sections is extended to the undefined section. Any address whose section is unknown at assembly time is by definition rendered {undefined U}--where U is filled in later. Since numbers are always defined, the only way to generate an undefined address is to mention an undefined symbol. A reference to a named common block would be such a symbol: its value is unknown at assembly time so it has section *undefined*.

By analogy the word *section* is used to describe groups of sections in the linked program. puts all partial programs' text sections in contiguous addresses in the linked program. It is customary to refer to the *text section* of a program, meaning all the addresses of all partial programs' text sections. Likewise for data and bss sections.

Some sections are manipulated by ; others are invented for use of and have no meaning except during assembly.

Linker Sections

deals with just four kinds of sections, summarized below.

These sections hold your program. and treat them as separate but equal sections. Anything you can say of

one section is true another.

bss section

This section contains zeroed bytes when your program begins running. It is used to hold unitialized variables or common storage. The length of each partial program's bss section is important, but because it starts out containing zeroed bytes there is no need to store explicit zero bytes in the object file. The bss section was invented to eliminate those explicit zeros from object files.

absolute section

Address 0 of this section is always "relocated" to runtime address 0. This is useful if you want to refer to an address that must not change when relocating. In this sense we speak of absolute addresses being "unrelocatable": they do not change during relocation.

undefined section

This "section" is a catch-all for address references to objects not in the preceding sections.

An idealized example of three relocatable sections follows. Memory addresses are on the horizontal axis.

Assembler Internal Sections

These sections are meant only for the internal use of . They have no meaning at run-time. You do not really need to know about these sections for most purposes; but they can be mentioned in warning messages, so it might be helpful to have an idea of their meanings to . These sections are used to permit the value of every expression in your assembly language program to be a section-relative address.

ASSEMBLER-INTERNAL-LOGIC-ERROR!

An internal assembler logic error has been found. This means there is a bug in the assembler.

expr section

The assembler stores complex expression internally as combinations of symbols. When it needs to represent an expression as a symbol, it puts it in the expr section.

Sub-Sections

fall into two sections: text and data. You may have separate groups of data in named sections that you want to end up near to each other in the object file, even though they are not contiguous in the assembler source. allows you to use subsections for this purpose. Within each section, there can be numbered subsections with values from 0 to 8192. Objects assembled into the same subsection go into the object file together with other objects in the same subsection. For example, a compiler might want to store constants in the text section, but might not want to have them interspersed with the program being assembled. In this case, the compiler could issue a `.text 0' before each section of code being output, and a `.text 1' before each group of constants being output.

Subsections are optional. If you do not use subsections, everything goes in subsection number zero.

Subsections appear in your object file in numeric order, lowest numbered to highest. (All this to be compatible with other people's assemblers.) The object file contains no representation of subsections; and

other programs that manipulate object files see no trace of them. They just see all your text subsections as a text section, and all your data subsections as a data section.

To specify which subsection you want subsequent statements assembled into, use a numeric argument to specify it, in a `.text expression' or a `.data expression' statement. You can also use an extra subsection argument with arbitrary named sections: `.section name, expression'. Expression should be an absolute expression. (See section Expressions.) If you just say `.text' then `.text 0' is assumed. Likewise `.data' means `.data 0'. Assembly begins in `text 0'. For instance:

```
.text 0     # The default subsection is text 0 anyway.
.ascii "This lives in the first text subsection. *"
.text 1
.ascii "But this lives in the second text subsection."
.data 0
.ascii "This lives in the data section,"
.ascii "in the first data subsection."
.text 0
.ascii "This lives in the first text section,"
.ascii "immediately following the asterisk (*)."
```

Each section has a location counter incremented by one for every byte assembled into that section. Because subsections are merely a convenience restricted to there is no concept of a subsection location counter. There is no way to directly manipulate a location counter--but the `.align` directive changes it, and any label definition captures its current value. The location counter of the section where statements are being assembled is said to be the active location counter.

bss Section

The bss section is used for local common variable storage. You may allocate address space in the bss section, but you may not dictate data to load into it before your program executes. When your program starts running, all the contents of the bss section are zeroed bytes.

The `.lcomm` pseudo-op defines a symbol in the bss section; see section `.lcomm symbol, length`.

The `.comm` pseudo-op may be used to declare a common symbol, which is another form of uninitialized symbol; see See section `.comm symbol, length` .

Symbols

Symbols are a central concept: the programmer uses symbols to name things, the linker uses symbols to link, and the debugger uses symbols to debug.

> *Warning:* does not place symbols in the object file in the same order they were declared. This may break some debuggers.

Labels

A label is written as a symbol immediately followed by a colon `:'. The symbol then represents the current value of the active location counter, and is, for example, a suitable instruction operand. You are warned if you use the same symbol to represent two different locations: the first definition overrides any other definitions.

Giving Symbols Other Values

A symbol can be given an arbitrary value by writing a symbol, followed by an equals sign `=', followed by an expression (see section Expressions). This is equivalent to using the `.set` directive. See section `.set symbol, expression`.

Symbol Names

Symbol names begin with a letter or with one of `._'. On most machines, you can also use `$` in symbol names; exceptions are noted in @xref{Machine Dependencies}. That character may be followed by any string of digits, letters, dollar signs (unless otherwise noted in @xref{Machine Dependencies}), and underscores.

Case of letters is significant: `foo` is a different symbol name than `Foo`.

Each symbol has exactly one name. Each name in an assembly language program refers to exactly one symbol. You may use that symbol name any number of times in a program.

Local Symbol Names

Local symbols help compilers and programmers use names temporarily. There are ten local symbol names, which are re-used throughout the program. You may refer to them using the names `0' `1' ... `9'. To define a local symbol, write a label of the form `**N**:' (where **N** represents any digit). To refer to the most recent previous definition of that symbol write `**N**b', using the same digit as when you defined the label. To refer to the next definition of a local label, write `**N**f'---where **N** gives you a choice of 10 forward references. The `b' stands for "backwards" and the `f' stands for "forwards".

Local symbols are not emitted by the current GNU C compiler.

There is no restriction on how you can use these labels, but remember that at any point in the assembly you can refer to at most 10 prior local labels and to at most 10 forward local labels.

Local symbol names are only a notation device. They are immediately transformed into more conventional symbol names before the assembler uses them. The symbol names stored in the symbol table, appearing in error messages and optionally emitted to the object file have these parts:

`L`

All local labels begin with `L'. Normally both and forget symbols that start with `L'. These labels are used for symbols you are never intended to see. If you use the `-L' option then retains these symbols in the object file. If you also instruct to retain these symbols, you may use them in debugging.

`digit`

If the label is written `0:' then the digit is `0'. If the label is written `1:' then the digit is `1'. And so on up through `9:'.

`C-A`

This unusual character is included so you do not accidentally invent a symbol of the same name. The character has ASCII value `\001'.

`ordinal number`

This is a serial number to keep the labels distinct. The first `0:' gets the number `1'; The 15th `0:' gets the number `15'; *etc.*. Likewise for the other labels `1:' through `9:'.

For instance, the first `1:` is named `L1C-A1`, the 44th `3:` is named `L3C-A44`.

The Special Dot Symbol

The special symbol `.' refers to the current address that is assembling into. Thus, the expression `melvin: .long .' defines `melvin` to contain its own address. Assigning a value to `.` is treated the same as a `.org` directive. Thus, the expression `.=.+4' is the same as saying `.space 4'.

Symbol Attributes

Every symbol has, as well as its name, the attributes "Value" and "Type". Depending on output format, symbols can also have auxiliary attributes.

If you use a symbol without defining it, assumes zero for all these attributes, and probably won't warn you. This makes the symbol an externally defined symbol, which is generally what you would want.

Value

The value of a symbol is (usually) 32 bits. For a symbol which labels a location in the text, data, bss or absolute sections the value is the number of addresses from the start of that section to the label. Naturally for text, data and bss sections the value of a symbol changes as changes section base addresses during linking. Absolute symbols' values do not change during linking: that is why they are called absolute.

The value of an undefined symbol is treated in a special way. If it is 0 then the symbol is not defined in this assembler source file, and tries to determine its value from other files linked into the same program. You make this kind of symbol simply by mentioning a symbol name without defining it. A non-zero value represents a `.comm` common declaration. The value is how much common storage to reserve, in bytes (addresses). The symbol refers to the first address of the allocated storage.

Type

The type attribute of a symbol contains relocation (section) information, any flag settings indicating that a symbol is external, and (optionally), other information for linkers and debuggers. The exact format depends on the object-code output format in use.

Symbol Attributes: `a.out`

Descriptor

This is an arbitrary 16-bit value. You may establish a symbol's descriptor value by using a `.desc` statement (@xref{Desc,,`.desc`}). A descriptor value means nothing to .

Other

This is an arbitrary 8-bit value. It means nothing to .

Expressions

An expression specifies an address or numeric value. Whitespace may precede and/or follow an expression.

The result of an expression must be an absolute number, or else an offset into a particular section. If an expression is not absolute, and there is not enough information when sees the expression to know its section, a second pass over the source program might be necessary to interpret the expression--but the second pass is currently not implemented. aborts with an error message in this situation.

Empty Expressions

An empty expression has no value: it is just whitespace or null. Wherever an absolute expression is required, you may omit the expression, and assumes a value of (absolute) 0. This is compatible with other assemblers.

Integer Expressions

An integer expression is one or more *arguments* delimited by *operators*.

Arguments

Arguments are symbols, numbers or subexpressions. In other contexts arguments are sometimes called "arithmetic operands". In this manual, to avoid confusing them with the "instruction operands" of the machine language, we use the term "argument" to refer to parts of expressions only, reserving the word "operand" to refer only to machine instruction operands.

Symbols are evaluated to yield {section NNN} where section is one of text, data, bss, absolute, or undefined. NNN is a signed, 2's complement 32 bit integer.

Numbers are usually integers.

A number can be a flonum or bignum. In this case, you are warned that only the low order 32 bits are used, and pretends these 32 bits are an integer. You may write integer-manipulating instructions that act on exotic constants, compatible with other assemblers.

Subexpressions are a left parenthesis `(' followed by an integer expression, followed by a right parenthesis `)'; or a prefix operator followed by an argument.

Operators

Operators are arithmetic functions, like + or %. Prefix operators are followed by an argument. Infix operators appear between their arguments. Operators may be preceded and/or followed by whitespace.

Prefix Operator

has the following prefix operators. They each take one argument, which must be absolute.

-

Negation. Two's complement negation.

~

Complementation. Bitwise not.

Infix Operators

Infix operators take two arguments, one on either side. Operators have precedence, but operations with equal precedence are performed left to right. Apart from + or -, both arguments must be absolute, and the result is absolute.

1. Highest Precedence

   *

   Multiplication.

`/`
: Division. Truncation is the same as the C operator `/'

`%`
: Remainder.

`<`

`<<`
: Shift Left. Same as the C operator `<<'.

`>`

`>>`
: Shift Right. Same as the C operator `>>'.

2. Intermediate precedence

   `|`
   : Bitwise Inclusive Or.

   `&`
   : Bitwise And.

   `^`
   : Bitwise Exclusive Or.

   `!`
   : Bitwise Or Not.

3. Lowest Precedence

   `+`
   : Addition. If either argument is absolute, the result has the section of the other argument. You may not add together arguments from different sections.

   `-`
   : Subtraction. If the right argument is absolute, the result has the section of the left argument. If both arguments are in the same section, the result is absolute. You may not subtract arguments from different sections.

In short, it's only meaningful to add or subtract the *offsets* in an address; you can only have a defined section in one of the two arguments.

Assembler Directives

All assembler directives have names that begin with a period (`.'). The rest of the name is letters, usually in lower case.

This chapter discusses directives that are available regardless of the target machine configuration for the GNU assembler.

.abort

This directive stops the assembly immediately. It is for compatibility with other assemblers. The original idea was that the assembly language source would be piped into the assembler. If the sender of the source quit, it could use this directive tells to quit also. One day `.abort` will not be supported.

.align abs-expr, abs-expr, abs-expr

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment required, as described below.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The way the required alignment is specified varies from system to system. For the a29k, hppa, m68k, m88k, w65, sparc, and Hitachi SH, and i386 using ELF format, the first expression is the alignment request in bytes. For example `.align 8' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

For other systems, including the i386 using a.out format, it is the number of low-order zero bits the location counter must have after advancement. For example `.align 3' advances the location counter until it a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

This inconsistency is due to the different behaviors of the various native assemblers for these systems which GAS must emulate. GAS also provides `.balign` and `.p2align` directives, described later, which have a consistent behavior across all architectures (but are specific to GAS).

.app-file string

`.app-file` (which may also be spelled `.file') tells that we are about to start a new logical file. string is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes `"'; but if you wish to specify an empty file name is permitted, you must give the quotes--`""'. This statement may go away in future: it is only recognized to be compatible with old programs.

.ascii "string"...

`.ascii` expects zero or more string literals (see section Strings) separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

.asciz "string"...

`.asciz` is just like `.ascii`, but each string is followed by a zero byte. The "z" in `.asciz' stands for "zero".

.balign[wl] abs-expr, abs-expr, abs-expr

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment request in bytes. For example `.balign 8' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The `.balignw` and `.balignl` directives are variants of the `.balign` directive. The `.balignw` directive treats the fill pattern as a two byte word value. The `.balignl` directives treats the fill pattern as a four byte longword value. For example, `.balignw 4,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value 0x368d (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

.byte expressions

`.byte` expects zero or more expressions, separated by commas. Each expression is assembled into the next byte.

.comm symbol , length

`.comm` declares a common symbol named symbol. When linking, a common symbol in one object file may be merged with a defined or common symbol of the same name in another object file. If does not see a definition for the symbol--just one or more common symbols--then it will allocate length bytes of uninitialized memory. length must be an absolute expression. If sees multiple common symbols with the same name, and they do not all have the same size, it will allocate space using the largest size.

.data subsection

`.data` tells to assemble the following statements onto the end of the data subsection numbered subsection (which is an absolute expression). If subsection is omitted, it defaults to zero.

.double flonums

`.double` expects zero or more flonums, separated by commas. It assembles floating point numbers.

.eject

Force a page break at this point, when generating assembly listings.

.else

`.else` is part of the support for conditional assembly; see section `.if absolute expression`. It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

.endif

`.endif` is part of the support for conditional assembly; it marks the end of a block of code that is only assembled conditionally. See section `.if absolute expression`.

.equ symbol, expression

This directive sets the value of symbol to expression. It is synonymous with `.set'; see section .set symbol, expression.

.equiv symbol, expression

The `.equiv` directive is like `.equ` and `.set`, except that the assembler will signal an error if symbol is already defined.

Except for the contents of the error message, this is roughly equivalent to

```
.ifdef SYM
.err
.endif
.equ SYM,VAL
```

.err

If assembles a `.err` directive, it will print an error message and, unless the `-Z` option was used, it will not generate an object file. This can be used to signal error an conditionally compiled code.

.extern

`.extern` is accepted in the source program--for compatibility with other assemblers--but it is ignored. treats all undefined symbols as external.

.file string

`.file` (which may also be spelled `.app-file') tells that we are about to start a new logical file. string is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes `"'; but if you wish to specify an empty file name, you must give the quotes--`""'. This statement may go away in future: it is only recognized to be compatible with old programs.

.fill repeat , size , value

result, size and value are absolute expressions. This emits repeat copies of size bytes. Repeat may be zero or more. Size may be zero or more, but if it is more than 8, then it is deemed to have the value 8, compatible with other people's assemblers. The contents of each repeat bytes is taken from an 8-byte number. The highest order 4 bytes are zero. The lowest order 4 bytes are value rendered in the byte-order of an integer on the computer is assembling for. Each size bytes in a repetition is taken from the lowest

order size bytes of this number. Again, this bizarre behavior is compatible with other people's assemblers.

size and value are optional. If the second comma and value are absent, value is assumed zero. If the first comma and following tokens are absent, size is assumed to be 1.

.float flonums

This directive assembles zero or more flonums, separated by commas. It has the same effect as `.single`.

.global symbol, .globl symbol

`.global` makes the symbol visible to . If you define symbol in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, symbol takes its attributes from a symbol of the same name from another file linked into the same program.

Both spellings (`.globl' and `.global') are accepted, for compatibility with other assemblers.

.hword expressions

This expects zero or more expressions, and emits a 16 bit number for each.

.ident

This directive is used by some assemblers to place tags in object files. simply accepts the directive for source-file compatibility with such assemblers, but does not actually emit anything for it.

.if absolute expression

`.if` marks the beginning of a section of code which is only considered part of the source program being assembled if the argument (which must be an absolute expression) is non-zero. The end of the conditional section of code must be marked by `.endif` (see section `.endif`); optionally, you may include code for the alternative condition, flagged by `.else` (see section `.else`).

The following variants of `.if` are also supported:

`.ifdef symbol`

Assembles the following section of code if the specified symbol has been defined.

`.ifndef symbol`

`.ifnotdef symbol`

Assembles the following section of code if the specified symbol has not been defined. Both

spelling variants are equivalent.

.include "file"

This directive provides a way to include supporting files at specified points in your source program. The code from file is assembled as if it followed the point of the `.include`; when the end of the included file is reached, assembly of the original file continues. You can control the search paths used with the `-I' command-line option (see section Command-Line Options). Quotation marks are required around file.

.int expressions

Expect zero or more expressions, of any section, separated by commas. For each expression, emit a number that, at run time, is the value of that expression. The byte order and bit size of the number depends on what kind of target the assembly is for.

.irp symbol,values…

Evaluate a sequence of statements assigning different values to symbol. The sequence of statements starts at the `.irp` directive, and is terminated by an `.endr` directive. For each value, symbol is set to value, and the sequence of statements is assembled. If no value is listed, the sequence of statements is assembled once, with symbol set to the null string. To refer to symbol within the sequence of statements, use \symbol.

For example, assembling

```
        .irp    param,1,2,3
        move    d\param,sp@-
        .endr
```

is equivalent to assembling

```
        move    d1,sp@-
        move    d2,sp@-
        move    d3,sp@-
```

.irpc symbol,values…

Evaluate a sequence of statements assigning different values to symbol. The sequence of statements starts at the `.irpc` directive, and is terminated by an `.endr` directive. For each character in value, symbol is set to the character, and the sequence of statements is assembled. If no value is listed, the sequence of statements is assembled once, with symbol set to the null string. To refer to symbol within the sequence of statements, use \symbol.

For example, assembling

```
        .irpc    param,123
        move    d\param,sp@-
        .endr
```

is equivalent to assembling

```
        move    d1,sp@-
        move    d2,sp@-
        move    d3,sp@-
```

.lcomm symbol , length

Reserve length (an absolute expression) bytes for a local common denoted by symbol. The section and value of symbol are those of the new local common. The addresses are allocated in the bss section, so that at run-time the bytes start off zeroed. Symbol is not declared global (see section `.global symbol`, `.globl symbol`), so is normally not visible to .

.lflags

accepts this directive, for compatibility with other assemblers, but ignores it.

.line line-number

Even though this is a directive associated with the `a.out` or `b.out` object-code formats, still recognizes it when producing COFF output, and treats `.line' as though it were the COFF `.ln' *if* it is found outside a `.def`/`.endef` pair.

Inside a `.def`, `.line' is, instead, one of the directives used by compilers to generate auxiliary symbol information for debugging.

.linkonce [type]

Mark the current section so that the linker only includes a single copy of it. This may be used to include the same section in several different object files, but ensure that the linker will only include it once in the final output file. The `.linkonce` pseudo-op must be used for each instance of the section. Duplicate sections are detected based on the section name, so it should be unique.

This directive is only supported by a few object file formats; as of this writing, the only object file format which supports it is the Portable Executable format used on Windows NT.

The type argument is optional. If specified, it must be one of the following strings. For example:

`.linkonce same_size`

Not all types may be supported on all object file formats.

`discard`

Silently discard duplicate sections. This is the default.

`one_only`

Warn if there are duplicate sections, but still keep only one copy.

`same_size`

Warn if any of the duplicates have different sizes.

`same_contents`

Warn if any of the duplicates do not have exactly the same contents.

`.ln line-number`

`.ln' is a synonym for `.line'.

`.mri val`

If val is non-zero, this tells to enter MRI mode. If val is zero, this tells to exit MRI mode. This change affects code assembled until the next `.mri` directive, or until the end of the file. See section Assemble in MRI Compatibility Mode: `-M`.

`.list`

Control (in conjunction with the `.nolist` directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). `.list` increments the counter, and `.nolist` decrements it. Assembly listings are generated whenever the counter is greater than zero.

By default, listings are disabled. When you enable them (with the `-a' command line option; see section Command-Line Options), the initial value of the listing counter is one.

`.long expressions`

`.long` is the same as `.int', see section `.int expressions`.

.macro

The commands `.macro` and `.endm` allow you to define macros that generate assembly output. For example, this definition specifies a macro `sum` that puts a sequence of numbers into memory:

```
        .macro  sum from=0, to=5
        .long   \from
        .if     \to-\from
        sum     "(\from+1)",\to
        .endif
        .endm
```

With that definition, `SUM 0,5' is equivalent to this assembly input:

```
        .long   0
        .long   1
        .long   2
        .long   3
        .long   4
        .long   5
```

`.macro macname`

`.macro macname macargs ...`

Begin the definition of a macro called macname. If your macro definition requires arguments, specify their names after the macro name, separated by commas or spaces. You can supply a default value for any macro argument by following the name with `=deflt'. For example, these are all valid `.macro` statements:

`.macro comm`

Begin the definition of a macro called `comm`, which takes no arguments.

`.macro plus1 p, p1`

`.macro plus1 p p1`

Either statement begins the definition of a macro called `plus1`, which takes two arguments; within the macro definition, write `\p' or `\p1' to evaluate the arguments.

`.macro reserve_str p1=0 p2`

Begin the definition of a macro called `reserve_str`, with two arguments. The first argument has a default value, but not the second. After the definition is complete, you can call the macro either as `reserve_str a,b' (with `\p1' evaluating to a and `\p2' evaluating to b), or as `reserve_str ,b' (with `\p1' evaluating as the default, in this case `0', and `\p2' evaluating to b).

When you call a macro, you can specify the argument values either by position, or by keyword. For example, `sum 9,17' is equivalent to `sum to=17, from=9'.

- .endm Mark the end of a macro definition.

- .exitm Exit early from the current macro definition.
- \@ maintains a counter of how many macros it has executed in this pseudo-variable; you can copy that number to your output with `\@', but *only within a macro definition*.

.nolist

Control (in conjunction with the `.list` directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). `.list` increments the counter, and `.nolist` decrements it. Assembly listings are generated whenever the counter is greater than zero.

.octa bignums

This directive expects zero or more bignums, separated by commas. For each bignum, it emits a 16-byte integer.

The term "octa" comes from contexts in which a "word" is two bytes; hence *octa*-word for 16 bytes.

.org new-lc , fill

Advance the location counter of the current section to new-lc. new-lc is either an absolute expression or an expression with the same section as the current subsection. That is, you can't use `.org` to cross sections: if new-lc has the wrong section, the `.org` directive is ignored. To be compatible with former assemblers, if the section of new-lc is absolute, issues a warning, then pretends the section of new-lc is the same as the current subsection.

`.org` may only increase the location counter, or leave it unchanged; you cannot use `.org` to move the location counter backwards.

Because tries to assemble programs in one pass, new-lc may not be undefined. If you really detest this restriction we eagerly await a chance to share your improved assembler.

Beware that the origin is relative to the start of the section, not to the start of the subsection. This is compatible with other people's assemblers.

When the location counter (of the current subsection) is advanced, the intervening bytes are filled with fill which should be an absolute expression. If the comma and fill are omitted, fill defaults to zero.

.p2align[wl] abs-expr, abs-expr, abs-expr

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the number of low-order zero bits the location counter must have after advancement. For example `.p2align 3' advances the location counter until it a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and th comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The `.p2alignw` and `.p2alignl` directives are variants of the `.p2align` directive. The `.p2alignw` directive treats the fill pattern as a two byte word value. The `.p2alignl` directives treats the fill pattern as a four byte longword value. For example, `.p2alignw 2,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value 0x368d (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

`.psize lines , columns`

Use this directive to declare the number of lines--and, optionally, the number of columns--to use for each page, when generating listings.

If you do not use `.psize`, listings use a default line-count of 60. You may omit the comma and columns specification; the default width is 200 columns.

generates formfeeds whenever the specified number of lines is exceeded (or whenever you explicitly request one, using `.eject`).

If you specify lines as `0`, no formfeeds are generated save those explicitly specified with `.eject`.

`.quad bignums`

`.quad` expects zero or more bignums, separated by commas. For each bignum, it emits an 8-byte integer. If the bignum won't fit in 8 bytes, it prints a warning message; and just takes the lowest order 8 bytes of the bignum.

The term "quad" comes from contexts in which a "word" is two bytes; hence *quad*-word for 8 bytes.

`.rept count`

Repeat the sequence of lines between the `.rept` directive and the next `.endr` directive count times.

For example, assembling

```
        .rept   3
```


```
        .long   0
        .endr
```

is equivalent to assembling

```
        .long   0
        .long   0
        .long   0
```

.sbttl "subheading"

Use subheading as the title (third line, immediately after the title line) when generating assembly listings.

This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

.section name

Use the `.section` directive to assemble the following code into a section named name.

This directive is only supported for targets that actually support arbitrarily named sections; on `a.out` targets, for example, it is not accepted, even with a standard `a.out` section name.

.set symbol, expression

Set the value of symbol to expression. This changes symbol's value and type to conform to expression. If symbol was flagged as external, it remains flagged (see section Symbol Attributes).

You may `.set` a symbol many times in the same assembly.

If you `.set` a global symbol, the value stored in the object file is the last value stored into it.

.short expressions

.single flonums

This directive assembles zero or more flonums, separated by commas. It has the same effect as `.float`.

.skip size , fill

This directive emits size bytes, each of value fill. Both size and fill are absolute expressions. If the comma and fill are omitted, fill is assumed to be zero. This is the same as `.space'.

.space size , fill

This directive emits size bytes, each of value fill. Both size and fill are absolute expressions. If the comma and fill are omitted, fill is assumed to be zero. This is the same as `.skip'.

.stabd, .stabn, .stabs

There are three directives that begin `.stab'. All emit symbols (see section Symbols), for use by symbolic debuggers. The symbols are not entered in the hash table: they cannot be referenced elsewhere in the source file. Up to five fields are required:

string

This is the symbol's name. It may contain any character except `\000', so is more general than ordinary symbol names. Some debuggers used to code arbitrarily complex structures into symbol names using this field.

type

An absolute expression. The symbol's type is set to the low 8 bits of this expression. Any bit pattern is permitted, but and debuggers choke on silly bit patterns.

other

An absolute expression. The symbol's "other" attribute is set to the low 8 bits of this expression.

desc

An absolute expression. The symbol's descriptor is set to the low 16 bits of this expression.

value

An absolute expression which becomes the symbol's value.

If a warning is detected while reading a `.stabd`, `.stabn`, or `.stabs` statement, the symbol has probably already been created; you get a half-formed symbol in your object file. This is compatible with earlier assemblers!

`.stabd type , other , desc`

The "name" of the symbol generated is not even an empty string. It is a null pointer, for compatibility. Older assemblers used a null pointer so they didn't waste space in object files with empty strings.

The symbol's value is set to the location counter, relocatably. When your program is linked, the value of this symbol is the address of the location counter when the `.stabd` was assembled.

`.stabn type , other , desc , value`

The name of the symbol is set to the empty string `""`.

`.stabs string , type , other , desc , value`

All five fields are specified.

.string "str"

Copy the characters in str to the object file. You may specify more than one string to copy, separated by commas. Unless otherwise specified for a particular machine, the assembler marks the end of each string with a 0 byte. You can use any of the escape sequences described in section Strings.

.text subsection

Tells to assemble the following statements onto the end of the text subsection numbered subsection, which is an absolute expression. If subsection is omitted, subsection number zero is used.

.title "heading"

Use heading as the title (second line, immediately after the source file name and pagenumber) when generating assembly listings.

This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

.word expressions

This directive expects zero or more expressions, of any section, separated by commas.

In order to assemble compiler output into something that works, occasionlly does strange things to `.word' directives. Directives of the form `.word sym1-sym2' are often emitted by compilers as part of jump tables. Therefore, when assembles a directive of the form `.word sym1-sym2', and the difference between `sym1` and `sym2` does not fit in 16 bits, creates a secondary jump table, immediately before the next label. This secondary jump table is preceded by a short-jump to the first byte after the secondary table. This short-jump prevents the flow of control from accidentally falling into the new table. Inside the table is a long-jump to `sym2`. The original `.word' contains `sym1` minus the address of the long-jump to `sym2`.

If there were several occurrences of `.word sym1-sym2' before the secondary jump table, all of them are adjusted. If there was a `.word sym3-sym4', that also did not fit in sixteen bits, a long-jump to `sym4` is included in the secondary jump table, and the `.word` directives are adjusted to contain `sym3` minus the address of the long-jump to `sym4`; and so on, for as many entries in the original jump table as necessary.

Deprecated Directives

One day these directives won't work. They are included for compatibility with older assemblers.

`.abort`

`.app-file`

@lowersections

Reporting Bugs

Your bug reports play an essential role in making reliable.

Reporting a bug may help you by bringing a solution to your problem, or it may not. But in any case the principal function of a bug report is to help the entire community by making the next version of work better. Bug reports are your contribution to the maintenance of .

In order for a bug report to serve its purpose, you must include the information that enables us to fix the bug.

Have you found a bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the assembler gets a fatal signal, for any input whatever, that is a bug. Reliable assemblers never crash.
- If produces an error message for valid input, that is a bug.
- If does not produce an error message for invalid input, that is a bug. However, you should note that your idea of "invalid input" might be our idea of "an extension" or "support for traditional practice".
- If you are an experienced user of assemblers, your suggestions for improvement of are welcome in any case.

How to report bugs

A number of companies and individuals offer support for GNU products. If you obtained from a support organization, we recommend you contact that organization first.

You can find contact information for many support companies and individuals in the file `` `etc/SERVICE' `` in the GNU Emacs distribution.

In any event, we also recommend that you send bug reports for to `bug-gnu-utils@prep.ai.mit.edu'.

The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to state a fact or leave it out, state it!

Often people omit facts because they think they know what causes the problem and assume that some details do not matter. Thus, you might assume that the name of a symbol you use in an example does not matter. Well, probably it does not, but one cannot be sure. Perhaps the bug is a stray memory reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name

were different, the contents of that location would fool the assembler into doing the right thing despi
the bug. Play it safe and give a specific, complete example. That is the easiest thing for you to do, an
most helpful.

Keep in mind that the purpose of a bug report is to enable us to fix the bug if it is new to us. Therefore, always write your bug reports on the assumption that the bug has not been reported previously.

Sometimes people give a few sketchy facts and ask, "Does this ring a bell?" Those bug reports are useless, and we urge everyone to *refuse to respond to them* except to chide the sender to report bugs properly.

To enable us to fix the bug, you should include all these things:

- The version of . announces it if you start it with the `--version' argument.

  Without this, we will not know whether there is any point in looking for the bug in the current version of .
- Any patches you may have applied to the source.
- The type of machine you are using, and the operating system name and version number.
- What compiler (and its version) was used to compile ---e.g. "`gcc-2.7`".
- The command arguments you gave the assembler to assemble your example and observe the bug. To guarantee you will not omit something important, list them all. A copy of the Makefile (or the output from make) is sufficient.

  If we were to try to guess the arguments, we would probably guess wrong and then we might not encounter the bug.
- A complete input file that will reproduce the bug. If the bug is observed when the assembler is invoked via a compiler, send the assembler source, not the high level language source. Most compilers will produce the assembler source when run with the `-S' option. If you are using , use the options `-v --save-temps'; this will save the assembler source in a file with an extension of `` `.s' ``, and also show you exactly how is being run.
- A description of what behavior you observe that you believe is incorrect. For example, "It gets a fatal signal."

  Of course, if the bug is that gets a fatal signal, then we will certainly notice it. But if the bug is incorrect output, we might not notice unless it is glaringly wrong. You might as well not give us a chance to make a mistake.

  Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of is out of synch, or you have encountered a bug in the C library on your system. (This has happened!) Your copy might crash and ours would not. If you told us to expect a crash, then when ours fails to crash, we would know that the bug was not happening for us. If you had not told us to expect a crash, then we would not be able to draw any conclusion from our observations.
- If you wish to suggest changes to the source, send us context diffs, as generated by `diff` with the `-u', `-c', or `-p' option. Always send diffs from the old file to the new file. If you even discuss something in the source, refer to it by context, not by line number.

The line numbers in our development sources will not match those in your sources. Your line numbers would convey no useful information to us.

Here are some things that are not necessary:

- A description of the envelope of the bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

  This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. We recommend that you save your time for something else.

  Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience for us. Errors in the output will be easier to spot, running under the debugger will take less time, and so on.

  However, simplification is not vital; if you do not want to do this, report the bug anyway and send us the entire test case you used.

- A patch for the bug.

  A patch for the bug does help us if it is a good one. But do not omit the necessary information, such as the test case, on the assumption that a patch is all we need. We might see problems with your patch and decide to fix the problem another way, or we might not understand it at all.

  Sometimes with a program as complicated as it is very hard to construct an example that will make the program follow a certain path through the code. If you do not send us the example, we will not be able to construct one, so we will not be able to verify that the bug is fixed.

  And if we cannot understand what bug you are trying to fix, or why your patch should be an improvement, we will not install it. A test case will help us to understand.

- A guess about what the bug is or what it depends on.

  Such guesses are usually wrong. Even we cannot guess right about such things without first using the debugger to find the facts.

Acknowledgements

If you have contributed to and your name isn't listed here, it is not meant as a slight. We just don't know about it. Send mail to the maintainer, and we'll correct the situation. Currently the maintainer is Ken Raeburn (email address `raeburn@cygnus.com`).

Dean Elsner wrote the original GNU assembler for the VAX.(1)

Jay Fenlason maintained GAS for a while, adding support for GDB-specific debug information and the 68k series machines, most of the preprocessing pass, and extensive changes in `` `messages.c' ``, `` `input-file.c' ``, `` `write.c' ``.

K. Richard Pixley maintained GAS for a while, adding various enhancements and many bug fixes, including merging support for several processors, breaking GAS up to handle multiple object file formats, adding back ends (including heavy rewrite, testing, an integration of the coff and b.out back ends), adding configuration including heavy testing and verification of cross assemblers and file splits and renaming, converted GAS to strictly ANSI C including full prototypes, added support for m680[34]0 and cpu32, did considerable work on i960 including a COFF port (including considerable amounts of reverse engineering), a SPARC opcode file rewrite, DECstation, rs6000, and hp300hpux host ports, updated "know" assertions and made them work, much other reorganization, cleanup, and lint.

Ken Raeburn wrote the high-level BFD interface code to replace most of the code in format-specific I/O modules.

The original VMS support was contributed by David L. Kashtan. Eric Youngdale has done much work with it since.

The Intel 80386 machine description was written by Eliot Dresselhaus.

Minh Tran-Le at IntelliCorp contributed some AIX 386 support.

The Motorola 88k machine description was contributed by Devon Bowen of Buffalo University and Torbjorn Granlund of the Swedish Institute of Computer Science.

Keith Knowles at the Open Software Foundation wrote the original MIPS back end (`tc-mips.c', `tc-mips.h'), and contributed Rose format support (which hasn't been merged in yet). Ralph Campbell worked with the MIPS code to support a.out format.

Support for the Zilog Z8k and Hitachi H8/300 and H8/500 processors (tc-z8k, tc-h8300, tc-h8500), and IEEE 695 object file format (obj-ieee), was written by Steve Chamberlain of Cygnus Support. Steve also modified the COFF back end to use BFD for some low-level operations, for use with the H8/300 and AMD 29k targets.

John Gilmore built the AMD 29000 support, added `.include` support, and simplified the configuration of which versions accept which directives. He updated the 68k machine description so that Motorola's opcodes always produced fixed-size instructions (e.g. `jsr`), while synthetic instructions remained shrinkable (`jbsr`). John fixed many bugs, including true tested cross-compilation support, and one bug in relaxation that took a week and required the proverbial one-bit fix.

Ian Lance Taylor of Cygnus Support merged the Motorola and MIT syntax for the 68k, completed support for some COFF targets (68k, i386 SVR3, and SCO Unix), added support for MIPS ECOFF and ELF targets, wrote the initial RS/6000 and PowerPC assembler, and made a few other minor patches.

Steve Chamberlain made able to generate listings.

Hewlett-Packard contributed support for the HP9000/300.

Jeff Law wrote GAS and BFD support for the native HPPA object format (SOM) along with a fairly extensive HPPA testsuite (for both SOM and ELF object formats). This work was supported by both the Center for Software Science at the University of Utah and Cygnus Support.

Support for ELF format files has been worked on by Mark Eichin of Cygnus Support (original,

incomplete implementation for SPARC), Pete Hoogenboom and Jeff Law at the University of Utah (HPPA mainly), Michael Meissner of the Open Software Foundation (i386 mainly), and Ken Raeburn Cygnus Support (sparc, and some initial 64-bit support).

Richard Henderson rewrote the Alpha assembler. Klaus Kaempf wrote GAS and BFD support for openVMS/Alpha.

Several engineers at Cygnus Support have also provided many small bug fixes and configuration enhancements.

Many others have contributed large or small bugfixes and enhancements. If you have contributed significant work and are not mentioned on this list, and want to be, let us know. Some of the history has been lost; we are not intentionally leaving anyone out.

Index

#

-

.

:

\

a

b

C

d

e

f

g

h

i

TOC

j

l

m

n

o

p

q

r

s

t

u

v

W

Z

Using

TOC

(1)

Any more details?

AWK Language Programming

A User's Guide for GNU AWK

Edition 1.0

January 1996

Arnold D. Robbins Based on The GAWK Manual, by Robbins, Close, Rubin, and Stallman

Go to the next section.

This is Edition 1.0 of AWK Language Programming,
for the 3.0 (or later) version of the GNU implementation of AWK.

Published by the Free Software Foundation
59 Temple Place -- Suite 330
Boston, MA 02111-1307 USA
Phone: +1-617-542-5942
Fax (including Japan): +1-617-542-2652
Printed copies are available for $25 each.
ISBN 1-882114-26-4

Cover art by Etienne Suvasa.

To Miriam, for making me complete.

To Chana, for the joy you bring us.

To Rivka, for the exponential increase.

@evenheading @thispage **@thistitle** @oddheading **@thischapter** @thispage

Preface

This book teaches you about the `awk` language and how you can use it effectively. You should already be familiar with basic system commands, such as `cat` and `ls`,(1) and basic shell facilities, such as Input/Output (I/O) redirection and pipes.

Implementations of the `awk` language are available for many different computing environments. This book, while describing the `awk` language in general, also describes a particular implementation of `awk` called `gawk` (which stands for "GNU Awk"). `gawk` runs on a broad range of Unix systems, ranging from 80386 PC-based computers, up through large scale systems, such as Crays. `gawk` has also been

ported to MS-DOS and OS/2 PC's, Atari and Amiga micro-computers, and VMS.

History of awk and gawk

The name `awk` comes from the initials of its designers: Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan. The original version of `awk` was written in 1977 at AT&T Bell Laboratories. In 1985 a new version made the programming language more powerful, introducing user-defined functions, multiple input streams, and computed regular expressions. This new version became generally available with Unix System V Release 3.1. The version in System V Release 4 added some new features and also cleaned up the behavior in some of the "dark corners" of the language. The specification for `awk` in the POSIX Command Language and Utilities standard further clarified the language based on feedback from both the `gawk` designers, and the original Bell Labs `awk` designers.

The GNU implementation, `gawk`, was written in 1986 by Paul Rubin and Jay Fenlason, with advice from Richard Stallman. John Woods contributed parts of the code as well. In 1988 and 1989, David Trueman, with help from Arnold Robbins, thoroughly reworked `gawk` for compatibility with the newer `awk`. Current development focuses on bug fixes, performance improvements, standards compliance, and occasionally, new features.

The GNU Project and This Book

The Free Software Foundation (FSF) is a non-profit organization dedicated to the production and distribution of freely distributable software. It was founded by Richard M. Stallman, the author of the original Emacs editor. GNU Emacs is the most widely used version of Emacs today.

The GNU project is an on-going effort on the part of the Free Software Foundation to create a complete, freely distributable, POSIX compliant computing environment. (GNU stands for "GNU's not Unix".) The FSF uses the "GNU General Public License" (or GPL) to ensure that source code for their software is always available to the end user. A copy of the GPL is included for your reference (see section GNU GENERAL PUBLIC LICENSE). The GPL applies to the C language source code for `gawk`.

As of this writing (1995), the only major component of the GNU environment still uncompleted is the operating system kernel, and work proceeds apace on that. A shell, an editor (Emacs), highly portable optimizing C, C++, and Objective-C compilers, a symbolic debugger, and dozens of large and small utilities (such as `gawk`), have all been completed and are freely available.

Until the GNU operating system is released, the FSF recommends the use of Linux, a freely distributable, Unix-like operating system for 80386 and other systems. There are many books on Linux. One freely available one is Linux Installation and Getting Started, by Matt Welsh. Many Linux distributions are available, often in computer stores or bundled on CD-ROM with books about Linux. Also, the FSF provides a Linux distribution ("Debian"); contact them for more information. See section Getting the `gawk` Distribution, for the FSF's contact information. (There are two other freely available, Unix-like operating systems for 80386 and other systems, NetBSD and FreeBSD. Both are based on the 4.4-Lite Berkeley Software Distribution, and both use recent versions of `gawk` for their versions of `awk`.)

This book you are reading now is actually free. The information in it is freely available to anyone, th machine readable source code for the book comes with `gawk`, and anyone may take this book to a copying machine and make as many copies of it as they like. (Take a moment to check the copying permissions on the Copyright page.)

If you paid money for this book, what you actually paid for was the book's nice printing and binding, and the publisher's associated costs to produce it. We have made an effort to keep these costs reasonable; most people would prefer a bound book to over 300 pages of photo-copied text that would then have to be held in a loose-leaf binder (not to mention the time and labor involved in doing the copying). The same is true of producing this book from the machine readable source; the retail price is only slightly more than the cost per page of printing it on a laser printer.

This book itself has gone through several previous, preliminary editions. I started working on a preliminary draft of The GAWK Manual, by Diane Close, Paul Rubin, and Richard Stallman in the fall of 1988. It was around 90 pages long, and barely described the original, "old" version of `awk`. After substantial revision, the first version of the The GAWK Manual to be released was Edition 0.11 Beta in October of 1989. The manual then underwent more substantial revision for Edition 0.13 of December 1991. David Trueman, Pat Rankin, and Michal Jaegermann contributed sections of the manual for Edition 0.13. That edition was published by the FSF as a bound book early in 1992. Since then there have been several minor revisions, notably Edition 0.14 of November 1992 that was published by the FSF in January of 1993, and Edition 0.16 of August 1993.

Edition 1.0 of AWK Language Programming represents a significant re-working of The GAWK Manual, with much additional material. The FSF and I agree that I am now the primary author. I also felt that it needed a more descriptive title.

AWK Language Programming will undoubtedly continue to evolve. An electronic version comes with the `gawk` distribution from the FSF. If you find an error in this book, please report it! See section Reporting Problems and Bugs, for information on submitting problem reports electronically, or write to me in care of the FSF.

Acknowledgements

I would like to acknowledge Richard M. Stallman, for his vision of a better world, and for his courage in founding the FSF and starting the GNU project.

The initial draft of The GAWK Manual had the following acknowledgements:

> Many people need to be thanked for their assistance in producing this manual. Jay Fenlason contributed many ideas and sample programs. Richard Mlynarik and Robert Chassell gave helpful comments on drafts of this manual. The paper A Supplemental Document for `awk` by John W. Pierce of the Chemistry Department at UC San Diego, pinpointed several issues relevant both to `awk` implementation and to this manual, that would otherwise have escaped us.

The following people provided many helpful comments on Edition 0.13 of The GAWK Manual: Rick Adams, Michael Brennan, Rich Burridge, Diane Close, Christopher ("Topher") Eliot, Michael Lijewski, Pat Rankin, Miriam Robbins, and Michal Jaegermann.

The following people provided many helpful comments for Edition 1.0 of AWK Language Programming: Karl Berry, Michael Brennan, Darrel Hankerson, Michal Jaegermann, Michael Lijewski, and Miriam Robbins. Pat Rankin, Michal Jaegermann, Darrel Hankerson and Scott Deifik updated their respective sections for Edition 1.0.

Robert J. Chassell provided much valuable advice on the use of Texinfo. He also deserves special thanks for convincing me *not* to title this book How To Gawk Politely. Karl Berry helped significantly with the TeX part of Texinfo.

David Trueman deserves special credit; he has done a yeoman job of evolving `gawk` so that it performs well, and without bugs. Although he is no longer involved with `gawk`, working with him on this project was a significant pleasure.

Scott Deifik, Darrel Hankerson, Kai Uwe Rommel, Pat Rankin, and Michal Jaegermann (in no particular order) are long time members of the `gawk` "crack portability team." Without their hard work and help, `gawk` would not be nearly the fine program it is today. It has been and continues to be a pleasure working with this team of fine people.

Jeffrey Friedl provided invaluable help in tracking down a number of last minute problems with regular expressions in `gawk` 3.0.

David and I would like to thank Brian Kernighan of Bell Labs for invaluable assistance during the testing and debugging of `gawk`, and for help in clarifying numerous points about the language. We could not have done nearly as good a job on either `gawk` or its documentation without his help.

I would like to thank Marshall and Elaine Hartholz of Seattle, and Dr. Bert and Rita Schreiber of Detroit for large amounts of quiet vacation time in their homes, which allowed me to make significant progress on this book and on `gawk` itself. Phil Hughes of SSC contributed in a very important way by loaning me his laptop Linux system, not once, but twice, allowing me to do a lot of work while away from home.

Finally, I must thank my wonderful wife, Miriam, for her patience through the many versions of this project, for her proof-reading, and for sharing me with the computer. I would like to thank my parents for their love, and for the grace with which they raised and educated me. I also must acknowledge my gratitude to G-d, for the many opportunities He has sent my way, as well as for the gifts He has given me with which to take advantage of those opportunities.

Arnold Robbins
Atlanta, Georgia
January, 1996

Go to the next section.

Go to the previous, next section.

Introduction

If you are like many computer users, you would frequently like to make changes in various text files wherever certain patterns appear, or extract data from parts of certain lines while discarding the rest. To write a program to do this in a language such as C or Pascal is a time-consuming inconvenience that may take many lines of code. The job may be easier with awk.

The awk utility interprets a special-purpose programming language that makes it possible to handle simple data-reformatting jobs with just a few lines of code.

The GNU implementation of awk is called gawk; it is fully upward compatible with the System V Release 4 version of awk. gawk is also upward compatible with the POSIX specification of the awk language. This means that all properly written awk programs should work with gawk. Thus, we usually don't distinguish between gawk and other awk implementations.

Using awk you can:

- manage small, personal databases
- generate reports
- validate data
- produce indexes, and perform other document preparation tasks
- even experiment with algorithms that can be adapted later to other computer languages

Using This Book

The term awk refers to a particular program, and to the language you use to tell this program what to do. When we need to be careful, we call the program "the awk utility" and the language "the awk language." The term gawk refers to a version of awk developed as part the GNU project. The purpose of this book is to explain both the awk language and how to run the awk utility.

The main purpose of the book is to explain the features of awk, as defined in the POSIX standard. It does so in the context of one particular implementation, gawk. While doing so, it will also attempt to describe important differences between gawk and other awk implementations. Finally, any gawk features that are not in the POSIX standard for awk will be noted.

This book has the difficult task of being both tutorial and reference. If you are a novice, feel free to skip over details that seem too complex. You should also ignore the many cross references; they are for the expert user, and for the on-line Info version of the document.

The term awk program refers to a program written by you in the awk programming language.

See section Getting Started with awk, for the bare essentials you need to know to start using awk.

Some useful "one-liners" are included to give you a feel for the awk language (see section Useful One

Line Programs).

Many sample `awk` programs have been provided for you (see section A Library of `awk` Functions; also see section Practical `awk` Programs).

The entire `awk` language is summarized for quick reference in section `gawk` Summary. Look there if you just need to refresh your memory about a particular feature.

If you find terms that you aren't familiar with, try looking them up in the glossary (see section Glossary).

Most of the time complete `awk` programs are used as examples, but in some of the more advanced sections, only the part of the `awk` program that illustrates the concept being described is shown.

While this book is aimed principally at people who have not been exposed to `awk`, there is a lot of information here that even the `awk` expert should find useful. In particular, the description of POSIX `awk`, and the example programs in section A Library of `awk` Functions, and section Practical `awk` Programs, should be of interest.

Dark Corners

Until the POSIX standard (and The Gawk Manual), many features of `awk` were either poorly documented, or not documented at all. Descriptions of such features (often called "dark corners") are noted in this book with "(d.c.)". They also appear in the index under the heading "dark corner."

Typographical Conventions

This book is written using Texinfo, the GNU documentation formatting language. A single Texinfo source file is used to produce both the printed and on-line versions of the documentation. Because of this, the typographical conventions are slightly different than in other books you may have read.

Examples you would type at the command line are preceded by the common shell primary and secondary prompts, `$' and `>'. Output from the command is preceded by the glyph "-|". This typically represents the command's standard output. Error messages, and other output on the command's standard error, are preceded by the glyph "error-->". For example:

```
$ echo hi on stdout
-| hi on stdout
$ echo hello on stderr 1>&2
error--> hello on stderr
```

In the text, command names appear in `this font`, while code segments appear in the same font and quoted, `like this'. Some things will be emphasized *like this*, and if a point needs to be made strongly, it will be done **like this**. The first occurrence of a new term is usually its definition, and appears in the same font as the previous occurrence of "definition" in this sentence. File names are indicated like this: `` `/path/to/ourfile' ``.

Characters that you type at the keyboard look like this. In particular, there are special characters calle "control characters." These are characters that you type by holding down both the CONTROL key ar another key, at the same time. For example, a Control-d is typed by first pressing and holding the CONTROL key, next pressing the d key, and finally releasing both keys.

Data Files for the Examples

Many of the examples in this book take their input from two sample data files. The first, called `BBS-list', represents a list of computer bulletin board systems together with information about those systems. The second data file, called `inventory-shipped', contains information about shipments on a monthly basis. In both files, each line is considered to be one record.

In the file `BBS-list', each record contains the name of a computer bulletin board, its phone number, the board's baud rate(s), and a code for the number of hours it is operational. An `A' in the last column means the board operates 24 hours a day. A `B' in the last column means the board operates evening and weekend hours, only. A `C' means the board operates only on weekends.

```
aardvark     555-5553     1200/300          B
alpo-net     555-3412     2400/1200/300     A
barfly       555-7685     1200/300          A
bites        555-1675     2400/1200/300     A
camelot      555-0542     300               C
core         555-2912     1200/300          C
fooey        555-1234     2400/1200/300     B
foot         555-6699     1200/300          B
macfoo       555-6480     1200/300          A
sdace        555-3430     2400/1200/300     A
sabafoo      555-2127     1200/300          C
```

The second data file, called `inventory-shipped', represents information about shipments during the year. Each record contains the month of the year, the number of green crates shipped, the number of red boxes shipped, the number of orange bags shipped, and the number of blue packages shipped, respectively. There are 16 entries, covering the 12 months of one year and four months of the next year.

```
Jan  13  25  15 115
Feb  15  32  24 226
Mar  15  24  34 228
Apr  31  52  63 420
May  16  34  29 208
Jun  31  42  75 492
Jul  24  34  67 436
Aug  15  34  47 316
Sep  13  55  37 277
Oct  29  54  68 525
Nov  20  87  82 577
```


```
Dec   17   35   61 401

Jan   21   36   64 620
Feb   26   58   80 652
Mar   24   75   70 495
Apr   21   70   74 514
```

Go to the previous, next section.

Go to the previous, next section.

Getting Started with awk

The basic function of awk is to search files for lines (or other units of text) that contain certain patterns. When a line matches one of the patterns, awk performs specified actions on that line. awk keeps processing input lines in this way until the end of the input files are reached.

Programs in awk are different from programs in most other languages, because awk programs are data-driven; that is, you describe the data you wish to work with, and then what to do when you find it. Most other languages are procedural; you have to describe, in great detail, every step the program is to take. When working with procedural languages, it is usually much harder to clearly describe the data your program will process. For this reason, awk programs are often refreshingly easy to both write and read.

When you run awk, you specify an awk program that tells awk what to do. The program consists of a series of rules. (It may also contain function definitions, an advanced feature which we will ignore for now. See section User-defined Functions.) Each rule specifies one pattern to search for, and one action to perform when that pattern is found.

Syntactically, a rule consists of a pattern followed by an action. The action is enclosed in curly braces to separate it from the pattern. Rules are usually separated by newlines. Therefore, an awk program looks like this:

```
pattern { action }
pattern { action }
...
```

A Rose By Any Other Name

The awk language has evolved over the years. Full details are provided in section The Evolution of the awk Language. The language described in this book is often referred to as "new awk."

Because of this, many systems have multiple versions of awk. Some systems have an awk utility that implements the original version of the awk language, and a nawk utility for the new version. Others have an oawk for the "old awk" language, and plain awk for the new one. Still others only have one version, usually the new one.(2)

All in all, this makes it difficult for you to know which version of awk you should run when writing your programs. The best advice we can give here is to check your local documentation. Look for awk, oawk, and nawk, as well as for gawk. Chances are, you will have some version of new awk on your system, and that is what you should use when running your programs. (Of course, if you're reading this book, chances are good that you have gawk!)

Throughout this book, whenever we refer to a language feature that should be available in any complete

implementation of POSIX awk, we simply use the term awk. When referring to a feature that is spec to the GNU implementation, we use the term gawk.

How to Run awk Programs

There are several ways to run an awk program. If the program is short, it is easiest to include it in the command that runs awk, like this:

```
awk 'program' input-file1 input-file2 ...
```

where program consists of a series of patterns and actions, as described earlier. (The reason for the single quotes is described below, in section One-shot Throw-away awk Programs.)

When the program is long, it is usually more convenient to put it in a file and run it with a command like this:

```
awk -f program-file input-file1 input-file2 ...
```

One-shot Throw-away awk Programs

Once you are familiar with awk, you will often type in simple programs the moment you want to use them. Then you can write the program as the first argument of the awk command, like this:

```
awk 'program' input-file1 input-file2 ...
```

where program consists of a series of patterns and actions, as described earlier.

This command format instructs the shell, or command interpreter, to start awk and use the program to process records in the input file(s). There are single quotes around program so that the shell doesn't interpret any awk characters as special shell characters. They also cause the shell to treat all of program as a single argument for awk and allow program to be more than one line long.

This format is also useful for running short or medium-sized awk programs from shell scripts, because it avoids the need for a separate file for the awk program. A self-contained shell script is more reliable since there are no other files to misplace.

section Useful One Line Programs, presents several short, self-contained programs.

As an interesting side point, the command

```
awk '/foo/' files ...
```

is essentially the same as

```
egrep foo files ...
```

Running awk without Input Files

You can also run awk without any input files. If you type the command line:

```
awk 'program'
```

then awk applies the program to the standard input, which usually means whatever you type on the terminal. This continues until you indicate end-of-file by typing Control-d. (On other operating systems, the end-of-file character may be different. For example, on OS/2 and MS-DOS, it is Control-z.)

For example, the following program prints a friendly piece of advice (from Douglas Adams' The Hitchhiker's Guide to the Galaxy), to keep you from worrying about the complexities of computer programming (`BEGIN' is a feature we haven't discussed yet).

```
$ awk "BEGIN { print \"Don't Panic!\" }"
-| Don't Panic!
```

This program does not read any input. The `\' before each of the inner double quotes is necessary because of the shell's quoting rules, in particular because it mixes both single quotes and double quotes.

This next simple awk program emulates the cat utility; it copies whatever you type at the keyboard to its standard output. (Why this works is explained shortly.)

```
$ awk '{ print }'
Now is the time for all good men
-| Now is the time for all good men
to come to the aid of their country.
-| to come to the aid of their country.
Four score and seven years ago, ...
-| Four score and seven years ago, ...
What, me worry?
-| What, me worry?
Control-d
```

Running Long Programs

Sometimes your awk programs can be very long. In this case it is more convenient to put the program into a separate file. To tell awk to use that file for its program, you type:

```
awk -f source-file input-file1 input-file2 ...
```

The `-f' instructs the awk utility to get the awk program from the file source-file. Any file name can be used for source-file. For example, you could put the program:

```
BEGIN { print "Don't Panic!" }
```

into the file `advice'. Then this command:

```
awk -f advice
```

does the same thing as this one:

```
awk "BEGIN { print \"Don't Panic!\" }"
```

which was explained earlier (see section Running awk without Input Files). Note that you don't usually need single quotes around the file name that you specify with `-f', because most file names don't contain any of the shell's special characters. Notice that in `advice', the awk program did not have single quotes around it. The quotes are only needed for programs that are provided on the awk command line.

If you want to identify your awk program files clearly as such, you can add the extension `.awk' to the file name. This doesn't affect the execution of the awk program, but it does make "housekeeping" easier.

Executable awk Programs

Once you have learned awk, you may want to write self-contained awk scripts, using the `#!' script mechanism. You can do this on many Unix systems(3) (and someday on the GNU system).

For example, you could update the file `advice' to look like this:

```
#! /bin/awk -f

BEGIN    { print "Don't Panic!" }
```

After making this file executable (with the chmod utility), you can simply type `advice' at the shell, and the system will arrange to run awk (4) as if you had typed `awk -f advice'.

```
$ advice
-| Don't Panic!
```

Self-contained awk scripts are useful when you want to write a program which users can invoke without their having to know that the program is written in awk.

Some older systems do not support the `#!' mechanism. You can get a similar effect using a regular shell script. It would look something like this:

```
: The colon ensures execution by the standard shell.
awk 'program' "$@"
```

Using this technique, it is *vital* to enclose the program in single quotes to protect it from interpretation by the shell. If you omit the quotes, only a shell wizard can predict the results.

The "$@" causes the shell to forward all the command line arguments to the awk program, without interpretation. The first line, which starts with a colon, is used so that this shell script will work even if

invoked by a user who uses the C shell. (Not all older systems obey this convention, but many do.)

Comments in awk Programs

A comment is some text that is included in a program for the sake of human readers; it is not really part of the program. Comments can explain what the program does, and how it works. Nearly all programming languages have provisions for comments, because programs are typically hard to understand without their extra help.

In the `awk` language, a comment starts with the sharp sign character, `#', and continues to the end of the line. The `#' does not have to be the first character on the line. The `awk` language ignores the rest of a line following a sharp sign. For example, we could have put the following into `advice':

```
# This program prints a nice friendly message.  It helps
# keep novice users from being afraid of the computer.
BEGIN    { print "Don't Panic!" }
```

You can put comment lines into keyboard-composed throw-away `awk` programs also, but this usually isn't very useful; the purpose of a comment is to help you or another person understand the program at a later time.

A Very Simple Example

The following command runs a simple `awk` program that searches the input file `BBS-list' for the string of characters: `foo'. (A string of characters is usually called a string. The term string is perhaps based on similar usage in English, such as "a string of pearls," or, "a string of cars in a train.")

```
awk '/foo/ { print $0 }' BBS-list
```

When lines containing `foo' are found, they are printed, because `print $0' means print the current line. (Just `print' by itself means the same thing, so we could have written that instead.)

You will notice that slashes, `/', surround the string `foo' in the `awk` program. The slashes indicate that `foo' is a pattern to search for. This type of pattern is called a regular expression, and is covered in more detail later (see section Regular Expressions). The pattern is allowed to match parts of words. There are single-quotes around the `awk` program so that the shell won't interpret any of it as special shell characters.

Here is what this program prints:

```
$ awk '/foo/ { print $0 }' BBS-list
-| fooey         555-1234     2400/1200/300     B
-| foot          555-6699     1200/300          B
-| macfoo        555-6480     1200/300          A
-| sabafoo       555-2127     1200/300          C
```

In an `awk` rule, either the pattern or the action can be omitted, but not both. If the pattern is omitted, the action is performed for *every* input line. If the action is omitted, the default action is to print all li that match the pattern.

Thus, we could leave out the action (the `print` statement and the curly braces) in the above example, and the result would be the same: all lines matching the pattern `foo' would be printed. By comparison, omitting the `print` statement but retaining the curly braces makes an empty action that does nothing; then no lines would be printed.

An Example with Two Rules

The `awk` utility reads the input files one line at a time. For each line, `awk` tries the patterns of each of the rules. If several patterns match then several actions are run, in the order in which they appear in the `awk` program. If no patterns match, then no actions are run.

After processing all the rules (perhaps none) that match the line, `awk` reads the next line (however, see section The `next` Statement, and also see section The `nextfile` Statement). This continues until the end of the file is reached.

For example, the `awk` program:

```
/12/  { print $0 }
/21/  { print $0 }
```

contains two rules. The first rule has the string `12' as the pattern and `print $0' as the action. The second rule has the string `21' as the pattern and also has `print $0' as the action. Each rule's action is enclosed in its own pair of braces.

This `awk` program prints every line that contains the string `12' *or* the string `21'. If a line contains both strings, it is printed twice, once by each rule.

This is what happens if we run this program on our two sample data files, `` `BBS-list' `` and `` `inventory-shipped' ``, as shown here:

```
$ awk '/12/ { print $0 }
>      /21/ { print $0 }' BBS-list inventory-shipped
-| aardvark     555-5553     1200/300          B
-| alpo-net     555-3412     2400/1200/300     A
-| barfly       555-7685     1200/300          A
-| bites        555-1675     2400/1200/300     A
-| core         555-2912     1200/300          C
-| fooey        555-1234     2400/1200/300     B
-| foot         555-6699     1200/300          B
-| macfoo       555-6480     1200/300          A
-| sdace        555-3430     2400/1200/300     A
-| sabafoo      555-2127     1200/300          C
```


```
-| sabafoo      555-2127     1200/300          C
-| Jan  21  36  64 620
-| Apr  21  70  74 514
```

Note how the line in `BBS-list' beginning with `sabafoo' was printed twice, once for each rule.

A More Complex Example

Here is an example to give you an idea of what typical `awk` programs do. This example shows how `awk` can be used to summarize, select, and rearrange the output of another utility. It uses features that haven't been covered yet, so don't worry if you don't understand all the details.

```
ls -lg | awk '$6 == "Nov" { sum += $5 }
             END { print sum }'
```

This command prints the total number of bytes in all the files in the current directory that were last modified in November (of any year). (In the C shell you would need to type a semicolon and then a backslash at the end of the first line; in a POSIX-compliant shell, such as the Bourne shell or Bash, the GNU Bourne-Again shell, you can type the example as shown.)

The `ls -lg' part of this example is a system command that gives you a listing of the files in a directory, including file size and the date the file was last modified. Its output looks like this:

```
-rw-r--r--  1 arnold   user   1933 Nov  7 13:05 Makefile
-rw-r--r--  1 arnold   user  10809 Nov  7 13:03 gawk.h
-rw-r--r--  1 arnold   user    983 Apr 13 12:14 gawk.tab.h
-rw-r--r--  1 arnold   user  31869 Jun 15 12:20 gawk.y
-rw-r--r--  1 arnold   user  22414 Nov  7 13:03 gawk1.c
-rw-r--r--  1 arnold   user  37455 Nov  7 13:03 gawk2.c
-rw-r--r--  1 arnold   user  27511 Dec  9 13:07 gawk3.c
-rw-r--r--  1 arnold   user   7989 Nov  7 13:03 gawk4.c
```

The first field contains read-write permissions, the second field contains the number of links to the file, and the third field identifies the owner of the file. The fourth field identifies the group of the file. The fifth field contains the size of the file in bytes. The sixth, seventh and eighth fields contain the month, day, and time, respectively, that the file was last modified. Finally, the ninth field contains the name of the file.

The `$6 == "Nov"' in our `awk` program is an expression that tests whether the sixth field of the output from `ls -lg' matches the string `Nov'. Each time a line has the string `Nov' for its sixth field, the action `sum += $5' is performed. This adds the fifth field (the file size) to the variable `sum`. As a result, when `awk` has finished reading all the input lines, `sum` is the sum of the sizes of files whose lines matched the pattern. (This works because `awk` variables are automatically initialized to zero.)

After the last line of output from `ls` has been processed, the `END` rule is executed, and the value of `sum` is printed. In this example, the value of `sum` would be 80600.

These more advanced `awk` techniques are covered in later sections (see section Overview of Actions). Before you can move on to more advanced `awk` programming, you have to know how `awk` interprets your input and displays your output. By manipulating fields and using `print` statements, you can produce some very useful and impressive looking reports.

awk Statements Versus Lines

Most often, each line in an `awk` program is a separate statement or separate rule, like this:

```
awk '/12/  { print $0 }
     /21/  { print $0 }' BBS-list inventory-shipped
```

However, `gawk` will ignore newlines after any of the following:

```
,    {    ?    :    ||    &&    do    else
```

A newline at any other point is considered the end of the statement. (Splitting lines after `?' and `:' is a minor `gawk` extension. The `?' and `:' referred to here is the three operand conditional expression described in section Conditional Expressions.)

If you would like to split a single statement into two lines at a point where a newline would terminate it, you can continue it by ending the first line with a backslash character, `\'. The backslash must be the final character on the line to be recognized as a continuation character. This is allowed absolutely anywhere in the statement, even in the middle of a string or regular expression. For example:

```
awk '/This regular expression is too long, so continue it\
 on the next line/ { print $1 }'
```

We have generally not used backslash continuation in the sample programs in this book. Since in `gawk` there is no limit on the length of a line, it is never strictly necessary; it just makes programs more readable. For this same reason, as well as for clarity, we have kept most statements short in the sample programs presented throughout the book. Backslash continuation is most useful when your `awk` program is in a separate source file, instead of typed in on the command line. You should also note that many `awk` implementations are more particular about where you may use backslash continuation. For example, they may not allow you to split a string constant using backslash continuation. Thus, for maximal portability of your `awk` programs, it is best not to split your lines in the middle of a regular expression or a string.

Caution: backslash continuation does not work as described above with the C shell. Continuation with backslash works for `awk` programs in files, and also for one-shot programs *provided* you are using a POSIX-compliant shell, such as the Bourne shell or Bash, the GNU Bourne-Again shell. But the C shell (`csh`) behaves differently! There, you must use two backslashes in a row, followed by a newline. Note also that when using the C shell, *every* newline in your awk program must be escaped with a backslash. To illustrate:

```
% awk 'BEGIN { \
```


```
?   print \\
?       "hello, world" \
? }'
-| hello, world
```

Here, the `%' and `?' are the C shell's primary and secondary prompts, analogous to the standard shell's `$' and `>'.

awk is a line-oriented language. Each rule's action has to begin on the same line as the pattern. To have the pattern and action on separate lines, you *must* use backslash continuation--there is no other way.

When awk statements within one rule are short, you might want to put more than one of them on a line. You do this by separating the statements with a semicolon, `;'.

This also applies to the rules themselves. Thus, the previous program could have been written:

```
/12/ { print $0 } ; /21/ { print $0 }
```

Note: the requirement that rules on the same line must be separated with a semicolon was not in the original awk language; it was added for consistency with the treatment of statements within an action.

Other Features of awk

The awk language provides a number of predefined, or built-in variables, which your programs can use to get information from awk. There are other variables your program can set to control how awk processes your data.

In addition, awk provides a number of built-in functions for doing common computational and string related operations.

As we develop our presentation of the awk language, we introduce most of the variables and many of the functions. They are defined systematically in section Built-in Variables, and section Built-in Functions.

When to Use awk

You might wonder how awk might be useful for you. Using utility programs, advanced patterns, field separators, arithmetic statements, and other selection criteria, you can produce much more complex output. The awk language is very useful for producing reports from large amounts of raw data, such as summarizing information from the output of other utility programs like ls. (See section A More Complex Example.)

Programs written with awk are usually much smaller than they would be in other languages. This makes awk programs easy to compose and use. Often, awk programs can be quickly composed at your terminal, used once, and thrown away. Since awk programs are interpreted, you can avoid the (usually lengthy) compilation part of the typical edit-compile-test-debug cycle of software development.

Complex programs have been written in awk, including a complete retargetable assembler for eight-bit

microprocessors (see section Glossary, for more information) and a microcode assembler for a special-purpose Prolog computer. However, awk's capabilities are strained by tasks of such complexity.

If you find yourself writing awk scripts of more than, say, a few hundred lines, you might consider using a different programming language. Emacs Lisp is a good choice if you need sophisticated string or pattern matching capabilities. The shell is also good at string and pattern matching; in addition, it allows powerful use of the system utilities. More conventional languages, such as C, C++, and Lisp, offer better facilities for system programming and for managing the complexity of large programs. Programs in these languages may require more lines of source code than the equivalent awk programs, but they are easier to maintain and usually run more efficiently.

Go to the previous, next section.

Go to the previous, next section.

Useful One Line Programs

Many useful `awk` programs are short, just a line or two. Here is a collection of useful, short programs to get you started. Some of these programs contain constructs that haven't been covered yet. The description of the program will give you a good idea of what is going on, but please read the rest of the book to become an `awk` expert!

Most of the examples use a data file named `` `data' ``. This is just a placeholder; if you were to use these programs yourself, you would substitute your own file names for `` `data' ``.

```
awk '{ if (length($0) > max) max = length($0) }
END { print max }' data
```

This program prints the length of the longest input line.

```
awk 'length($0) > 80' data
```

This program prints every line that is longer than 80 characters. The sole rule has a relational expression as its pattern, and has no action (so the default action, printing the record, is used).

```
expand data | awk '{ if (x < length()) x = length() }
END { print "maximum line length is " x }'
```

This program prints the length of the longest line in `` `data' ``. The input is processed by the `expand` program to change tabs into spaces, so the widths compared are actually the right-margin columns.

```
awk 'NF > 0' data
```

This program prints every line that has at least one field. This is an easy way to delete blank lines from a file (or rather, to create a new file similar to the old file but from which the blank lines have been deleted).

```
awk 'BEGIN { for (i = 1; i <= 7; i++)
print int(101 * rand()) }'
```

This program prints seven random numbers from zero to 100, inclusive.

```
ls -lg files | awk '{ x += $5 } ; END { print "total bytes: " x }'
```

This program prints the total number of bytes used by files.

```
ls -lg files | awk '{ x += $5 }
END { print "total K-bytes: " (x + 1023)/1024 }'
```

This program prints the total number of kilobytes used by files.

```
awk -F: '{ print $1 }' /etc/passwd | sort
```

This program prints a sorted list of the login names of all users.

```
awk 'END { print NR }' data
```

This program counts lines in a file.

```
awk 'NR % 2' data
```

This program prints the even numbered lines in the data file. If you were to use the expression 'NR % 2 == 1' instead, it would print the odd number lines.

Go to the previous, next section.

Go to the previous, next section.

Regular Expressions

A regular expression, or regexp, is a way of describing a set of strings. Because regular expressions are such a fundamental part of `awk` programming, their format and use deserve a separate chapter.

A regular expression enclosed in slashes (`/') is an `awk` pattern that matches every input record whose text belongs to that set.

The simplest regular expression is a sequence of letters, numbers, or both. Such a regexp matches any string that contains that sequence. Thus, the regexp `foo' matches any string containing `foo'. Therefore, the pattern `/foo/` matches any input record containing the three characters `foo', *anywhere* in the record. Other kinds of regexps let you specify more complicated classes of strings.

Initially, the examples will be simple. As we explain more about how regular expressions work, we will present more complicated examples.

How to Use Regular Expressions

A regular expression can be used as a pattern by enclosing it in slashes. Then the regular expression is tested against the entire text of each record. (Normally, it only needs to match some part of the text in order to succeed.) For example, this prints the second field of each record that contains the three characters `foo' anywhere in it:

```
$ awk '/foo/ { print $2 }' BBS-list
-| 555-1234
-| 555-6699
-| 555-6480
-| 555-2127
```

Regular expressions can also be used in matching expressions. These expressions allow you to specify the string to match against; it need not be the entire current input record. The two operators, `~' and `!~', perform regular expression comparisons. Expressions using these operators can be used as patterns or in `if`, `while`, `for`, and `do` statements.

`exp ~ /regexp/`

> This is true if the expression exp (taken as a string) is matched by regexp. The following example matches, or selects, all input records with the upper-case letter `J' somewhere in the first field:
>
> ```
> $ awk '$1 ~ /J/' inventory-shipped
> -| Jan  13  25  15 115
> -| Jun  31  42  75 492
> -| Jul  24  34  67 436
> -| Jan  21  36  64 620
> ```

So does this:

```
awk '{ if ($1 ~ /J/) print }' inventory-shipped
```

`exp !~ /regexp/`

This is true if the expression exp (taken as a character string) is *not* matched by regexp. The following example matches, or selects, all input records whose first field *does not* contain the upper-case letter `J':

```
$ awk '$1 !~ /J/' inventory-shipped
-| Feb  15  32  24 226
-| Mar  15  24  34 228
-| Apr  31  52  63 420
-| May  16  34  29 208
...
```

When a regexp is written enclosed in slashes, like `/foo/`, we call it a regexp constant, much like `5.27` is a numeric constant, and `"foo"` is a string constant.

Escape Sequences

Some characters cannot be included literally in string constants (`"foo"`) or regexp constants (`/foo/`). You represent them instead with escape sequences, which are character sequences beginning with a backslash (`\').

One use of an escape sequence is to include a double-quote character in a string constant. Since a plain double-quote would end the string, you must use `\"' to represent an actual double-quote character as a part of the string. For example:

```
$ awk 'BEGIN { print "He said \"hi!\" to her." }'
-| He said "hi!" to her.
```

The backslash character itself is another character that cannot be included normally; you write `\\' to put one backslash in the string or regexp. Thus, the string whose contents are the two characters `"' and `\' must be written `"\"\\"'.

Another use of backslash is to represent unprintable characters such as tab or newline. While there is nothing to stop you from entering most unprintable characters directly in a string constant or regexp constant, they may look ugly.

Here is a table of all the escape sequences used in `awk`, and what they represent. Unless noted otherwise, all of these escape sequences apply to both string constants and regexp constants.

`\\`

A literal backslash, `\'.

`\a`

The "alert" character, Control-g, ASCII code 7 (BEL).

`\b`

Backspace, Control-h, ASCII code 8 (BS).

`\f`

Formfeed, Control-l, ASCII code 12 (FF).

`\n`

Newline, Control-j, ASCII code 10 (LF).

`\r`

Carriage return, Control-m, ASCII code 13 (CR).

`\t`

Horizontal tab, Control-i, ASCII code 9 (HT).

`\v`

Vertical tab, Control-k, ASCII code 11 (VT).

`\nnn`

The octal value nnn, where nnn are one to three digits between `0' and `7'. For example, the code for the ASCII ESC (escape) character is `\033'.

`\xhh...`

The hexadecimal value hh, where hh are hexadecimal digits (`0' through `9' and either `A' through `F' or `a' through `f'). Like the same construct in ANSI C, the escape sequence continues until the first non-hexadecimal digit is seen. However, using more than two hexadecimal digits produces undefined results. (The `\x' escape sequence is not allowed in POSIX `awk`.)

`\/`

A literal slash (necessary for regexp constants only). You use this when you wish to write a regexp constant that contains a slash. Since the regexp is delimited by slashes, you need to escape the slash that is part of the pattern, in order to tell `awk` to keep processing the rest of the regexp.

`\"`

A literal double-quote (necessary for string constants only). You use this when you wish to write a string constant that contains a double-quote. Since the string is delimited by double-quotes, you need to escape the quote that is part of the string, in order to tell `awk` to keep processing the rest of the string.

In `gawk`, there are additional two character sequences that begin with backslash that have special meaning in regexps. See section Additional Regexp Operators Only in `gawk`.

In a string constant, what happens if you place a backslash before something that is not one of the characters listed above? POSIX `awk` purposely leaves this case undefined. There are two choices.

- Strip the backslash out. This is what Unix `awk` and `gawk` both do. For example, `"a\qc"` is the same as `"aqc"`.
- Leave the backslash alone. Some other `awk` implementations do this. In such implementations, `"a\qc"` is the same as if you had typed `"a\\qc"`.

In a regexp, a backslash before any character that is not in the above table, and not listed in section Additional Regexp Operators Only in gawk, means that the next character should be taken literally, if it would normally be a regexp operator. E.g., `/a\+b/` matches the three characters `a+b'.

For complete portability, do not use a backslash before any character not listed in the table above.

Another interesting question arises. Suppose you use an octal or hexadecimal escape to represent a regexp metacharacter (see section Regular Expression Operators). Does `awk` treat the character as literal character, or as a regexp operator?

It turns out that historically, such characters were taken literally (d.c.). However, the POSIX standard indicates that they should be treated as real metacharacters, and this is what `gawk` does. However, in compatibility mode (see section Command Line Options), `gawk` treats the characters represented by octal and hexadecimal escape sequences literally when used in regexp constants. Thus, `/a\52b/` is equivalent to `/a\*b/`.

To summarize:

1. The escape sequences in the table above are always processed first, for both string constants and regexp constants. This happens very early, as soon as `awk` reads your program.
2. `gawk` processes both regexp constants and dynamic regexps (see section Using Dynamic Regexps), for the special operators listed in section Additional Regexp Operators Only in `gawk`.
3. A backslash before any other character means to treat that character literally.

Regular Expression Operators

You can combine regular expressions with the following characters, called regular expression operators, or metacharacters, to increase the power and versatility of regular expressions.

The escape sequences described above in section Escape Sequences, are valid inside a regexp. They are introduced by a `\'. They are recognized and converted into the corresponding real characters as the very first step in processing regexps.

Here is a table of metacharacters. All characters that are not escape sequences and that are not listed in the table stand for themselves.

`\`

This is used to suppress the special meaning of a character when matching. For example:

```
\$
```

matches the character `$'.

`^`

This matches the beginning of a string. For example:

```
^@chapter
```

matches the `@chapter' at the beginning of a string, and can be used to identify chapter beginn in Texinfo source files. The `^' is known as an anchor, since it anchors the pattern to matching at the beginning of the string.

It is important to realize that `^' does not match the beginning of a line embedded in a string. In this example the condition is not true:

```
if ("line1\nLINE 2" ~ /^L/) ...
```

`$`

This is similar to `^', but it matches only at the end of a string. For example:

```
p$
```

matches a record that ends with a `p'. The `$' is also an anchor, and also does not match the end of a line embedded in a string. In this example the condition is not true:

```
if ("line1\nLINE 2" ~ /1$/) ...
```

`.`

The period, or dot, matches any single character, *including* the newline character. For example:

```
.P
```

matches any single character followed by a `P' in a string. Using concatenation we can make a regular expression like `U.A', which matches any three-character sequence that begins with `U' and ends with `A'.

In strict POSIX mode (see section Command Line Options), `.' does not match the NUL character, which is a character with all bits equal to zero. Otherwise, NUL is just another character. Other versions of `awk` may not be able to match the NUL character.

`[...]`

This is called a character list. It matches any *one* of the characters that are enclosed in the square brackets. For example:

```
[MVX]
```

matches any one of the characters `M', `V', or `X' in a string.

Ranges of characters are indicated by using a hyphen between the beginning and ending characters, and enclosing the whole thing in brackets. For example:

```
[0-9]
```

matches any digit. Multiple ranges are allowed. E.g., the list `[A-Za-z0-9]` is a common way to express the idea of "all alphanumeric characters."

To include one of the characters `\', `]', `-' or `^' in a character list, put a `\' in front of it. For example:

```
[d\]]
```

matches either `d', or `]'.

This treatment of `\' in character lists is compatible with other `awk` implementations, and is also mandated by POSIX. The regular expressions in `awk` are a superset of the POSIX specification for Extended Regular Expressions (EREs). POSIX EREs are based on the regular expressions accepted by the traditional `egrep` utility.

Character classes are a new feature introduced in the POSIX standard. A character class is a special notation for describing lists of characters that have a specific attribute, but where the actual characters themselves can vary from country to country and/or from character set to character set. For example, the notion of what is an alphabetic character differs in the USA and in France.

A character class is only valid in a regexp *inside* the brackets of a character list. Character classes consist of `[:', a keyword denoting the class, and `:]'. Here are the character classes defined by the POSIX standard.

`[:alnum:]`
: Alphanumeric characters.

`[:alpha:]`
: Alphabetic characters.

`[:blank:]`
: Space and tab characters.

`[:cntrl:]`
: Control characters.

`[:digit:]`
: Numeric characters.

`[:graph:]`
: Characters that are printable and are also visible. (A space is printable, but not visible, while an `a' is both.)

`[:lower:]`
: Lower-case alphabetic characters.

`[:print:]`
: Printable characters (characters that are not control characters.)

`[:punct:]`
: Punctuation characters (characters that are not letter, digits, control characters, or space characters).

`[:space:]`
: Space characters (such as space, tab, and formfeed, to name a few).

`[:upper:]`

Upper-case alphabetic characters.

`[:xdigit:]`

Characters that are hexadecimal digits.

For example, before the POSIX standard, to match alphanumeric characters, you had to write `/[A-Za-z0-9]/`. If your character set had other alphabetic characters in it, this would not match them. With the POSIX character classes, you can write `/[[:alnum:]]/`, and this will match *all* the alphabetic and numeric characters in your character set.

Two additional special sequences can appear in character lists. These apply to non-ASCII character sets, which can have single symbols (called collating elements) that are represented with more than one character, as well as several characters that are equivalent for collating, or sorting, purposes. (E.g., in French, a plain "e" and a grave-accented "`e" are equivalent.)

Collating Symbols

A collating symbol is a multi-character collating element enclosed in `[.' and `.]'. For example, if `ch' is a collating element, then `[[.ch.]]` is a regexp that matches this collating element, while `[ch]` is a regexp that matches either `c' or `h'.

Equivalence Classes

An equivalence class is a list of equivalent characters enclosed in `[=' and `=]'. Thus, `[[=e`e=]]` is regexp that matches either `e' or ``e'.

These features are very valuable in non-English speaking locales.

Caution: The library functions that `gawk` uses for regular expression matching currently only recognize POSIX character classes; they do not recognize collating symbols or equivalence classes.

- [^ ...] This is a complemented character list. The first character after the `[' *must* be a `^'. It matches any characters *except* those in the square brackets, or newline. For example:

```
[^0-9]
```

matches any character that is not a digit.

- | This is the alternation operator, and it is used to specify alternatives. For example:

```
^P|[0-9]
```

matches any string that matches either `^P' or `[0-9]'. This means it matches any string that starts with `P' or contains a digit.

The alternation applies to the largest possible regexps on either side. In other words, `|' has the lowest precedence of all the regular expression operators.

- (...) Parentheses are used for grouping in regular expressions as in arithmetic. They can be used to concatenate regular expressions containing the alternation operator, `|'. For example, `@(samp|code)\{[^}]+\}' matches both `@code{foo}' and `@samp{bar}'. (These are Texinfo formatting

control sequences.)

- * This symbol means that the preceding regular expression is to be repeated as many times as necessary to find a match. For example:

```
ph*
```

applies the `*' symbol to the preceding `h' and looks for matches of one `p' followed by any number of `h's. This will also match just `p' if no `h's are present.

The `*' repeats the *smallest* possible preceding expression. (Use parentheses if you wish to repeat a larger expression.) It finds as many repetitions as possible. For example:

```
awk '/\(c[ad][ad]*r x\)/ { print }' sample
```

prints every record in `sample' containing a string of the form `(car x)', `(cdr x)', `(cadr x)', and so on. Notice the escaping of the parentheses by preceding them with backslashes.

- + This symbol is similar to `*', but the preceding expression must be matched at least once. This means that:

```
wh+y
```

would match `why' and `whhy' but not `wy', whereas `wh*y' would match all three of these strings. This is a simpler way of writing the last `*' example:

```
awk '/\(c[ad]+r x\)/ { print }' sample
```

- ? This symbol is similar to `*', but the preceding expression can be matched either once or not at all. For example:

```
fe?d
```

will match `fed' and `fd', but nothing else.

- {n}
- {n,}
- {n,m} One or two numbers inside braces denote an interval expression. If there is one number in the braces, the preceding regexp is repeated n times. If there are two numbers separated by a comma, the preceding regexp is repeated n to m times. If there is one number followed by a comma, then the preceding regexp is repeated at least n times.

`wh{3}y`
: matches `whhhy' but not `why' or `whhhhy'.

`wh{3,5}y`
: matches `whhhy' or `whhhhy' or `whhhhhy', only.

`wh{2,}y`
: matches `whhy' or `whhhy', and so on.

Interval expressions were not traditionally available in `awk`. As part of the POSIX standard they were added, to make `awk` and `egrep` consistent with each other.

However, since old programs may use `{' and `}' in regexp constants, by default `gawk` does *not* match interval expressions in regexps. If either `--posix' or `--re-interval' are specified (see section Command Line Options), then interval expressions are allowed in regexps.

In regular expressions, the `*', `+', and `?' operators, as well as the braces `{' and `}', have the highest precedence, followed by concatenation, and finally by `|'. As in arithmetic, parentheses can change how operators are grouped.

If `gawk` is in compatibility mode (see section Command Line Options), character classes and interval expressions are not available in regular expressions.

The next section discusses the GNU-specific regexp operators, and provides more detail concerning how command line options affect the way `gawk` interprets the characters in regular expressions.

Additional Regexp Operators Only in gawk

GNU software that deals with regular expressions provides a number of additional regexp operators. These operators are described in this section, and are specific to `gawk`; they are not available in other `awk` implementations.

Most of the additional operators are for dealing with word matching. For our purposes, a word is a sequence of one or more letters, digits, or underscores (`_').

`\w`

This operator matches any word-constituent character, i.e. any letter, digit, or underscore. Think of it as a short-hand for `[[:alnum:]_]`.

`\W`

This operator matches any character that is not word-constituent. Think of it as a short-hand for `[^[:alnum:]_]`.

`\<`

This operator matches the empty string at the beginning of a word. For example, `/\<away/` matches `away', but not `stowaway'.

`\>`

This operator matches the empty string at the end of a word. For example, `/stow\>/` matches `stow', but not `stowaway'.

`\y`

This operator matches the empty string at either the beginning or the end of a word (the word boundary). For example, `\yballs?\y' matches either `ball' or `balls' as a separate word.

`\B`

This operator matches the empty string within a word. In other words, `\B' matches the empty string that occurs between two word-constituent characters. For example, `/\Brat\B/` matches

`crate', but it does not match `dirty rat'. `\B' is essentially the opposite of `\y'.

There are two other operators that work on buffers. In Emacs, a buffer is, naturally, an Emacs buffer. For other programs, the regexp library routines that `gawk` uses consider the entire string to be matched as the buffer.

For `awk`, since `^' and `$' always work in terms of the beginning and end of strings, these operators don't add any new capabilities. They are provided for compatibility with other GNU software.

`\``

This operator matches the empty string at the beginning of the buffer.

`\'`

This operator matches the empty string at the end of the buffer.

In other GNU software, the word boundary operator is `\b'. However, that conflicts with the `awk` language's definition of `\b' as backspace, so `gawk` uses a different letter.

An alternative method would have been to require two backslashes in the GNU operators, but this was deemed to be too confusing, and the current method of using `\y' for the GNU `\b' appears to be the lesser of two evils.

The various command line options (see section Command Line Options) control how `gawk` interprets characters in regexps.

No options

In the default case, `gawk` provide all the facilities of POSIX regexps and the GNU regexp operators described above. However, interval expressions are not supported.

`--posix`

Only POSIX regexps are supported, the GNU operators are not special (e.g., `\w' matches a literal `w'). Interval expressions are allowed.

`--traditional`

Traditional Unix `awk` regexps are matched. The GNU operators are not special, interval expressions are not available, and neither are the POSIX character classes (`[[:alnum:]]` and so on). Characters described by octal and hexadecimal escape sequences are treated literally, even if they represent regexp metacharacters.

`--re-interval`

Allow interval expressions in regexps, even if `--traditional' has been provided.

Case-sensitivity in Matching

Case is normally significant in regular expressions, both when matching ordinary characters (i.e. not metacharacters), and inside character sets. Thus a `w' in a regular expression matches only a lower-case `w' and not an upper-case `W'.

The simplest way to do a case-independent match is to use a character list: `[Ww]'. However, this can be cumbersome if you need to use it often; and it can make the regular expressions harder to read. There are

two alternatives that you might prefer.

One way to do a case-insensitive match at a particular point in the program is to convert the data to a single case, using the `tolower` or `toupper` built-in string functions (which we haven't discussed yet; see section Built-in Functions for String Manipulation). For example:

```
tolower($1) ~ /foo/  { ... }
```

converts the first field to lower-case before matching against it. This will work in any POSIX-compliant implementation of `awk`.

Another method, specific to `gawk`, is to set the variable `IGNORECASE` to a non-zero value (see section Built-in Variables). When `IGNORECASE` is not zero, *all* regexp and string operations ignore case. Changing the value of `IGNORECASE` dynamically controls the case sensitivity of your program as it runs. Case is significant by default because `IGNORECASE` (like most variables) is initialized to zero.

```
x = "aB"
if (x ~ /ab/) ...   # this test will fail

IGNORECASE = 1
if (x ~ /ab/) ...   # now it will succeed
```

In general, you cannot use `IGNORECASE` to make certain rules case-insensitive and other rules case-sensitive, because there is no way to set `IGNORECASE` just for the pattern of a particular rule. To do this, you must use character lists or `tolower`. However, one thing you can do only with `IGNORECASE` is turn case-sensitivity on or off dynamically for all the rules at once.

`IGNORECASE` can be set on the command line, or in a `BEGIN` rule (see section Other Command Line Arguments; also see section Startup and Cleanup Actions). Setting `IGNORECASE` from the command line is a way to make a program case-insensitive without having to edit it.

Prior to version 3.0 of `gawk`, the value of `IGNORECASE` only affected regexp operations. It did not affect string comparison with `==', `!=', and so on. Beginning with version 3.0, both regexp and string comparison operations are affected by `IGNORECASE`.

Beginning with version 3.0 of `gawk`, the equivalences between upper-case and lower-case characters are based on the ISO-8859-1 (ISO Latin-1) character set. This character set is a superset of the traditional 128 ASCII characters, that also provides a number of characters suitable for use with European languages.

The value of `IGNORECASE` has no effect if `gawk` is in compatibility mode (see section Command Line Options). Case is always significant in compatibility mode.

How Much Text Matches?

Consider the following example:

```
echo aaaabcd | awk '{ sub(/a+/, "<A>"); print }'
```

This example uses the `sub` function (which we haven't discussed yet, see section Built-in Functions for String Manipulation) to make a change to the input record. Here, the regexp `/a+/` indicates "one or more `a' characters," and the replacement text is `<A>'.

The input contains four `a' characters. What will the output be? In other words, how many is "one or more"---will `awk` match two, three, or all four `a' characters?

The answer is, `awk` (and POSIX) regular expressions always match the leftmost, *longest* sequence of input characters that can match. Thus, in this example, all four `a' characters are replaced with `<A>'.

```
$ echo aaaabcd | awk '{ sub(/a+/, "<A>"); print }'
-| <A>bcd
```

For simple match/no-match tests, this is not so important. But when doing regexp-based field and record splitting, and text matching and substitutions with the `match`, `sub`, `gsub`, and `gensub` functions, it is very important. Understanding this principle is also important for regexp-based record and field splitting (see section How Input is Split into Records, and also see section Specifying How Fields are Separated).

Using Dynamic Regexps

The right hand side of a `~' or `!~' operator need not be a regexp constant (i.e. a string of characters between slashes). It may be any expression. The expression is evaluated, and converted if necessary to a string; the contents of the string are used as the regexp. A regexp that is computed in this way is called a dynamic regexp. For example:

```
BEGIN { identifier_regexp = "[A-Za-z_][A-Za-z_0-9]+" }
$0 ~ identifier_regexp    { print }
```

sets `identifier_regexp` to a regexp that describes `awk` variable names, and tests if the input record matches this regexp.

Caution: When using the `~' and `!~' operators, there is a difference between a regexp constant enclosed in slashes, and a string constant enclosed in double quotes. If you are going to use a string constant, you have to understand that the string is in essence scanned *twice*; the first time when `awk` reads your program, and the second time when it goes to match the string on the left-hand side of the operator with the pattern on the right. This is true of any string valued expression (such as `identifier_regexp` above), not just string constants.

What difference does it make if the string is scanned twice? The answer has to do with escape sequences,

and particularly with backslashes. To get a backslash into a regular expression inside a string, you ha to type two backslashes.

For example, `/\*/` is a regexp constant for a literal `*'. Only one backslash is needed. To do the same thing with a string, you would have to type `"*"`. The first backslash escapes the second one, so that the string actually contains the two characters `\' and `*'.

Given that you can use both regexp and string constants to describe regular expressions, which should you use? The answer is "regexp constants," for several reasons.

1. String constants are more complicated to write, and more difficult to read. Using regexp constants makes your programs less error-prone. Not understanding the difference between the two kinds of constants is a common source of errors.
2. It is also more efficient to use regexp constants: `awk` can note that you have supplied a regexp and store it internally in a form that makes pattern matching more efficient. When using a string constant, `awk` must first convert the string into this internal form, and then perform the pattern matching.
3. Using regexp constants is better style; it shows clearly that you intend a regexp match.

Go to the previous, next section.

Go to the previous, next section.

Reading Input Files

In the typical `awk` program, all input is read either from the standard input (by default the keyboard, but often a pipe from another command) or from files whose names you specify on the `awk` command line. If you specify input files, `awk` reads them in order, reading all the data from one before going on to the next. The name of the current input file can be found in the built-in variable `FILENAME` (see section Built-in Variables).

The input is read in units called records, and processed by the rules of your program one record at a time. By default, each record is one line. Each record is automatically split into chunks called fields. This makes it more convenient for programs to work on the parts of a record.

On rare occasions you will need to use the `getline` command. The `getline` command is valuable, both because it can do explicit input from any number of files, and because the files used with it do not have to be named on the `awk` command line (see section Explicit Input with `getline`).

How Input is Split into Records

The `awk` utility divides the input for your `awk` program into records and fields. Records are separated by a character called the record separator. By default, the record separator is the newline character. This is why records are, by default, single lines. You can use a different character for the record separator by assigning the character to the built-in variable `RS`.

You can change the value of `RS` in the `awk` program, like any other variable, with the assignment operator, `=' (see section Assignment Expressions). The new record-separator character should be enclosed in quotation marks, which indicate a string constant. Often the right time to do this is at the beginning of execution, before any input has been processed, so that the very first record will be read with the proper separator. To do this, use the special `BEGIN` pattern (see section The `BEGIN` and `END` Special Patterns). For example:

```
awk 'BEGIN { RS = "/" } ; { print $0 }' BBS-list
```

changes the value of `RS` to `"/"`, before reading any input. This is a string whose first character is a slash; as a result, records are separated by slashes. Then the input file is read, and the second rule in the `awk` program (the action with no pattern) prints each record. Since each `print` statement adds a newline at the end of its output, the effect of this `awk` program is to copy the input with each slash changed to a newline. Here are the results of running the program on `` `BBS-list' ``:

```
$ awk 'BEGIN { RS = "/" } ; { print $0 }' BBS-list
-| aardvark     555-5553     1200
-| 300          B
-| alpo-net     555-3412     2400
```


```
-| 1200
-| 300     A
-| barfly        555-7685     1200
-| 300           A
-| bites         555-1675     2400
-| 1200
-| 300     A
-| camelot       555-0542     300                C
-| core          555-2912     1200
-| 300           C
-| fooey         555-1234     2400
-| 1200
-| 300     B
-| foot          555-6699     1200
-| 300           B
-| macfoo        555-6480     1200
-| 300           A
-| sdace         555-3430     2400
-| 1200
-| 300     A
-| sabafoo       555-2127     1200
-| 300           C
-|
```

Note that the entry for the `camelot' BBS is not split. In the original data file (see section Data Files for the Examples), the line looks like this:

```
camelot       555-0542     300                C
```

It only has one baud rate; there are no slashes in the record.

Another way to change the record separator is on the command line, using the variable-assignment feature (see section Other Command Line Arguments).

```
awk '{ print $0 }' RS="/" BBS-list
```

This sets RS to `/' before processing `BBS-list'.

Using an unusual character such as `/' for the record separator produces correct behavior in the vast majority of cases. However, the following (extreme) pipeline prints a surprising `1'. There is one field, consisting of a newline. The value of the built-in variable NF is the number of fields in the current record.

```
$ echo | awk 'BEGIN { RS = "a" } ; { print NF }'
-| 1
```

Reaching the end of an input file terminates the current input record, even if the last character in the file is not the character in RS (d.c.).

The empty string, "" (a string of no characters), has a special meaning as the value of RS: it means that records are separated by one or more blank lines, and nothing else. See section Multiple-Line Records for more details.

If you change the value of RS in the middle of an awk run, the new value is used to delimit subsequent records, but the record currently being processed (and records already processed) are not affected.

After the end of the record has been determined, gawk sets the variable RT to the text in the input that matched RS.

The value of RS is in fact not limited to a one-character string. It can be any regular expression (see section Regular Expressions). In general, each record ends at the next string that matches the regular expression; the next record starts at the end of the matching string. This general rule is actually at work in the usual case, where RS contains just a newline: a record ends at the beginning of the next matching string (the next newline in the input) and the following record starts just after the end of this string (at the first character of the following line). The newline, since it matches RS, is not part of either record.

When RS is a single character, RT will contain the same single character. However, when RS is a regular expression, then RT becomes more useful; it contains the actual input text that matched the regular expression.

The following example illustrates both of these features. It sets RS equal to a regular expression that matches either a newline, or a series of one or more upper-case letters with optional leading and/or trailing white space (see section Regular Expressions).

```
$ echo record 1 AAAA record 2 BBBB record 3 |
> gawk 'BEGIN { RS = "\n|( *[[:upper:]]+ *)" }
>             { print "Record =", $0, "and RT =", RT }'
-| Record = record 1 and RT =  AAAA
-| Record = record 2 and RT =  BBBB
-| Record = record 3 and RT =
-|
```

The final line of output has an extra blank line. This is because the value of RT is a newline, and then the print statement supplies its own terminating newline.

See section A Simple Stream Editor, for a more useful example of RS as a regexp and RT.

The use of RS as a regular expression and the RT variable are gawk extensions; they are not available in compatibility mode (see section Command Line Options). In compatibility mode, only the first character of the value of RS is used to determine the end of the record.

The awk utility keeps track of the number of records that have been read so far from the current input file. This value is stored in a built-in variable called FNR. It is reset to zero when a new file is started. Another built-in variable, NR, is the total number of input records read so far from all data files. It starts at zero but is never automatically reset to zero.

Examining Fields

When `awk` reads an input record, the record is automatically separated or parsed by the interpreter into chunks called fields. By default, fields are separated by whitespace, like words in a line. Whitespace in `awk` means any string of one or more spaces and/or tabs; other characters such as newline, formfeed, and so on, that are considered whitespace by other languages are *not* considered whitespace by `awk`.

The purpose of fields is to make it more convenient for you to refer to these pieces of the record. You don't have to use them--you can operate on the whole record if you wish--but fields are what make simple `awk` programs so powerful.

To refer to a field in an `awk` program, you use a dollar-sign, `$', followed by the number of the field you want. Thus, `$1` refers to the first field, `$2` to the second, and so on. For example, suppose the following is a line of input:

```
This seems like a pretty nice example.
```

Here the first field, or `$1`, is `This'; the second field, or `$2`, is `seems'; and so on. Note that the last field, `$7`, is `example.'. Because there is no space between the `e' and the `.', the period is considered part of the seventh field.

`NF` is a built-in variable whose value is the number of fields in the current record. `awk` updates the value of `NF` automatically, each time a record is read.

No matter how many fields there are, the last field in a record can be represented by `$NF`. So, in the example above, `$NF` would be the same as `$7`, which is `example.'. Why this works is explained below (see section Non-constant Field Numbers). If you try to reference a field beyond the last one, such as `$8` when the record has only seven fields, you get the empty string.

`$0`, which looks like a reference to the "zeroth" field, is a special case: it represents the whole input record. `$0` is used when you are not interested in fields.

Here are some more examples:

```
$ awk '$1 ~ /foo/ { print $0 }' BBS-list
-| fooey         555-1234     2400/1200/300     B
-| foot          555-6699     1200/300          B
-| macfoo        555-6480     1200/300          A
-| sabafoo       555-2127     1200/300          C
```

This example prints each record in the file `BBS-list` whose first field contains the string `foo'. The operator `~' is called a matching operator (see section How to Use Regular Expressions); it tests whether a string (here, the field `$1`) matches a given regular expression.

By contrast, the following example looks for `foo' in *the entire record* and prints the first field and the last field for each input record containing a match.


```
$ awk '/foo/ { print $1, $NF }' BBS-list
-| fooey B
-| foot B
-| macfoo A
-| sabafoo C
```

Non-constant Field Numbers

The number of a field does not need to be a constant. Any expression in the `awk` language can be used after a `$' to refer to a field. The value of the expression specifies the field number. If the value is a string, rather than a number, it is converted to a number. Consider this example:

```
awk '{ print $NR }'
```

Recall that `NR` is the number of records read so far: one in the first record, two in the second, etc. So this example prints the first field of the first record, the second field of the second record, and so on. For the twentieth record, field number 20 is printed; most likely, the record has fewer than 20 fields, so this prints a blank line.

Here is another example of using expressions as field numbers:

```
awk '{ print $(2*2) }' BBS-list
```

`awk` must evaluate the expression `(2*2)' and use its value as the number of the field to print. The `*' sign represents multiplication, so the expression `2*2' evaluates to four. The parentheses are used so that the multiplication is done before the `$' operation; they are necessary whenever there is a binary operator in the field-number expression. This example, then, prints the hours of operation (the fourth field) for every line of the file `BBS-list'. (All of the `awk` operators are listed, in order of decreasing precedence, in section Operator Precedence (How Operators Nest).)

If the field number you compute is zero, you get the entire record. Thus, `$(2-2)` has the same value as `$0`. Negative field numbers are not allowed; trying to reference one will usually terminate your running `awk` program. (The POSIX standard does not define what happens when you reference a negative field number. `gawk` will notice this and terminate your program. Other `awk` implementations may behave differently.)

As mentioned in section Examining Fields, the number of fields in the current record is stored in the built-in variable `NF` (also see section Built-in Variables). The expression `$NF` is not a special feature: it is the direct consequence of evaluating `NF` and using its value as a field number.

Changing the Contents of a Field

You can change the contents of a field as seen by `awk` within an `awk` program; this changes what `awk` perceives as the current input record. (The actual input is untouched; `awk` *never* modifies the input file.)

Consider this example and its output:

```
$ awk '{ $3 = $2 - 10; print $2, $3 }' inventory-shipped
-| 13 3
-| 15 5
-| 15 5
...
```

The `-' sign represents subtraction, so this program reassigns field three, `$3', to be the value of field two minus ten, `$2 - 10'. (See section Arithmetic Operators.) Then field two, and the new value for field three, are printed.

In order for this to work, the text in field `$2' must make sense as a number; the string of characters must be converted to a number in order for the computer to do arithmetic on it. The number resulting from the subtraction is converted back to a string of characters which then becomes field three. See section Conversion of Strings and Numbers.

When you change the value of a field (as perceived by `awk'), the text of the input record is recalculated to contain the new field where the old one was. Therefore, `$0' changes to reflect the altered field. Thus, this program prints a copy of the input file, with 10 subtracted from the second field of each line.

```
$ awk '{ $2 = $2 - 10; print $0 }' inventory-shipped
-| Jan 3 25 15 115
-| Feb 5 32 24 226
-| Mar 5 24 34 228
...
```

You can also assign contents to fields that are out of range. For example:

```
$ awk '{ $6 = ($5 + $4 + $3 + $2)
>        print $6 }' inventory-shipped
-| 168
-| 297
-| 301
...
```

We've just created `$6', whose value is the sum of fields `$2', `$3', `$4', and `$5'. The `+' sign represents addition. For the file `inventory-shipped', `$6' represents the total number of parcels shipped for a particular month.

Creating a new field changes `awk''s internal copy of the current input record--the value of `$0'. Thus, if you do `print $0' after adding a field, the record printed includes the new field, with the appropriate number of field separators between it and the previously existing fields.

This recomputation affects and is affected by `NF' (the number of fields; see section Examining Fields), and by a feature that has not been discussed yet, the output field separator, `OFS', which is used to separate the fields (see section Output Separators). For example, the value of `NF' is set to the number of

the highest field you create.

Note, however, that merely *referencing* an out-of-range field does *not* change the value of either `$0` or `NF`. Referencing an out-of-range field only produces an empty string. For example:

```
if ($(NF+1) != "")
    print "can't happen"
else
    print "everything is normal"
```

should print `everything is normal', because `NF+1` is certain to be out of range. (See section The `if-else` Statement, for more information about `awk`'s `if-else` statements. See section Variable Typing and Comparison Expressions, for more information about the `!=' operator.)

It is important to note that making an assignment to an existing field will change the value of `$0`, but will not change the value of `NF`, even when you assign the empty string to a field. For example:

```
$ echo a b c d | awk '{ OFS = ":"; $2 = ""
>                       print $0; print NF }'
-| a::c:d
-| 4
```

The field is still there; it just has an empty value. You can tell because there are two colons in a row.

This example shows what happens if you create a new field.

```
$ echo a b c d | awk '{ OFS = ":"; $2 = ""; $6 = "new"
>                       print $0; print NF }'
-| a::c:d::new
-| 6
```

The intervening field, `$5` is created with an empty value (indicated by the second pair of adjacent colons), and `NF` is updated with the value six.

Specifying How Fields are Separated

This section is rather long; it describes one of the most fundamental operations in `awk`.

The Basics of Field Separating

The field separator, which is either a single character or a regular expression, controls the way `awk` splits an input record into fields. `awk` scans the input record for character sequences that match the separator; the fields themselves are the text between the matches.

In the examples below, we use the bullet symbol "*" to represent spaces in the output.

If the field separator is `oo', then the following line:

```
moo goo gai pan
```

would be split into three fields: `m', `*g' and `*gai*pan'. Note the leading spaces in the values of the second and third fields.

The field separator is represented by the built-in variable `FS`. Shell programmers take note! `awk` does *not* use the name `IFS` which is used by the POSIX compatible shells (such as the Bourne shell, `sh`, or the GNU Bourne-Again Shell, Bash).

You can change the value of `FS` in the `awk` program with the assignment operator, `=' (see section Assignment Expressions). Often the right time to do this is at the beginning of execution, before any input has been processed, so that the very first record will be read with the proper separator. To do this, use the special `BEGIN` pattern (see section The `BEGIN` and `END` Special Patterns). For example, here we set the value of `FS` to the string `","`:

```
awk 'BEGIN { FS = "," } ; { print $2 }'
```

Given the input line,

```
John Q. Smith, 29 Oak St., Walamazoo, MI 42139
```

this `awk` program extracts and prints the string `*29*Oak*St.'.

Sometimes your input data will contain separator characters that don't separate fields the way you thought they would. For instance, the person's name in the example we just used might have a title or suffix attached, such as `John Q. Smith, LXIX'. From input containing such a name:

```
John Q. Smith, LXIX, 29 Oak St., Walamazoo, MI 42139
```

the above program would extract `*LXIX', instead of `*29*Oak*St.'. If you were expecting the program to print the address, you would be surprised. The moral is: choose your data layout and separator characters carefully to prevent such problems.

As you know, normally, fields are separated by whitespace sequences (spaces and tabs), not by single spaces: two spaces in a row do not delimit an empty field. The default value of the field separator `FS` is a string containing a single space, `" "`. If this value were interpreted in the usual way, each space character would separate fields, so two spaces in a row would make an empty field between them. The reason this does not happen is that a single space as the value of `FS` is a special case: it is taken to specify the default manner of delimiting fields.

If `FS` is any other single character, such as `","`, then each occurrence of that character separates two fields. Two consecutive occurrences delimit an empty field. If the character occurs at the beginning or the end of the line, that too delimits an empty field. The space character is the only single character which does not follow these rules.

Using Regular Expressions to Separate Fields

The previous subsection discussed the use of single characters or simple strings as the value of FS. More generally, the value of FS may be a string containing any regular expression. In this case, each match in the record for the regular expression separates fields. For example, the assignment:

```
FS = ", \t"
```

makes every area of an input line that consists of a comma followed by a space and a tab, into a field separator. (`\t' is an escape sequence that stands for a tab; see section Escape Sequences, for the complete list of similar escape sequences.)

For a less trivial example of a regular expression, suppose you want single spaces to separate fields the way single commas were used above. You can set FS to "[ ]" (left bracket, space, right bracket). This regular expression matches a single space and nothing else (see section Regular Expressions).

There is an important difference between the two cases of `FS = " "' (a single space) and `FS = "[\t]+"' (left bracket, space, backslash, "t", right bracket, which is a regular expression matching one or more spaces or tabs). For both values of FS, fields are separated by runs of spaces and/or tabs. However, when the value of FS is " ", awk will first strip leading and trailing whitespace from the record, and then decide where the fields are.

For example, the following pipeline prints `b':

```
$ echo ' a b c d ' | awk '{ print $2 }'
-| b
```

However, this pipeline prints `a' (note the extra spaces around each letter):

```
$ echo ' a  b  c  d ' | awk 'BEGIN { FS = "[ \t]+" }
>                                  { print $2 }'
-| a
```

In this case, the first field is null, or empty.

The stripping of leading and trailing whitespace also comes into play whenever $0 is recomputed. For instance, study this pipeline:

```
$ echo '   a b c d' | awk '{ print; $2 = $2; print }'
-|    a b c d
-| a b c d
```

The first print statement prints the record as it was read, with leading whitespace intact. The assignment to $2 rebuilds $0 by concatenating $1 through $NF together, separated by the value of OFS. Since the leading whitespace was ignored when finding $1, it is not part of the new $0. Finally, the last print statement prints the new $0.

Making Each Character a Separate Field

There are times when you may want to examine each character of a record separately. In gawk, this is easy to do, you simply assign the null string ("") to FS. In this case, each individual character in the record will become a separate field. Here is an example:

```
echo a b | gawk 'BEGIN { FS = "" }
                 {
                     for (i = 1; i <= NF; i = i + 1)
                         print "Field", i, "is", $i
                 }'
```

The output from this is:

```
Field 1 is a
Field 2 is
Field 3 is b
```

Traditionally, the behavior for FS equal to "" was not defined. In this case, Unix awk would simply treat the entire record as only having one field (d.c.). In compatibility mode (see section Command Line Options), if FS is the null string, then gawk will also behave this way.

Setting FS from the Command Line

FS can be set on the command line. You use the `-F' option to do so. For example:

```
awk -F, 'program' input-files
```

sets FS to be the `,' character. Notice that the option uses a capital `F'. Contrast this with `-f', which specifies a file containing an awk program. Case is significant in command line options: the `-F' and `-f' options have nothing to do with each other. You can use both options at the same time to set the FS variable *and* get an awk program from a file.

The value used for the argument to `-F' is processed in exactly the same way as assignments to the built-in variable FS. This means that if the field separator contains special characters, they must be escaped appropriately. For example, to use a `\' as the field separator, you would have to type:

```
# same as FS = "\\"
awk -F\\\\ '...' files ...
```

Since `\' is used for quoting in the shell, awk will see `-F\\'. Then awk processes the `\\' for escape characters (see section Escape Sequences), finally yielding a single `\' to be used for the field separator.

As a special case, in compatibility mode (see section Command Line Options), if the argument to `-F' is `t', then FS is set to the tab character. This is because if you type `-F\t' at the shell, without any quotes, the `\' gets deleted, so awk figures that you really want your fields to be separated with tabs, and not `t's.

Use `-v FS="t"' on the command line if you really do want to separate your fields with `t's (see section Command Line Options).

For example, let's use an awk program file called `baud.awk' that contains the pattern /300/, and the action `print $1'. Here is the program:

```
/300/   { print $1 }
```

Let's also set FS to be the `-' character, and run the program on the file `BBS-list'. The following command prints a list of the names of the bulletin boards that operate at 300 baud and the first three digits of their phone numbers:

```
$ awk -F- -f baud.awk BBS-list
-| aardvark     555
-| alpo
-| barfly       555
...
```

Note the second line of output. In the original file (see section Data Files for the Examples), the second line looked like this:

```
alpo-net     555-3412     2400/1200/300     A
```

The `-' as part of the system's name was used as the field separator, instead of the `-' in the phone number that was originally intended. This demonstrates why you have to be careful in choosing your field and record separators.

On many Unix systems, each user has a separate entry in the system password file, one line per user. The information in these lines is separated by colons. The first field is the user's logon name, and the second is the user's encrypted password. A password file entry might look like this:

```
arnold:xyzzy:2076:10:Arnold Robbins:/home/arnold:/bin/sh
```

The following program searches the system password file, and prints the entries for users who have no password:

```
awk -F: '$2 == ""' /etc/passwd
```

Field Splitting Summary

According to the POSIX standard, awk is supposed to behave as if each record is split into fields at the time that it is read. In particular, this means that you can change the value of FS after a record is read, and the value of the fields (i.e. how they were split) should reflect the old value of FS, not the new one.

However, many implementations of awk do not work this way. Instead, they defer splitting the fields until a field is actually referenced. The fields will be split using the *current* value of FS! (d.c.) This

behavior can be difficult to diagnose. The following example illustrates the difference between the tw methods. (The `sed`(5) command prints just the first line of `/etc/passwd'.)

```
sed 1q /etc/passwd | awk '{ FS = ":" ; print $1 }'
```

will usually print

```
root
```

on an incorrect implementation of `awk`, while `gawk` will print something like

```
root:nSijPlPhZZwgE:0:0:Root:/:
```

The following table summarizes how fields are split, based on the value of `FS`. (`==' means "is equal to.")

`FS == " "`
: Fields are separated by runs of whitespace. Leading and trailing whitespace are ignored. This is the default.

`FS == any other single character`
: Fields are separated by each occurrence of the character. Multiple successive occurrences delimit empty fields, as do leading and trailing occurrences. The character can even be a regexp metacharacter; it does not need to be escaped.

`FS == regexp`
: Fields are separated by occurrences of characters that match regexp. Leading and trailing matches of regexp delimit empty fields.

`FS == ""`
: Each individual character in the record becomes a separate field.

Reading Fixed-width Data

(This section discusses an advanced, experimental feature. If you are a novice `awk` user, you may wish to skip it on the first reading.)

`gawk` version 2.13 introduced a new facility for dealing with fixed-width fields with no distinctive field separator. Data of this nature arises, for example, in the input for old FORTRAN programs where numbers are run together; or in the output of programs that did not anticipate the use of their output as input for other programs.

An example of the latter is a table where all the columns are lined up by the use of a variable number of spaces and *empty fields are just spaces*. Clearly, `awk`'s normal field splitting based on `FS` will not work well in this case. Although a portable `awk` program can use a series of `substr` calls on `$0` (see section Built-in Functions for String Manipulation), this is awkward and inefficient for a large number of fields.

The splitting of an input record into fixed-width fields is specified by assigning a string containing

space-separated numbers to the built-in variable `FIELDWIDTHS`. Each number specifies the width of a field *including* columns between fields. If you want to ignore the columns between fields, you can specify the width as a separate field that is subsequently ignored.

The following data is the output of the Unix `w` utility. It is useful to illustrate the use of `FIELDWIDTHS`.

```
 10:06pm  up 21 days, 14:04,  23 users
User     tty       login  idle   JCPU   PCPU  what
hzuo     ttyV0     8:58pm            9      5  vi p24.tex
hzang    ttyV3     6:37pm    50                -csh
eklye    ttyV5     9:53pm             7      1  em thes.tex
dportein ttyV6     8:17pm  1:47                -csh
gierd    ttyD3    10:00pm     1                elm
dave     ttyD4     9:47pm             4      4  w
brent    ttyp0    26Jun91  4:46  26:46   4:41  bash
dave     ttyq4    26Jun9115days     46     46  wnewmail
```

The following program takes the above input, converts the idle time to number of seconds and prints out the first two fields and the calculated idle time. (This program uses a number of `awk` features that haven't been introduced yet.)

```
BEGIN  { FIELDWIDTHS = "9 6 10 6 7 7 35" }
NR > 2 {
    idle = $4
    sub(/^  */, "", idle)   # strip leading spaces
    if (idle == "")
        idle = 0
    if (idle ~ /:/) {
        split(idle, t, ":")
        idle = t[1] * 60 + t[2]
    }
    if (idle ~ /days/)
        idle *= 24 * 60 * 60

    print $1, $2, idle
}
```

Here is the result of running the program on the data:

```
hzuo      ttyV0  0
hzang     ttyV3  50
eklye     ttyV5  0
dportein  ttyV6  107
gierd     ttyD3  1
dave      ttyD4  0
brent     ttyp0  286
```


```
dave      ttyq4  1296000
```

Another (possibly more practical) example of fixed-width input data would be the input from a deck of balloting cards. In some parts of the United States, voters mark their choices by punching holes in computer cards. These cards are then processed to count the votes for any particular candidate or on any particular issue. Since a voter may choose not to vote on some issue, any column on the card may be empty. An `awk` program for processing such data could use the `FIELDWIDTHS` feature to simplify reading the data. (Of course, getting `gawk` to run on a system with card readers is another story!)

Assigning a value to `FS` causes `gawk` to return to using `FS` for field splitting. Use `FS = FS' to make this happen, without having to know the current value of `FS`.

This feature is still experimental, and may evolve over time. Note that in particular, `gawk` does not attempt to verify the sanity of the values used in the value of `FIELDWIDTHS`.

Multiple-Line Records

In some data bases, a single line cannot conveniently hold all the information in one entry. In such cases, you can use multi-line records.

The first step in doing this is to choose your data format: when records are not defined as single lines, how do you want to define them? What should separate records?

One technique is to use an unusual character or string to separate records. For example, you could use the formfeed character (written `\f' in `awk`, as in C) to separate them, making each record a page of the file. To do this, just set the variable `RS` to `"\f"` (a string containing the formfeed character). Any other character could equally well be used, as long as it won't be part of the data in a record.

Another technique is to have blank lines separate records. By a special dispensation, an empty string as the value of `RS` indicates that records are separated by one or more blank lines. If you set `RS` to the empty string, a record always ends at the first blank line encountered. And the next record doesn't start until the first non-blank line that follows--no matter how many blank lines appear in a row, they are considered one record-separator.

You can achieve the same effect as `RS = "" by assigning the string `"\n\n+"` to `RS`. This regexp matches the newline at the end of the record, and one or more blank lines after the record. In addition, a regular expression always matches the longest possible sequence when there is a choice (see section How Much Text Matches?) So the next record doesn't start until the first non-blank line that follows--no matter how many blank lines appear in a row, they are considered one record-separator.

There is an important difference between `RS = "" and `RS = "\n\n+"'. In the first case, leading newlines in the input data file are ignored, and if a file ends without extra blank lines after the last record, the final newline is removed from the record. In the second case, this special processing is not done (d.c.).

Now that the input is separated into records, the second step is to separate the fields in the record. One way to do this is to divide each of the lines into fields in the normal manner. This happens by default as the result of a special feature: when `RS` is set to the empty string, the newline character *always* acts as a field separator. This is in addition to whatever field separations result from `FS`.

The original motivation for this special exception was probably to provide useful behavior in the default case (i.e. FS is equal to " "). This feature can be a problem if you really don't want the newline character to separate fields, since there is no way to prevent it. However, you can work around this by using the `split` function to break up the record manually (see section Built-in Functions for String Manipulation).

Another way to separate fields is to put each field on a separate line: to do this, just set the variable FS to the string `"\n"`. (This simple regular expression matches a single newline.)

A practical example of a data file organized this way might be a mailing list, where each entry is separated by blank lines. If we have a mailing list in a file named `` `addresses' ``, that looks like this:

```
Jane Doe
123 Main Street
Anywhere, SE 12345-6789

John Smith
456 Tree-lined Avenue
Smallville, MW 98765-4321

...
```

A simple program to process this file would look like this:

```
# addrs.awk -- simple mailing list program

# Records are separated by blank lines.
# Each line is one field.
BEGIN { RS = "" ; FS = "\n" }

{
      print "Name is:", $1
      print "Address is:", $2
      print "City and State are:", $3
      print ""
}
```

Running the program produces the following output:

```
$ awk -f addrs.awk addresses
-| Name is: Jane Doe
-| Address is: 123 Main Street
-| City and State are: Anywhere, SE 12345-6789
-|
-| Name is: John Smith
-| Address is: 456 Tree-lined Avenue
```


```
-| City and State are: Smallville, MW 98765-4321
-|
...
```

See section Printing Mailing Labels, for a more realistic program that deals with address lists.

The following table summarizes how records are split, based on the value of RS. (`==' means "is equal to.")

`RS == "\n"`

Records are separated by the newline character (`\n'). In effect, every line in the data file is a separate record, including blank lines. This is the default.

`RS == any single character`

Records are separated by each occurrence of the character. Multiple successive occurrences delimit empty records.

`RS == ""`

Records are separated by runs of blank lines. The newline character always serves as a field separator, in addition to whatever value FS may have. Leading and trailing newlines in a file are ignored.

`RS == regexp`

Records are separated by occurrences of characters that match regexp. Leading and trailing matches of regexp delimit empty records.

In all cases, gawk sets RT to the input text that matched the value specified by RS.

Explicit Input with `getline`

So far we have been getting our input data from awk's main input stream--either the standard input (usually your terminal, sometimes the output from another program) or from the files specified on the command line. The awk language has a special built-in command called `getline` that can be used to read input under your explicit control.

Introduction to `getline`

This command is used in several different ways, and should *not* be used by beginners. It is covered here because this is the chapter on input. The examples that follow the explanation of the `getline` command include material that has not been covered yet. Therefore, come back and study the `getline` command *after* you have reviewed the rest of this book and have a good knowledge of how awk works.

`getline` returns one if it finds a record, and zero if the end of the file is encountered. If there is some error in getting a record, such as a file that cannot be opened, then `getline` returns -1. In this case, gawk sets the variable ERRNO to a string describing the error that occurred.

In the following examples, command stands for a string value that represents a shell command.

Using `getline` with No Arguments

The `getline` command can be used without arguments to read input from the current input file. All it does in this case is read the next input record and split it up into fields. This is useful if you've finished processing the current record, but you want to do some special processing *right now* on the next record. Here's an example:

```
awk '{
     if ((t = index($0, "/*")) != 0) {
          # value will be "" if t is 1
          tmp = substr($0, 1, t - 1)
          u = index(substr($0, t + 2), "*/")
          while (u == 0) {
               if (getline <= 0) {
                    m = "unexpected EOF or error"
                    m = (m ": " ERRNO)
                    print m > "/dev/stderr"
                    exit
               }
               t = -1
               u = index($0, "*/")
          }
          # substr expression will be "" if */
          # occurred at end of line
          $0 = tmp substr($0, t + u + 3)
     }
     print $0
}'
```

This `awk` program deletes all C-style comments, `/* ... */', from the input. By replacing the `print $0' with other statements, you could perform more complicated processing on the decommented input, like searching for matches of a regular expression. This program has a subtle problem--it does not work if one comment ends and another begins on the same line.

This form of the `getline` command sets `NF` (the number of fields; see section Examining Fields), `NR` (the number of records read so far; see section How Input is Split into Records), `FNR` (the number of records read from this input file), and the value of `$0`.

Note: the new value of `$0` is used in testing the patterns of any subsequent rules. The original value of `$0` that triggered the rule which executed `getline` is lost (d.c.). By contrast, the `next` statement reads a new record but immediately begins processing it normally, starting with the first rule in the program. See section The `next` Statement.

Using `getline` Into a Variable

You can use `getline var' to read the next record from `awk`'s input into the variable var. No other processing is done.

For example, suppose the next line is a comment, or a special string, and you want to read it, without triggering any rules. This form of `getline` allows you to read that line and store it in a variable so that the main read-a-line-and-check-each-rule loop of `awk` never sees it.

The following example swaps every two lines of input. For example, given:

```
wan
tew
free
phore
```

it outputs:

```
tew
wan
phore
free
```

Here's the program:

```
awk '{
     if ((getline tmp) > 0) {
          print tmp
          print $0
     } else
          print $0
}'
```

The `getline` command used in this way sets only the variables `NR` and `FNR` (and of course, var). The record is not split into fields, so the values of the fields (including `$0`) and the value of `NF` do not change.

Using `getline` from a File

Use `getline < file' to read the next record from the file file. Here file is a string-valued expression that specifies the file name. `< file' is called a redirection since it directs input to come from a different place.

For example, the following program reads its input record from the file `` `secondary.input' `` when it encounters a first field with a value equal to 10 in the current input file.

```
awk '{
```


```
    if ($1 == 10) {
         getline < "secondary.input"
         print
    } else
         print
}'
```

Since the main input stream is not used, the values of `NR` and `FNR` are not changed. But the record read is split into fields in the normal manner, so the values of `$0` and other fields are changed. So is the value of `NF`.

Using `getline` Into a Variable from a File

Use `getline var < file' to read input the file file and put it in the variable var. As above, file is a string-valued expression that specifies the file from which to read.

In this version of `getline`, none of the built-in variables are changed, and the record is not split into fields. The only variable changed is var.

For example, the following program copies all the input files to the output, except for records that say `@include filename'. Such a record is replaced by the contents of the file filename.

```
awk '{
     if (NF == 2 && $1 == "@include") {
          while ((getline line < $2) > 0)
               print line
          close($2)
     } else
          print
}'
```

Note here how the name of the extra input file is not built into the program; it is taken directly from the data, from the second field on the `@include' line.

The `close` function is called to ensure that if two identical `@include' lines appear in the input, the entire specified file is included twice. See section Closing Input and Output Files and Pipes.

One deficiency of this program is that it does not process nested `@include' statements (`@include' statements in included files) the way a true macro preprocessor would. See section An Easy Way to Use Library Functions, for a program that does handle nested `@include' statements.

Using `getline` from a Pipe

You can pipe the output of a command into `getline`, using `command | getline'. In this case, the string command is run as a shell command and its output is piped into `awk` to be used as input. This form of `getline` reads one record at a time from the pipe.

For example, the following program copies its input to its output, except for lines that begin with `@execute', which are replaced by the output produced by running the rest of the line as a shell command:

```
awk '{
     if ($1 == "@execute") {
          tmp = substr($0, 10)
          while ((tmp | getline) > 0)
               print
          close(tmp)
     } else
          print
}'
```

The `close` function is called to ensure that if two identical `@execute' lines appear in the input, the command is run for each one. See section Closing Input and Output Files and Pipes.

Given the input:

```
foo
bar
baz
@execute who
bletch
```

the program might produce:

```
foo
bar
baz
arnold     ttyv0   Jul 13 14:22
miriam     ttyp0   Jul 13 14:23     (murphy:0)
bill       ttyp1   Jul 13 14:23     (murphy:0)
bletch
```

Notice that this program ran the command `who` and printed the result. (If you try this program yourself, you will of course get different results, showing you who is logged in on your system.)

This variation of `getline` splits the record into fields, sets the value of `NF` and recomputes the value of `$0`. The values of `NR` and `FNR` are not changed.

Using `getline` Into a Variable from a Pipe

When you use `command | getline var', the output of the command command is sent through a pipe to `getline` and into the variable var. For example, the following program reads the current date and time into the variable `current_time`, using the `date` utility, and then prints it.


```
awk 'BEGIN {
     "date" | getline current_time
     close("date")
     print "Report printed on " current_time
}'
```

In this version of `getline`, none of the built-in variables are changed, and the record is not split into fields.

Summary of `getline` Variants

With all the forms of `getline`, even though `$0` and `NF`, may be updated, the record will not be tested against all the patterns in the `awk` program, in the way that would happen if the record were read normally by the main processing loop of `awk`. However the new record is tested against any subsequent rules.

Many `awk` implementations limit the number of pipelines an `awk` program may have open to just one! In `gawk`, there is no such limit. You can open as many pipelines as the underlying operating system will permit.

The following table summarizes the six variants of `getline`, listing which built-in variables are set by each one.

`getline`
: sets `$0`, `NF`, `FNR`, and `NR`.

`getline var`
: sets var, `FNR`, and `NR`.

`getline < file`
: sets `$0`, and `NF`.

`getline var < file`
: sets var.

`command | getline`
: sets `$0`, and `NF`.

`command | getline var`
: sets var.

Go to the previous, next section.

Go to the previous, next section.

Printing Output

One of the most common actions is to print, or output, some or all of the input. You use the `print` statement for simple output. You use the `printf` statement for fancier formatting. Both are described in this chapter.

The `print` Statement

The `print` statement does output with simple, standardized formatting. You specify only the strings or numbers to be printed, in a list separated by commas. They are output, separated by single spaces, followed by a newline. The statement looks like this:

```
print item1, item2, ...
```

The entire list of items may optionally be enclosed in parentheses. The parentheses are necessary if any of the item expressions uses the `>' relational operator; otherwise it could be confused with a redirection (see section Redirecting Output of `print` and `printf`).

The items to be printed can be constant strings or numbers, fields of the current record (such as `$1`), variables, or any `awk` expressions. Numeric values are converted to strings, and then printed.

The `print` statement is completely general for computing *what* values to print. However, with two exceptions, you cannot specify *how* to print them--how many columns, whether to use exponential notation or not, and so on. (For the exceptions, see section Output Separators, and section Controlling Numeric Output with `print`.) For that, you need the `printf` statement (see section Using `printf` Statements for Fancier Printing).

The simple statement `print' with no items is equivalent to `print $0': it prints the entire current record. To print a blank line, use `print "", where `""` is the empty string.

To print a fixed piece of text, use a string constant such as `"Don't Panic"` as one item. If you forget to use the double-quote characters, your text will be taken as an `awk` expression, and you will probably get an error. Keep in mind that a space is printed between any two items.

Each `print` statement makes at least one line of output. But it isn't limited to one line. If an item value is a string that contains a newline, the newline is output along with the rest of the string. A single `print` can make any number of lines this way.

Examples of `print` Statements

Here is an example of printing a string that contains embedded newlines (the `\n' is an escape sequence, used to represent the newline character; see section Escape Sequences):

```
$ awk 'BEGIN { print "line one\nline two\nline three" }'
-| line one
-| line two
-| line three
```

Here is an example that prints the first two fields of each input record, with a space between them:

```
$ awk '{ print $1, $2 }' inventory-shipped
-| Jan 13
-| Feb 15
-| Mar 15
...
```

A common mistake in using the `print` statement is to omit the comma between two items. This often has the effect of making the items run together in the output, with no space. The reason for this is that juxtaposing two string expressions in `awk` means to concatenate them. Here is the same program, without the comma:

```
$ awk '{ print $1 $2 }' inventory-shipped
-| Jan13
-| Feb15
-| Mar15
...
```

To someone unfamiliar with the file `inventory-shipped', neither example's output makes much sense. A heading line at the beginning would make it clearer. Let's add some headings to our table of months (`$1`) and green crates shipped (`$2`). We do this using the `BEGIN` pattern (see section The `BEGIN` and `END` Special Patterns) to force the headings to be printed only once:

```
awk 'BEGIN {  print "Month Crates"
              print "-----  ------" }
           {  print $1, $2 }' inventory-shipped
```

Did you already guess what happens? When run, the program prints the following:

```
Month Crates
-----  ------
Jan 13
Feb 15
Mar 15
```


```
...
```

The headings and the table data don't line up! We can fix this by printing some spaces between the two fields:

```
awk 'BEGIN { print "Month Crates"
             print "-----  ------" }
           { print $1, "     ", $2 }' inventory-shipped
```

You can imagine that this way of lining up columns can get pretty complicated when you have many columns to fix. Counting spaces for two or three columns can be simple, but more than this and you can get lost quite easily. This is why the `printf` statement was created (see section Using `printf` Statements for Fancier Printing); one of its specialties is lining up columns of data.

As a side point, you can continue either a `print` or `printf` statement simply by putting a newline after any comma (see section `awk` Statements Versus Lines).

Output Separators

As mentioned previously, a `print` statement contains a list of items, separated by commas. In the output, the items are normally separated by single spaces. This need not be the case; a single space is only the default. You can specify any string of characters to use as the output field separator by setting the built-in variable `OFS`. The initial value of this variable is the string `" "`, that is, a single space.

The output from an entire `print` statement is called an output record. Each `print` statement outputs one output record and then outputs a string called the output record separator. The built-in variable `ORS` specifies this string. The initial value of `ORS` is the string `"\n"`, i.e. a newline character; thus, normally each `print` statement makes a separate line.

You can change how output fields and records are separated by assigning new values to the variables `OFS` and/or `ORS`. The usual place to do this is in the `BEGIN` rule (see section The `BEGIN` and `END` Special Patterns), so that it happens before any input is processed. You may also do this with assignments on the command line, before the names of your input files, or using the `-v' command line option (see section Command Line Options).

The following example prints the first and second fields of each input record separated by a semicolon, with a blank line added after each line:

```
$ awk 'BEGIN { OFS = ";"; ORS = "\n\n" }
>            { print $1, $2 }' BBS-list
-| aardvark;555-5553
-|
-| alpo-net;555-3412
-|
-| barfly;555-7685
...
```

If the value of `ORS` does not contain a newline, all your output will be run together on a single line, unless you output newlines some other way.

Controlling Numeric Output with `print`

When you use the `print` statement to print numeric values, `awk` internally converts the number to a string of characters, and prints that string. `awk` uses the `sprintf` function to do this conversion (see section Built-in Functions for String Manipulation). For now, it suffices to say that the `sprintf` function accepts a format specification that tells it how to format numbers (or strings), and that there are a number of different ways in which numbers can be formatted. The different format specifications are discussed more fully in section Format-Control Letters.

The built-in variable `OFMT` contains the default format specification that `print` uses with `sprintf` when it wants to convert a number to a string for printing. The default value of `OFMT` is `"%.6g"`. By supplying different format specifications as the value of `OFMT`, you can change how `print` will print your numbers. As a brief example:

```
$ awk 'BEGIN {
>   OFMT = "%.0f"  # print numbers as integers (rounds)
>   print 17.23 }'
-| 17
```

According to the POSIX standard, `awk`'s behavior will be undefined if `OFMT` contains anything but a floating point conversion specification (d.c.).

Using `printf` Statements for Fancier Printing

If you want more precise control over the output format than `print` gives you, use `printf`. With `printf` you can specify the width to use for each item, and you can specify various formatting choices for numbers (such as what radix to use, whether to print an exponent, whether to print a sign, and how many digits to print after the decimal point). You do this by supplying a string, called the format string, which controls how and where to print the other arguments.

Introduction to the `printf` Statement

The `printf` statement looks like this:

```
printf format, item1, item2, ...
```

The entire list of arguments may optionally be enclosed in parentheses. The parentheses are necessary if any of the item expressions use the `>' relational operator; otherwise it could be confused with a redirection (see section Redirecting Output of `print` and `printf`).

The difference between `printf` and `print` is the format argument. This is an expression whose value

is taken as a string; it specifies how to output each of the other arguments. It is called the format strin

The format string is very similar to that in the ANSI C library function `printf`. Most of format is text to be output verbatim. Scattered among this text are format specifiers, one per item. Each format specifier says to output the next item in the argument list at that place in the format.

The `printf` statement does not automatically append a newline to its output. It outputs only what the format string specifies. So if you want a newline, you must include one in the format string. The output separator variables `OFS` and `ORS` have no effect on `printf` statements. For example:

```
BEGIN {
   ORS = "\nOUCH!\n"; OFS = "!"
   msg = "Don't Panic!"; printf "%s\n", msg
}
```

This program still prints the familiar `Don't Panic!' message.

Format-Control Letters

A format specifier starts with the character `%' and ends with a format-control letter; it tells the `printf` statement how to output one item. (If you actually want to output a `%', write `%%'.) The format-control letter specifies what kind of value to print. The rest of the format specifier is made up of optional modifiers which are parameters to use, such as the field width.

Here is a list of the format-control letters:

`c`

This prints a number as an ASCII character. Thus, `printf "%c", 65' outputs the letter `A'. The output for a string value is the first character of the string.

`d`

`i`

These are equivalent. They both print a decimal integer. The `%i' specification is for compatibility with ANSI C.

`e`

`E`

This prints a number in scientific (exponential) notation. For example,

```
printf "%4.3e\n", 1950
```

prints `1.950e+03', with a total of four significant figures of which three follow the decimal point. The `4.3' are modifiers, discussed below. `%E' uses `E' instead of `e' in the output.

`f`

This prints a number in floating point notation. For example,

```
printf "%4.3f", 1950
```

prints `1950.000', with a total of four significant figures of which three follow the decimal po
The `4.3' are modifiers, discussed below.

`g`
`G`

This prints a number in either scientific notation or floating point notation, whichever uses fewer characters. If the result is printed in scientific notation, `%G' uses `E' instead of `e'.

`o`

This prints an unsigned octal integer. (In octal, or base-eight notation, the digits run from `0' to `7'; the decimal number eight is represented as `10' in octal.)

`s`

This prints a string.

`x`
`X`

This prints an unsigned hexadecimal integer. (In hexadecimal, or base-16 notation, the digits are `0' through `9' and `a' through `f'. The hexadecimal digit `f' represents the decimal number 15.) `%X' uses the letters `A' through `F' instead of `a' through `f'.

`%`

This isn't really a format-control letter, but it does have a meaning when used after a `%': the sequence `%%' outputs one `%'. It does not consume an argument, and it ignores any modifiers.

When using the integer format-control letters for values that are outside the range of a C `long` integer, `gawk` will switch to the `%g' format specifier. Other versions of `awk` may print invalid values, or do something else entirely (d.c.).

Modifiers for `printf` Formats

A format specification can also include modifiers that can control how much of the item's value is printed and how much space it gets. The modifiers come between the `%' and the format-control letter. In the examples below, we use the bullet symbol "*" to represent spaces in the output. Here are the possible modifiers, in the order in which they may appear:

`-`

The minus sign, used before the width modifier (see below), says to left-justify the argument within its specified width. Normally the argument is printed right-justified in the specified width. Thus,

```
printf "%-4s", "foo"
```

prints `foo*'.

`space`

For numeric conversions, prefix positive values with a space, and negative values with a minus sign.

`+`

The plus sign, used before the width modifier (see below), says to always supply a sign for numeric conversions, even if the data to be formatted is positive. The `+' overrides the space modifier.

`#`

Use an "alternate form" for certain control letters. For `%o', supply a leading zero. For `%x', and `%X', supply a leading `0x' or `0X' for a non-zero result. For `%e', `%E', and `%f', the result will always contain a decimal point. For `%g', and `%G', trailing zeros are not removed from the result.

`0`

A leading `0' (zero) acts as a flag, that indicates output should be padded with zeros instead of spaces. This applies even to non-numeric output formats (d.c.). This flag only has an effect when the field width is wider than the value to be printed.

`width`

This is a number specifying the desired minimum width of a field. Inserting any number between the `%' sign and the format control character forces the field to be expanded to this width. The default way to do this is to pad with spaces on the left. For example,

```
printf "%4s", "foo"
```

prints `*foo'.

The value of width is a minimum width, not a maximum. If the item value requires more than width characters, it can be as wide as necessary. Thus,

```
printf "%4s", "foobar"
```

prints `foobar'.

Preceding the width with a minus sign causes the output to be padded with spaces on the right, instead of on the left.

`.prec`

This is a number that specifies the precision to use when printing. For the `e', `E', and `f' formats, this specifies the number of digits you want printed to the right of the decimal point. For the `g', and `G' formats, it specifies the maximum number of significant digits. For the `d', `o', `i', `u', `x', and `X' formats, it specifies the minimum number of digits to print. For a string, it specifies the maximum number of characters from the string that should be printed. Thus,

```
printf "%.4s", "foobar"
```

prints `foob'.

The C library `printf`'s dynamic width and prec capability (for example, `"%*.*s"`) is supported. Instead of supplying explicit width and/or prec values in the format string, you pass them in the argument list. For example:


```
w = 5
p = 3
s = "abcdefg"
printf "%*.*s\n", w, p, s
```

is exactly equivalent to

```
s = "abcdefg"
printf "%5.3s\n", s
```

Both programs output `**abc'.

Earlier versions of `awk` did not support this capability. If you must use such a version, you may simulate this feature by using concatenation to build up the format string, like so:

```
w = 5
p = 3
s = "abcdefg"
printf "%" w "." p "s\n", s
```

This is not particularly easy to read, but it does work.

C programmers may be used to supplying additional `l' and `h' flags in `printf` format strings. These are not valid in `awk`. Most `awk` implementations silently ignore these flags. If `--lint' is provided on the command line (see section Command Line Options), `gawk` will warn about their use. If `--posix' is supplied, their use is a fatal error.

Examples Using `printf`

Here is how to use `printf` to make an aligned table:

```
awk '{ printf "%-10s %s\n", $1, $2 }' BBS-list
```

prints the names of bulletin boards (`$1`) of the file `` `BBS-list' `` as a string of 10 characters, left justified. It also prints the phone numbers (`$2`) afterward on the line. This produces an aligned two-column table of names and phone numbers:

```
$ awk '{ printf "%-10s %s\n", $1, $2 }' BBS-list
-| aardvark   555-5553
-| alpo-net   555-3412
-| barfly     555-7685
-| bites      555-1675
-| camelot    555-0542
-| core       555-2912
-| fooey      555-1234
```


```
-| foot       555-6699
-| macfoo     555-6480
-| sdace      555-3430
-| sabafoo    555-2127
```

Did you notice that we did not specify that the phone numbers be printed as numbers? They had to be printed as strings because the numbers are separated by a dash. If we had tried to print the phone numbers as numbers, all we would have gotten would have been the first three digits, `555'. This would have been pretty confusing.

We did not specify a width for the phone numbers because they are the last things on their lines. We don't need to put spaces after them.

We could make our table look even nicer by adding headings to the tops of the columns. To do this, we use the `BEGIN` pattern (see section The `BEGIN` and `END` Special Patterns) to force the header to be printed only once, at the beginning of the `awk` program:

```
awk 'BEGIN { print "Name      Number"
             print "----      ------" }
     { printf "%-10s %s\n", $1, $2 }' BBS-list
```

Did you notice that we mixed `print` and `printf` statements in the above example? We could have used just `printf` statements to get the same results:

```
awk 'BEGIN { printf "%-10s %s\n", "Name", "Number"
             printf "%-10s %s\n", "----", "------" }
     { printf "%-10s %s\n", $1, $2 }' BBS-list
```

By printing each column heading with the same format specification used for the elements of the column, we have made sure that the headings are aligned just like the columns.

The fact that the same format specification is used three times can be emphasized by storing it in a variable, like this:

```
awk 'BEGIN { format = "%-10s %s\n"
             printf format, "Name", "Number"
             printf format, "----", "------" }
     { printf format, $1, $2 }' BBS-list
```

See if you can use the `printf` statement to line up the headings and table data for our `inventory-shipped' example covered earlier in the section on the `print` statement (see section The `print` Statement).

Redirecting Output of `print` and `printf`

So far we have been dealing only with output that prints to the standard output, usually your terminal. Both `print` and `printf` can also send their output to other places. This is called redirection.

A redirection appears after the `print` or `printf` statement. Redirections in `awk` are written just like redirections in shell commands, except that they are written inside the `awk` program.

There are three forms of output redirection: output to a file, output appended to a file, and output through a pipe to another command. They are all shown for the `print` statement, but they work identically for `printf` also.

`print items > output-file`

This type of redirection prints the items into the output file output-file. The file name output-file can be any expression. Its value is changed to a string and then used as a file name (see section Expressions).

When this type of redirection is used, the output-file is erased before the first output is written to it. Subsequent writes to the same output-file do not erase output-file, but append to it. If output-file does not exist, then it is created.

For example, here is how an `awk` program can write a list of BBS names to a file `` `name-list' `` and a list of phone numbers to a file `` `phone-list' ``. Each output file contains one name or number per line.

```
$ awk '{ print $2 > "phone-list"
>        print $1 > "name-list" }' BBS-list
$ cat phone-list
-| 555-5553
-| 555-3412
...
$ cat name-list
-| aardvark
-| alpo-net
...
```

`print items >> output-file`

This type of redirection prints the items into the pre-existing output file output-file. The difference between this and the single-`>' redirection is that the old contents (if any) of output-file are not erased. Instead, the `awk` output is appended to the file. If output-file does not exist, then it is created.

`print items | command`

It is also possible to send output to another program through a pipe instead of into a file. This type of redirection opens a pipe to command and writes the values of items through this pipe, to another process created to execute command.

The redirection argument command is actually an `awk` expression. Its value is converted to a

string, whose contents give the shell command to be run.

For example, this produces two files, one unsorted list of BBS names and one list sorted in reverse alphabetical order:

```
awk '{ print $1 > "names.unsorted"
       command = "sort -r > names.sorted"
       print $1 | command }' BBS-list
```

Here the unsorted list is written with an ordinary redirection while the sorted list is written by piping through the `sort` utility.

This example uses redirection to mail a message to a mailing list `bug-system'. This might be useful when trouble is encountered in an `awk` script run periodically for system maintenance.

```
report = "mail bug-system"
print "Awk script failed:", $0 | report
m = ("at record number " FNR " of " FILENAME)
print m | report
close(report)
```

The message is built using string concatenation and saved in the variable `m`. It is then sent down the pipeline to the `mail` program.

We call the `close` function here because it's a good idea to close the pipe as soon as all the intended output has been sent to it. See section Closing Input and Output Files and Pipes, for more information on this. This example also illustrates the use of a variable to represent a file or command: it is not necessary to always use a string constant. Using a variable is generally a good idea, since `awk` requires you to spell the string value identically every time.

Redirecting output using `>', `>>', or `|' asks the system to open a file or pipe only if the particular file or command you've specified has not already been written to by your program, or if it has been closed since it was last written to.

Many `awk` implementations limit the number of pipelines an `awk` program may have open to just one! In `gawk`, there is no such limit. You can open as many pipelines as the underlying operating system will permit.

Special File Names in gawk

Running programs conventionally have three input and output streams already available to them for reading and writing. These are known as the standard input, standard output, and standard error output. These streams are, by default, connected to your terminal, but they are often redirected with the shell, via the `<', `<<', `>', `>>', `>&' and `|' operators. Standard error is typically used for writing error messages; the reason we have two separate streams, standard output and standard error, is so that they can be redirected separately.

In other implementations of `awk`, the only way to write an error message to standard error in an `awk` program is as follows:

```
print "Serious error detected!" | "cat 1>&2"
```

This works by opening a pipeline to a shell command which can access the standard error stream which it inherits from the `awk` process. This is far from elegant, and is also inefficient, since it requires a separate process. So people writing `awk` programs often neglect to do this. Instead, they send the error messages to the terminal, like this:

```
print "Serious error detected!" > "/dev/tty"
```

This usually has the same effect, but not always: although the standard error stream is usually the terminal, it can be redirected, and when that happens, writing to the terminal is not correct. In fact, if `awk` is run from a background job, it may not have a terminal at all. Then opening `` `/dev/tty' `` will fail.

`gawk` provides special file names for accessing the three standard streams. When you redirect input or output in `gawk`, if the file name matches one of these special names, then `gawk` directly uses the stream it stands for.

`` `/dev/stdin' ``
: The standard input (file descriptor 0).

`` `/dev/stdout' ``
: The standard output (file descriptor 1).

`` `/dev/stderr' ``
: The standard error output (file descriptor 2).

`` `/dev/fd/N' ``
: The file associated with file descriptor N. Such a file must have been opened by the program initiating the `awk` execution (typically the shell). Unless you take special pains in the shell from which you invoke `gawk`, only descriptors 0, 1 and 2 are available.

The file names `` `/dev/stdin' ``, `` `/dev/stdout' ``, and `` `/dev/stderr' `` are aliases for `` `/dev/fd/0' ``, `` `/dev/fd/1' ``, and `` `/dev/fd/2' ``, respectively, but they are more self-explanatory.

The proper way to write an error message in a `gawk` program is to use `` `/dev/stderr' ``, like this:

```
print "Serious error detected!" > "/dev/stderr"
```

`gawk` also provides special file names that give access to information about the running `gawk` process. Each of these "files" provides a single record of information. To read them more than once, you must first close them with the `close` function (see section Closing Input and Output Files and Pipes). The filenames are:

`` `/dev/pid' ``

Reading this file returns the process ID of the current process, in decimal, terminated with a newline.

`/dev/ppid'

Reading this file returns the parent process ID of the current process, in decimal, terminated with a newline.

`/dev/pgrpid'

Reading this file returns the process group ID of the current process, in decimal, terminated with a newline.

`/dev/user'

Reading this file returns a single record terminated with a newline. The fields are separated with spaces. The fields represent the following information:

$1

The return value of the `getuid` system call (the real user ID number).

$2

The return value of the `geteuid` system call (the effective user ID number).

$3

The return value of the `getgid` system call (the real group ID number).

$4

The return value of the `getegid` system call (the effective group ID number).

If there are any additional fields, they are the group IDs returned by `getgroups` system call. (Multiple groups may not be supported on all systems.)

These special file names may be used on the command line as data files, as well as for I/O redirections within an `awk` program. They may not be used as source files with the `-f' option.

Recognition of these special file names is disabled if `gawk` is in compatibility mode (see section Command Line Options).

Caution: Unless your system actually has a `/dev/fd' directory (or any of the other above listed special files), the interpretation of these file names is done by `gawk` itself. For example, using `/dev/fd/4' for output will actually write on file descriptor 4, and not on a new file descriptor that was `dup`'ed from file descriptor 4. Most of the time this does not matter; however, it is important to *not* close any of the files related to file descriptors 0, 1, and 2. If you do close one of these files, unpredictable behavior will result.

The special files that provide process-related information may disappear in a future version of `gawk`. See section Probable Future Extensions.

Closing Input and Output Files and Pipes

If the same file name or the same shell command is used with `getline` (see section Explicit Input with `getline`) more than once during the execution of an `awk` program, the file is opened (or the command is executed) only the first time. At that time, the first record of input is read from that file or command. The next time the same file or command is used in `getline`, another record is read from it, and so on.

Similarly, when a file or pipe is opened for output, the file name or command associated with it is remembered by `awk` and subsequent writes to the same file or command are appended to the previous writes. The file or pipe stays open until `awk` exits.

This implies that if you want to start reading the same file again from the beginning, or if you want to rerun a shell command (rather than reading more output from the command), you must take special steps. What you must do is use the `close` function, as follows:

```
close(filename)
```

or

```
close(command)
```

The argument filename or command can be any expression. Its value must *exactly* match the string that was used to open the file or start the command (spaces and other "irrelevant" characters included). For example, if you open a pipe with this:

```
"sort -r names" | getline foo
```

then you must close it with this:

```
close("sort -r names")
```

Once this function call is executed, the next `getline` from that file or command, or the next `print` or `printf` to that file or command, will reopen the file or rerun the command.

Because the expression that you use to close a file or pipeline must exactly match the expression used to open the file or run the command, it is good practice to use a variable to store the file name or command. The previous example would become

```
sortcom = "sort -r names"
sortcom | getline foo
...
close(sortcom)
```

This helps avoid hard-to-find typographical errors in your `awk` programs.

Here are some reasons why you might need to close an output file:

- To write a file and read it back later on in the same `awk` program. Close the file when you are finished writing it; then you can start reading it with `getline`.
- To write numerous files, successively, in the same `awk` program. If you don't close the files, eventually you may exceed a system limit on the number of open files in one process. So close each one when you are finished writing it.
- To make a command finish. When you redirect output through a pipe, the command reading the pipe normally continues to try to read input as long as the pipe is open. Often this means the command cannot really do its work until the pipe is closed. For example, if you redirect output to the `mail` program, the message is not actually sent until the pipe is closed.
- To run the same program a second time, with the same arguments. This is not the same thing as giving more input to the first run!

  For example, suppose you pipe output to the `mail` program. If you output several lines redirected to this pipe without closing it, they make a single message of several lines. By contrast, if you close the pipe after each line of output, then each line makes a separate message.

`close` returns a value of zero if the close succeeded. Otherwise, the value will be non-zero. In this case, `gawk` sets the variable `ERRNO` to a string describing the error that occurred.

If you use more files than the system allows you to have open, `gawk` will attempt to multiplex the available open files among your data files. `gawk`'s ability to do this depends upon the facilities of your operating system: it may not always work. It is therefore both good practice and good portability advice to always use `close` on your files when you are done with them.

Go to the previous, next section.

Go to the previous, next section.

Expressions

Expressions are the basic building blocks of `awk` patterns and actions. An expression evaluates to a value, which you can print, test, store in a variable or pass to a function. Additionally, an expression can assign a new value to a variable or a field, with an assignment operator.

An expression can serve as a pattern or action statement on its own. Most other kinds of statements contain one or more expressions which specify data on which to operate. As in other languages, expressions in `awk` include variables, array references, constants, and function calls, as well as combinations of these with various operators.

Constant Expressions

The simplest type of expression is the constant, which always has the same value. There are three types of constants: numeric constants, string constants, and regular expression constants.

Numeric and String Constants

A numeric constant stands for a number. This number can be an integer, a decimal fraction, or a number in scientific (exponential) notation.(6) Here are some examples of numeric constants, which all have the same value:

```
105
1.05e+2
1050e-1
```

A string constant consists of a sequence of characters enclosed in double-quote marks. For example:

```
"parrot"
```

represents the string whose contents are `parrot'. Strings in `gawk` can be of any length and they can contain any of the possible eight-bit ASCII characters including ASCII NUL (character code zero). Other `awk` implementations may have difficulty with some character codes.

Regular Expression Constants

A regexp constant is a regular expression description enclosed in slashes, such as `/^beginning and end$/`. Most regexps used in `awk` programs are constant, but the `~' and `!~' matching operators can also match computed or "dynamic" regexps (which are just ordinary strings or variables that contain a regexp).

Using Regular Expression Constants

When used on the right hand side of the `~' or `!~' operators, a regexp constant merely stands for the regexp that is to be matched.

Regexp constants (such as `/foo/') may be used like simple expressions. When a regexp constant appears by itself, it has the same meaning as if it appeared in a pattern, i.e. `($0 ~ /foo/)' (d.c.) (see section Expressions as Patterns). This means that the two code segments,

```
if ($0 ~ /barfly/ || $0 ~ /camelot/)
    print "found"
```

and

```
if (/barfly/ || /camelot/)
    print "found"
```

are exactly equivalent.

One rather bizarre consequence of this rule is that the following boolean expression is valid, but does not do what the user probably intended:

```
# note that /foo/ is on the left of the ~
if (/foo/ ~ $1) print "found foo"
```

This code is "obviously" testing `$1` for a match against the regexp `/foo/`. But in fact, the expression `/foo/ ~ $1' actually means `($0 ~ /foo/) ~ $1'. In other words, first match the input record against the regexp `/foo/`. The result will be either zero or one, depending upon the success or failure of the match. Then match that result against the first field in the record.

Since it is unlikely that you would ever really wish to make this kind of test, `gawk` will issue a warning when it sees this construct in a program.

Another consequence of this rule is that the assignment statement

```
matches = /foo/
```

will assign either zero or one to the variable `matches`, depending upon the contents of the current input record.

This feature of the language was never well documented until the POSIX specification.

Constant regular expressions are also used as the first argument for the `gensub`, `sub` and `gsub` functions, and as the second argument of the `match` function (see section Built-in Functions for String Manipulation). Modern implementations of `awk`, including `gawk`, allow the third argument of `split` to be a regexp constant, while some older implementations do not (d.c.).

This can lead to confusion when attempting to use regexp constants as arguments to user defined functions (see section User-defined Functions). For example:

```
function mysub(pat, repl, str, global)
{
    if (global)
        gsub(pat, repl, str)
    else
        sub(pat, repl, str)
    return str
}

{
    ...
    text = "hi! hi yourself!"
    mysub(/hi/, "howdy", text, 1)
    ...
}
```

In this example, the programmer wishes to pass a regexp constant to the user-defined function `mysub`, which will in turn pass it on to either `sub` or `gsub`. However, what really happens is that the `pat` parameter will be either one or zero, depending upon whether or not `$0` matches `/hi/`.

As it is unlikely that you would ever really wish to pass a truth value in this way, `gawk` will issue a warning when it sees a regexp constant used as a parameter to a user-defined function.

Variables

Variables are ways of storing values at one point in your program for use later in another part of your program. You can manipulate them entirely within your program text, and you can also assign values to them on the `awk` command line.

Using Variables in a Program

Variables let you give names to values and refer to them later. You have already seen variables in many of the examples. The name of a variable must be a sequence of letters, digits and underscores, but it may not begin with a digit. Case is significant in variable names; `a` and `A` are distinct variables.

A variable name is a valid expression by itself; it represents the variable's current value. Variables are given new values with assignment operators, increment operators and decrement operators. See section Assignment Expressions.

A few variables have special built-in meanings, such as `FS`, the field separator, and `NF`, the number of fields in the current input record. See section Built-in Variables, for a list of them. These built-in variables can be used and assigned just like all other variables, but their values are also used or changed

automatically by `awk`. All built-in variables names are entirely upper-case.

Variables in `awk` can be assigned either numeric or string values. By default, variables are initialized to the empty string, which is zero if converted to a number. There is no need to "initialize" each variable explicitly in `awk`, the way you would in C and in most other traditional languages.

Assigning Variables on the Command Line

You can set any `awk` variable by including a variable assignment among the arguments on the command line when you invoke `awk` (see section Other Command Line Arguments). Such an assignment has this form:

```
variable=text
```

With it, you can set a variable either at the beginning of the `awk` run or in between input files.

If you precede the assignment with the `-v' option, like this:

```
-v variable=text
```

then the variable is set at the very beginning, before even the `BEGIN` rules are run. The `-v' option and its assignment must precede all the file name arguments, as well as the program text. (See section Command Line Options, for more information about the `-v' option.)

Otherwise, the variable assignment is performed at a time determined by its position among the input file arguments: after the processing of the preceding input file argument. For example:

```
awk '{ print $n }' n=4 inventory-shipped n=2 BBS-list
```

prints the value of field number `n` for all input records. Before the first file is read, the command line sets the variable `n` equal to four. This causes the fourth field to be printed in lines from the file `` `inventory-shipped' ``. After the first file has finished, but before the second file is started, `n` is set to two, so that the second field is printed in lines from `` `BBS-list' ``.

```
$ awk '{ print $n }' n=4 inventory-shipped n=2 BBS-list
-| 15
-| 24
...
-| 555-5553
-| 555-3412
...
```

Command line arguments are made available for explicit examination by the `awk` program in an array named `ARGV` (see section Using ARGC and ARGV).

`awk` processes the values of command line assignments for escape sequences (d.c.) (see section Escape Sequences).

Conversion of Strings and Numbers

Strings are converted to numbers, and numbers to strings, if the context of the `awk` program demands it. For example, if the value of either `foo` or `bar` in the expression `foo + bar' happens to be a string, it is converted to a number before the addition is performed. If numeric values appear in string concatenation, they are converted to strings. Consider this:

```
two = 2; three = 3
print (two three) + 4
```

This prints the (numeric) value 27. The numeric values of the variables `two` and `three` are converted to strings and concatenated together, and the resulting string is converted back to the number 23, to which four is then added.

If, for some reason, you need to force a number to be converted to a string, concatenate the empty string, `""`, with that number. To force a string to be converted to a number, add zero to that string.

A string is converted to a number by interpreting any numeric prefix of the string as numerals: `"2.5"` converts to 2.5, `"1e3"` converts to 1000, and `"25fix"` has a numeric value of 25. Strings that can't be interpreted as valid numbers are converted to zero.

The exact manner in which numbers are converted into strings is controlled by the `awk` built-in variable `CONVFMT` (see section Built-in Variables). Numbers are converted using the `sprintf` function (see section Built-in Functions for String Manipulation) with `CONVFMT` as the format specifier.

`CONVFMT`'s default value is `"%.6g"`, which prints a value with at least six significant digits. For some applications you will want to change it to specify more precision. Double precision on most modern machines gives you 16 or 17 decimal digits of precision.

Strange results can happen if you set `CONVFMT` to a string that doesn't tell `sprintf` how to format floating point numbers in a useful way. For example, if you forget the `%' in the format, all numbers will be converted to the same constant string.

As a special case, if a number is an integer, then the result of converting it to a string is *always* an integer, no matter what the value of `CONVFMT` may be. Given the following code fragment:

```
CONVFMT = "%2.2f"
a = 12
b = a ""
```

`b` has the value `"12"`, not `"12.00"` (d.c.).

Prior to the POSIX standard, `awk` specified that the value of `OFMT` was used for converting numbers to strings. `OFMT` specifies the output format to use when printing numbers with `print`. `CONVFMT` was introduced in order to separate the semantics of conversion from the semantics of printing. Both `CONVFMT` and `OFMT` have the same default value: `"%.6g"`. In the vast majority of cases, old `awk`

programs will not change their behavior. However, this use of `OFMT` is something to keep in mind if you must port your program to other implementations of `awk`; we recommend that instead of changing your programs, you just port `gawk` itself! See section The `print` Statement, for more information on the `print` statement.

Arithmetic Operators

The `awk` language uses the common arithmetic operators when evaluating expressions. All of these arithmetic operators follow normal precedence rules, and work as you would expect them to.

Here is a file `` `grades' `` containing a list of student names and three test scores per student (it's a small class):

```
Pat   100 97 58
Sandy  84 72 93
Chris  72 92 89
```

This programs takes the file `` `grades' ``, and prints the average of the scores.

```
$ awk '{ sum = $2 + $3 + $4 ; avg = sum / 3
>        print $1, avg }' grades
-| Pat 85
-| Sandy 83
-| Chris 84.3333
```

This table lists the arithmetic operators in `awk`, in order from highest precedence to lowest:

`- x`

Negation.

`+ x`

Unary plus. The expression is converted to a number.

`x ^ y`

`x ** y`

Exponentiation: x raised to the y power. `2 ^ 3' has the value eight. The character sequence `**' is equivalent to `^'. (The POSIX standard only specifies the use of `^' for exponentiation.)

`x * y`

Multiplication.

`x / y`

Division. Since all numbers in `awk` are real numbers, the result is not rounded to an integer: `3 / 4' has the value 0.75.

`x % y`

Remainder. The quotient is rounded toward zero to an integer, multiplied by y and this result is subtracted from x. This operation is sometimes known as "trunc-mod." The following relation

always holds:

```
b * int(a / b) + (a % b) == a
```

One possibly undesirable effect of this definition of remainder is that `x % y` is negative if x is negative. Thus,

```
-17 % 8 = -1
```

In other `awk` implementations, the signedness of the remainder may be machine dependent.

`x + y`

Addition.

`x - y`

Subtraction.

For maximum portability, do not use the `**' operator.

Unary plus and minus have the same precedence, the multiplication operators all have the same precedence, and addition and subtraction have the same precedence.

String Concatenation

There is only one string operation: concatenation. It does not have a specific operator to represent it. Instead, concatenation is performed by writing expressions next to one another, with no operator. For example:

```
$ awk '{ print "Field number one: " $1 }' BBS-list
-| Field number one: aardvark
-| Field number one: alpo-net
...
```

Without the space in the string constant after the `:', the line would run together. For example:

```
$ awk '{ print "Field number one:" $1 }' BBS-list
-| Field number one:aardvark
-| Field number one:alpo-net
...
```

Since string concatenation does not have an explicit operator, it is often necessary to insure that it happens where you want it to by using parentheses to enclose the items to be concatenated. For example, the following code fragment does not concatenate `file` and `name` as you might expect:

```
file = "file"
name = "name"
print "something meaningful" > file name
```

It is necessary to use the following:

```
print "something meaningful" > (file name)
```

We recommend that you use parentheses around concatenation in all but the most common contexts (such as on the right-hand side of `=').

Assignment Expressions

An assignment is an expression that stores a new value into a variable. For example, let's assign the value one to the variable `z`:

```
z = 1
```

After this expression is executed, the variable `z` has the value one. Whatever old value `z` had before the assignment is forgotten.

Assignments can store string values also. For example, this would store the value `"this food is good"` in the variable `message`:

```
thing = "food"
predicate = "good"
message = "this " thing " is " predicate
```

(This also illustrates string concatenation.)

The `=' sign is called an assignment operator. It is the simplest assignment operator because the value of the right-hand operand is stored unchanged.

Most operators (addition, concatenation, and so on) have no effect except to compute a value. If you ignore the value, you might as well not use the operator. An assignment operator is different; it does produce a value, but even if you ignore the value, the assignment still makes itself felt through the alteration of the variable. We call this a side effect.

The left-hand operand of an assignment need not be a variable (see section Variables); it can also be a field (see section Changing the Contents of a Field) or an array element (see section Arrays in `awk`). These are all called lvalues, which means they can appear on the left-hand side of an assignment operator. The right-hand operand may be any expression; it produces the new value which the assignment stores in the specified variable, field or array element. (Such values are called rvalues).

It is important to note that variables do *not* have permanent types. The type of a variable is simply the type of whatever value it happens to hold at the moment. In the following program fragment, the variable `foo` has a numeric value at first, and a string value later on:

```
foo = 1
print foo
```


```
foo = "bar"
print foo
```

When the second assignment gives `foo` a string value, the fact that it previously had a numeric value is forgotten.

String values that do not begin with a digit have a numeric value of zero. After executing this code, the value of `foo` is five:

```
foo = "a string"
foo = foo + 5
```

(Note that using a variable as a number and then later as a string can be confusing and is poor programming style. The above examples illustrate how `awk` works, *not* how you should write your own programs!)

An assignment is an expression, so it has a value: the same value that is assigned. Thus, `z = 1' as an expression has the value one. One consequence of this is that you can write multiple assignments together:

```
x = y = z = 0
```

stores the value zero in all three variables. It does this because the value of `z = 0', which is zero, is stored into `y`, and then the value of `y = z = 0', which is zero, is stored into `x`.

You can use an assignment anywhere an expression is called for. For example, it is valid to write `x != (y = 1)' to set `y` to one and then test whether `x` equals one. But this style tends to make programs hard to read; except in a one-shot program, you should not use such nesting of assignments.

Aside from `=', there are several other assignment operators that do arithmetic with the old value of the variable. For example, the operator `+=' computes a new value by adding the right-hand value to the old value of the variable. Thus, the following assignment adds five to the value of `foo`:

```
foo += 5
```

This is equivalent to the following:

```
foo = foo + 5
```

Use whichever one makes the meaning of your program clearer.

There are situations where using `+=' (or any assignment operator) is *not* the same as simply repeating the left-hand operand in the right-hand expression. For example:

```
# Thanks to Pat Rankin for this example
BEGIN  {
    foo[rand()] += 5
    for (x in foo)
```


```
        print x, foo[x]

    bar[rand()] = bar[rand()] + 5
    for (x in bar)
        print x, bar[x]
}
```

The indices of `bar` are guaranteed to be different, because `rand` will return different values each time it is called. (Arrays and the `rand` function haven't been covered yet. See section Arrays in `awk`, and see section Numeric Built-in Functions, for more information). This example illustrates an important fact about the assignment operators: the left-hand expression is only evaluated *once*.

It is also up to the implementation as to which expression is evaluated first, the left-hand one or the right-hand one. Consider this example:

```
i = 1
a[i += 2] = i + 1
```

The value of `a[3]` could be either two or four.

Here is a table of the arithmetic assignment operators. In each case, the right-hand operand is an expression whose value is converted to a number.

`lvalue += increment`
: Adds increment to the value of lvalue to make the new value of lvalue.

`lvalue -= decrement`
: Subtracts decrement from the value of lvalue.

`lvalue *= coefficient`
: Multiplies the value of lvalue by coefficient.

`lvalue /= divisor`
: Divides the value of lvalue by divisor.

`lvalue %= modulus`
: Sets lvalue to its remainder by modulus.

`lvalue ^= power`
`lvalue **= power`
: Raises lvalue to the power power. (Only the `^=' operator is specified by POSIX.)

For maximum portability, do not use the `**=' operator.

Increment and Decrement Operators

Increment and decrement operators increase or decrease the value of a variable by one. You could do the same thing with an assignment operator, so the increment operators add no power to the `awk` language; but they are convenient abbreviations for very common operations.

The operator to add one is written `++'. It can be used to increment a variable either before or after ta its value.

To pre-increment a variable v, write `++v'. This adds one to the value of v and that new value is also the value of this expression. The assignment expression `v += 1' is completely equivalent.

Writing the `++' after the variable specifies post-increment. This increments the variable value just the same; the difference is that the value of the increment expression itself is the variable's *old* value. Thus, if `foo` has the value four, then the expression `foo++' has the value four, but it changes the value of `foo` to five.

The post-increment `foo++' is nearly equivalent to writing `(foo += 1) - 1'. It is not perfectly equivalent because all numbers in `awk` are floating point: in floating point, `foo + 1 - 1' does not necessarily equal `foo`. But the difference is minute as long as you stick to numbers that are fairly small (less than 10e12).

Any lvalue can be incremented. Fields and array elements are incremented just like variables. (Use `$(i++)' when you wish to do a field reference and a variable increment at the same time. The parentheses are necessary because of the precedence of the field reference operator, `$'.)

The decrement operator `--' works just like `++' except that it subtracts one instead of adding. Like `++', it can be used before the lvalue to pre-decrement or after it to post-decrement.

Here is a summary of increment and decrement expressions.

`++lvalue`
: This expression increments lvalue and the new value becomes the value of the expression.

`lvalue++`
: This expression increments lvalue, but the value of the expression is the *old* value of lvalue.

`--lvalue`
: Like `++lvalue', but instead of adding, it subtracts. It decrements lvalue and delivers the value that results.

`lvalue--`
: Like `lvalue++', but instead of adding, it subtracts. It decrements lvalue. The value of the expression is the *old* value of lvalue.

True and False in awk

Many programming languages have a special representation for the concepts of "true" and "false." Such languages usually use the special constants `true` and `false`, or perhaps their upper-case equivalents.

`awk` is different. It borrows a very simple concept of true and false from C. In `awk`, any non-zero numeric value, *or* any non-empty string value is true. Any other value (zero or the null string, `""`) is false. The following program will print `A strange truth value' three times:

```
BEGIN {
   if (3.1415927)
```


```
        print "A strange truth value"
    if ("Four Score And Seven Years Ago")
        print "A strange truth value"
    if (j = 57)
        print "A strange truth value"
}
```

There is a surprising consequence of the "non-zero or non-null" rule: The string constant `"0"` is actually true, since it is non-null (d.c.).

Variable Typing and Comparison Expressions

Unlike other programming languages, `awk` variables do not have a fixed type. Instead, they can be either a number or a string, depending upon the value that is assigned to them.

The 1992 POSIX standard introduced the concept of a numeric string, which is simply a string that looks like a number, for example, `" +2"`. This concept is used for determining the type of a variable.

The type of the variable is important, since the types of two variables determine how they are compared.

In `gawk`, variable typing follows these rules.

1. A numeric literal or the result of a numeric operation has the numeric attribute.
2. A string literal or the result of a string operation has the string attribute.
3. Fields, `getline` input, `FILENAME`, `ARGV` elements, `ENVIRON` elements and the elements of an array created by `split` that are numeric strings have the strnum attribute. Otherwise, they have the string attribute. Uninitialized variables also have the strnum attribute.
4. Attributes propagate across assignments, but are not changed by any use.

The last rule is particularly important. In the following program, `a` has numeric type, even though it is later used in a string operation.

```
BEGIN {
        a = 12.345
        b = a " is a cute number"
        print b
}
```

When two operands are compared, either string comparison or numeric comparison may be used, depending on the attributes of the operands, according to the following, symmetric, matrix:

The basic idea is that user input that looks numeric, and *only* user input, should be treated as numeric, even though it is actually made of characters, and is therefore also a string.

Comparison expressions compare strings or numbers for relationships such as equality. They are written using relational operators, which are a superset of those in C. Here is a table of them:

`x < y`

True if x is less than y.

`x <= y`

True if x is less than or equal to y.

`x > y`

True if x is greater than y.

`x >= y`

True if x is greater than or equal to y.

`x == y`

True if x is equal to y.

`x != y`

True if x is not equal to y.

`x ~ y`

True if the string x matches the regexp denoted by y.

`x !~ y`

True if the string x does not match the regexp denoted by y.

`subscript in array`

True if the array array has an element with the subscript subscript.

Comparison expressions have the value one if true and zero if false.

When comparing operands of mixed types, numeric operands are converted to strings using the value of `CONVFMT` (see section Conversion of Strings and Numbers).

Strings are compared by comparing the first character of each, then the second character of each, and so on. Thus `"10"` is less than `"9"`. If there are two strings where one is a prefix of the other, the shorter string is less than the longer one. Thus `"abc"` is less than `"abcd"`.

It is very easy to accidentally mistype the `==' operator, and leave off one of the `='s. The result is still valid `awk` code, but the program will not do what you mean:

```
if (a = b)   # oops! should be a == b
   ...
else
   ...
```

Unless `b` happens to be zero or the null string, the `if` part of the test will always succeed. Because the operators are so similar, this kind of error is very difficult to spot when scanning the source code.

Here are some sample expressions, how `gawk` compares them, and what the result of the comparison is.

`1.5 <= 2.0`

numeric comparison (true)

`"abc" >= "xyz"`

string comparison (false)

```
1.5 != " +2"
```

string comparison (true)

```
"1e2" < "3"
```

string comparison (true)

```
a = 2; b = "2"
a == b
```

string comparison (true)

```
a = 2; b = " +2"
a == b
```

string comparison (false)

In this example,

```
$ echo 1e2 3 | awk '{ print ($1 < $2) ? "true" : "false" }'
-| false
```

the result is `false' since both `$1` and `$2` are numeric strings and thus both have the strnum attribute, dictating a numeric comparison.

The purpose of the comparison rules and the use of numeric strings is to attempt to produce the behavior that is "least surprising," while still "doing the right thing."

String comparisons and regular expression comparisons are very different. For example,

```
x == "foo"
```

has the value of one, or is true, if the variable `x` is precisely `foo'. By contrast,

```
x ~ /foo/
```

has the value one if `x` contains `foo', such as `"Oh, what a fool am I!"`.

The right hand operand of the `~' and `!~' operators may be either a regexp constant (`/.../`), or an ordinary expression, in which case the value of the expression as a string is used as a dynamic regexp (see section How to Use Regular Expressions; also see section Using Dynamic Regexps).

In recent implementations of `awk`, a constant regular expression in slashes by itself is also an expression. The regexp `/regexp/` is an abbreviation for this comparison expression:

```
$0 ~ /regexp/
```

One special place where `/foo/` is *not* an abbreviation for `$0 ~ /foo/' is when it is the right-hand operand of `~' or `!~'! See section Using Regular Expression Constants, where this is discussed in more detail.

Boolean Expressions

A boolean expression is a combination of comparison expressions or matching expressions, using the boolean operators "or" (`||'), "and" (`&&'), and "not" (`!'), along with parentheses to control nesting. The truth value of the boolean expression is computed by combining the truth values of the component expressions. Boolean expressions are also referred to as logical expressions. The terms are equivalent.

Boolean expressions can be used wherever comparison and matching expressions can be used. They can be used in `if`, `while`, `do` and `for` statements (see section Control Statements in Actions). They have numeric values (one if true, zero if false), which come into play if the result of the boolean expression is stored in a variable, or used in arithmetic.

In addition, every boolean expression is also a valid pattern, so you can use one as a pattern to control the execution of rules.

Here are descriptions of the three boolean operators, with examples.

`boolean1 && boolean2`

True if both boolean1 and boolean2 are true. For example, the following statement prints the current input record if it contains both `2400' and `foo'.

```
if ($0 ~ /2400/ && $0 ~ /foo/) print
```

The subexpression boolean2 is evaluated only if boolean1 is true. This can make a difference when boolean2 contains expressions that have side effects: in the case of `$0 ~ /foo/ && ($2 == bar++)', the variable `bar` is not incremented if there is no `foo' in the record.

`boolean1 || boolean2`

True if at least one of boolean1 or boolean2 is true. For example, the following statement prints all records in the input that contain *either* `2400' or `foo', or both.

```
if ($0 ~ /2400/ || $0 ~ /foo/) print
```

The subexpression boolean2 is evaluated only if boolean1 is false. This can make a difference when boolean2 contains expressions that have side effects.

`! boolean`

True if boolean is false. For example, the following program prints all records in the input file `` `BBS-list' `` that do *not* contain the string `foo'.

```
awk '{ if (! ($0 ~ /foo/)) print }' BBS-list
```

The `&&' and `||' operators are called short-circuit operators because of the way they work. Evaluation of the full expression is "short-circuited" if the result can be determined part way through its evaluation.

You can continue a statement that uses `&&' or `||' simply by putting a newline after them. But you cannot put a newline in front of either of these operators without using backslash continuation (see

section awk Statements Versus Lines).

The actual value of an expression using the `!' operator will be either one or zero, depending upon the truth value of the expression it is applied to.

The `!' operator is often useful for changing the sense of a flag variable from false to true and back again. For example, the following program is one way to print lines in between special bracketing lines:

```
$1 == "START"   { interested = ! interested }
interested == 1 { print }
$1 == "END"     { interested = ! interested }
```

The variable `interested`, like all `awk` variables, starts out initialized to zero, which is also false. When a line is seen whose first field is `START', the value of `interested` is toggled to true, using `!'. The next rule prints lines as long as `interested` is true. When a line is seen whose first field is `END', `interested` is toggled back to false.

Conditional Expressions

A conditional expression is a special kind of expression with three operands. It allows you to use one expression's value to select one of two other expressions.

The conditional expression is the same as in the C language:

```
selector ? if-true-exp : if-false-exp
```

There are three subexpressions. The first, selector, is always computed first. If it is "true" (not zero and not null) then if-true-exp is computed next and its value becomes the value of the whole expression. Otherwise, if-false-exp is computed next and its value becomes the value of the whole expression.

For example, this expression produces the absolute value of `x`:

```
x > 0 ? x : -x
```

Each time the conditional expression is computed, exactly one of if-true-exp and if-false-exp is computed; the other is ignored. This is important when the expressions contain side effects. For example, this conditional expression examines element `i` of either array `a` or array `b`, and increments `i`.

```
x == y ? a[i++] : b[i++]
```

This is guaranteed to increment `i` exactly once, because each time only one of the two increment expressions is executed, and the other is not. See section Arrays in awk, for more information about arrays.

As a minor `gawk` extension, you can continue a statement that uses `?:' simply by putting a newline after either character. However, you cannot put a newline in front of either character without using backslash continuation (see section awk Statements Versus Lines).

Function Calls

A function is a name for a particular calculation. Because it has a name, you can ask for it by name at any point in the program. For example, the function `sqrt` computes the square root of a number.

A fixed set of functions are built-in, which means they are available in every `awk` program. The `sqrt` function is one of these. See section Built-in Functions, for a list of built-in functions and their descriptions. In addition, you can define your own functions for use in your program. See section User-defined Functions, for how to do this.

The way to use a function is with a function call expression, which consists of the function name followed immediately by a list of arguments in parentheses. The arguments are expressions which provide the raw materials for the function's calculations. When there is more than one argument, they are separated by commas. If there are no arguments, write just `()' after the function name. Here are some examples:

```
sqrt(x^2 + y^2)         one argument
atan2(y, x)             two arguments
rand()                  no arguments
```

Do not put any space between the function name and the open-parenthesis! A user-defined function name looks just like the name of a variable, and space would make the expression look like concatenation of a variable with an expression inside parentheses. Space before the parenthesis is harmless with built-in functions, but it is best not to get into the habit of using space to avoid mistakes with user-defined functions.

Each function expects a particular number of arguments. For example, the `sqrt` function must be called with a single argument, the number to take the square root of:

```
sqrt(argument)
```

Some of the built-in functions allow you to omit the final argument. If you do so, they use a reasonable default. See section Built-in Functions, for full details. If arguments are omitted in calls to user-defined functions, then those arguments are treated as local variables, initialized to the empty string (see section User-defined Functions).

Like every other expression, the function call has a value, which is computed by the function based on the arguments you give it. In this example, the value of `sqrt(argument)' is the square root of argument. A function can also have side effects, such as assigning values to certain variables or doing I/O.

Here is a command to read numbers, one number per line, and print the square root of each one:

```
$ awk '{ print "The square root of", $1, "is", sqrt($1) }'
1
-| The square root of 1 is 1
```


```
3
-| The square root of 3 is 1.73205
5
-| The square root of 5 is 2.23607
Control-d
```

Operator Precedence (How Operators Nest)

Operator precedence determines how operators are grouped, when different operators appear close by in one expression. For example, `*' has higher precedence than `+'; thus, `a + b * c' means to multiply `b' and `c', and then add `a' to the product (i.e. `a + (b * c)').

You can overrule the precedence of the operators by using parentheses. You can think of the precedence rules as saying where the parentheses are assumed to be if you do not write parentheses yourself. In fact, it is wise to always use parentheses whenever you have an unusual combination of operators, because other people who read the program may not remember what the precedence is in this case. You might forget, too; then you could make a mistake. Explicit parentheses will help prevent any such mistake.

When operators of equal precedence are used together, the leftmost operator groups first, except for the assignment, conditional and exponentiation operators, which group in the opposite order. Thus, `a - b + c' groups as `(a - b) + c', and `a = b = c' groups as `a = (b = c)'.

The precedence of prefix unary operators does not matter as long as only unary operators are involved, because there is only one way to interpret them--innermost first. Thus, `$++i' means `$(++i)' and `++$x' means `++($x)'. However, when another operator follows the operand, then the precedence of the unary operators can matter. Thus, `$x^2' means `($x)^2', but `-x^2' means `-(x^2)', because `-' has lower precedence than `^' while `$' has higher precedence.

Here is a table of `awk''s operators, in order from highest precedence to lowest:

`(...)`
: Grouping.

`$`
: Field.

`++ --`
: Increment, decrement.

`^ **`
: Exponentiation. These operators group right-to-left. (The `**' operator is not specified by POSIX.)

`+ - !`
: Unary plus, minus, logical "not".

`* / %`
: Multiplication, division, modulus.

`+ -`
: Addition, subtraction.

`Concatenation`

No special token is used to indicate concatenation. The operands are simply written side by side.

`< <= == !=`

`> >= >> |`

Relational, and redirection. The relational operators and the redirections have the same precedence level. Characters such as `>' serve both as relationals and as redirections; the context distinguishes between the two meanings.

Note that the I/O redirection operators in `print` and `printf` statements belong to the statement level, not to expressions. The redirection does not produce an expression which could be the operand of another operator. As a result, it does not make sense to use a redirection operator near another operator of lower precedence, without parentheses. Such combinations, for example `print foo > a ? b : c', result in syntax errors. The correct way to write this statement is `print foo > (a ? b : c)'.

`~ !~`

Matching, non-matching.

`in`

Array membership.

`&&`

Logical "and".

`||`

Logical "or".

`?:`

Conditional. This operator groups right-to-left.

`= += -= *=`

`/= %= ^= **=`

Assignment. These operators group right-to-left. (The `**=' operator is not specified by POSIX.)

Go to the previous, next section.

Go to the previous, next section.

Patterns and Actions

As you have already seen, each `awk` statement consists of a pattern with an associated action. This chapter describes how you build patterns and actions.

Pattern Elements

Patterns in `awk` control the execution of rules: a rule is executed when its pattern matches the current input record. This section explains all about how to write patterns.

Kinds of Patterns

Here is a summary of the types of patterns supported in `awk`.

`/regular expression/`

A regular expression as a pattern. It matches when the text of the input record fits the regular expression. (See section Regular Expressions.)

`expression`

A single expression. It matches when its value is non-zero (if a number) or non-null (if a string). (See section Expressions as Patterns.)

`pat1, pat2`

A pair of patterns separated by a comma, specifying a range of records. The range includes both the initial record that matches pat1, and the final record that matches pat2. (See section Specifying Record Ranges with Patterns.)

`BEGIN`

`END`

Special patterns for you to supply start-up or clean-up actions for your `awk` program. (See section The `BEGIN` and `END` Special Patterns.)

`empty`

The empty pattern matches every input record. (See section The Empty Pattern.)

Regular Expressions as Patterns

We have been using regular expressions as patterns since our early examples. This kind of pattern is simply a regexp constant in the pattern part of a rule. Its meaning is `$0 ~ /pattern/'. The pattern matches when the input record matches the regexp. For example:

```
/foo|bar|baz/  { buzzwords++ }
```


```
END             { print buzzwords, "buzzwords seen" }
```

Expressions as Patterns

Any `awk` expression is valid as an `awk` pattern. Then the pattern matches if the expression's value is non-zero (if a number) or non-null (if a string).

The expression is reevaluated each time the rule is tested against a new input record. If the expression uses fields such as `$1`, the value depends directly on the new input record's text; otherwise, it depends only on what has happened so far in the execution of the `awk` program, but that may still be useful.

A very common kind of expression used as a pattern is the comparison expression, using the comparison operators described in section Variable Typing and Comparison Expressions.

Regexp matching and non-matching are also very common expressions. The left operand of the `~' and `!~' operators is a string. The right operand is either a constant regular expression enclosed in slashes (`/regexp/`), or any expression, whose string value is used as a dynamic regular expression (see section Using Dynamic Regexps).

The following example prints the second field of each input record whose first field is precisely `foo'.

```
$ awk '$1 == "foo" { print $2 }' BBS-list
```

(There is no output, since there is no BBS site named "foo".) Contrast this with the following regular expression match, which would accept any record with a first field that contains `foo':

```
$ awk '$1 ~ /foo/ { print $2 }' BBS-list
-| 555-1234
-| 555-6699
-| 555-6480
-| 555-2127
```

Boolean expressions are also commonly used as patterns. Whether the pattern matches an input record depends on whether its subexpressions match.

For example, the following command prints all records in `BBS-list' that contain both `2400' and `foo'.

```
$ awk '/2400/ && /foo/' BBS-list
-| fooey         555-1234     2400/1200/300     B
```

The following command prints all records in `BBS-list' that contain *either* `2400' or `foo', or both.

```
$ awk '/2400/ || /foo/' BBS-list
-| alpo-net     555-3412     2400/1200/300     A
-| bites         555-1675     2400/1200/300     A
-| fooey         555-1234     2400/1200/300     B
```


```
-| foot          555-6699     1200/300          B
-| macfoo        555-6480     1200/300          A
-| sdace         555-3430     2400/1200/300     A
-| sabafoo       555-2127     1200/300          C
```

The following command prints all records in `BBS-list' that do *not* contain the string `foo'.

```
$ awk '! /foo/' BBS-list
-| aardvark     555-5553     1200/300          B
-| alpo-net     555-3412     2400/1200/300     A
-| barfly       555-7685     1200/300          A
-| bites        555-1675     2400/1200/300     A
-| camelot      555-0542     300               C
-| core         555-2912     1200/300          C
-| sdace        555-3430     2400/1200/300     A
```

The subexpressions of a boolean operator in a pattern can be constant regular expressions, comparisons, or any other `awk` expressions. Range patterns are not expressions, so they cannot appear inside boolean patterns. Likewise, the special patterns `BEGIN` and `END`, which never match any input record, are not expressions and cannot appear inside boolean patterns.

A regexp constant as a pattern is also a special case of an expression pattern. `/foo/` as an expression has the value one if `foo' appears in the current input record; thus, as a pattern, `/foo/` matches any record containing `foo'.

Specifying Record Ranges with Patterns

A range pattern is made of two patterns separated by a comma, of the form `begpat, endpat'. It matches ranges of consecutive input records. The first pattern, begpat, controls where the range begins, and the second one, endpat, controls where it ends. For example,

```
awk '$1 == "on", $1 == "off"'
```

prints every record between `on'/`off' pairs, inclusive.

A range pattern starts out by matching begpat against every input record; when a record matches begpat, the range pattern becomes turned on. The range pattern matches this record. As long as it stays turned on, it automatically matches every input record read. It also matches endpat against every input record; when that succeeds, the range pattern is turned off again for the following record. Then it goes back to checking begpat against each record.

The record that turns on the range pattern and the one that turns it off both match the range pattern. If you don't want to operate on these records, you can write `if` statements in the rule's action to distinguish them from the records you are interested in.

It is possible for a pattern to be turned both on and off by the same record, if the record satisfies both conditions. Then the action is executed for just that record.

For example, suppose you have text between two identical markers (say the `%' symbol) that you wish to ignore. You might try to combine a range pattern that describes the delimited text with the `next' statement (not discussed yet, see section The `next' Statement), which causes `awk' to skip any further processing of the current record and start over again with the next input record. Such a program would like this:

```
/^%$/,/^%$/     { next }
                { print }
```

This program fails because the range pattern is both turned on and turned off by the first line with just a `%' on it. To accomplish this task, you must write the program this way, using a flag:

```
/^%$/      { skip = ! skip; next }
skip == 1  { next } # skip lines with `skip' set
```

Note that in a range pattern, the `,' has the lowest precedence (is evaluated last) of all the operators. Thus, for example, the following program attempts to combine a range pattern with another, simpler test.

```
echo Yes | awk '/1/,/2/ || /Yes/'
```

The author of this program intended it to mean `(/1/,/2/) || /Yes/'. However, `awk' interprets this as `/1/,(/2/ || /Yes/)'. This cannot be changed or worked around; range patterns do not combine with other patterns.

The `BEGIN' and `END' Special Patterns

`BEGIN' and `END' are special patterns. They are not used to match input records. Rather, they supply start-up or clean-up actions for your `awk' script.

Startup and Cleanup Actions

A `BEGIN' rule is executed, once, before the first input record has been read. An `END' rule is executed, once, after all the input has been read. For example:

```
$ awk '
> BEGIN { print "Analysis of \"foo\"" }
> /foo/ { ++n }
> END   { print "\"foo\" appears " n " times." }' BBS-list
-| Analysis of "foo"
-| "foo" appears 4 times.
```

This program finds the number of records in the input file `BBS-list' that contain the string `foo'. The `BEGIN' rule prints a title for the report. There is no need to use the `BEGIN' rule to initialize the counter `n' to zero, as `awk' does this automatically (see section Variables).

The second rule increments the variable `n' every time a record containing the pattern `foo' is read. The

END rule prints the value of n at the end of the run.

The special patterns BEGIN and END cannot be used in ranges or with boolean operators (indeed, they cannot be used with any operators).

An awk program may have multiple BEGIN and/or END rules. They are executed in the order they appear, all the BEGIN rules at start-up and all the END rules at termination. BEGIN and END rules may be intermixed with other rules. This feature was added in the 1987 version of awk, and is included in the POSIX standard. The original (1978) version of awk required you to put the BEGIN rule at the beginning of the program, and the END rule at the end, and only allowed one of each. This is no longer required, but it is a good idea in terms of program organization and readability.

Multiple BEGIN and END rules are useful for writing library functions, since each library file can have its own BEGIN and/or END rule to do its own initialization and/or cleanup. Note that the order in which library functions are named on the command line controls the order in which their BEGIN and END rules are executed. Therefore you have to be careful to write such rules in library files so that the order in which they are executed doesn't matter. See section Command Line Options, for more information on using library functions. See section A Library of awk Functions, for a number of useful library functions.

If an awk program only has a BEGIN rule, and no other rules, then the program exits after the BEGIN rule has been run. (The original version of awk used to keep reading and ignoring input until end of file was seen.) However, if an END rule exists, then the input will be read, even if there are no other rules in the program. This is necessary in case the END rule checks the FNR and NR variables (d.c.).

BEGIN and END rules must have actions; there is no default action for these rules since there is no current record when they run.

Input/Output from BEGIN and END Rules

There are several (sometimes subtle) issues involved when doing I/O from a BEGIN or END rule.

The first has to do with the value of $0 in a BEGIN rule. Since BEGIN rules are executed before any input is read, there simply is no input record, and therefore no fields, when executing BEGIN rules. References to $0 and the fields yield a null string or zero, depending upon the context. One way to give $0 a real value is to execute a getline command without a variable (see section Explicit Input with getline). Another way is to simply assign a value to it.

The second point is similar to the first, but from the other direction. Inside an END rule, what is the value of $0 and NF? Traditionally, due largely to implementation issues, $0 and NF were *undefined* inside an END rule. The POSIX standard specified that NF was available in an END rule, containing the number of fields from the last input record. Due most probably to an oversight, the standard does not say that $0 is also preserved, although logically one would think that it should be. In fact, gawk does preserve the value of $0 for use in END rules. Be aware, however, that Unix awk, and possibly other implementations, do not.

The third point follows from the first two. What is the meaning of `print' inside a BEGIN or END rule? The meaning is the same as always, `print $0'. If $0 is the null string, then this prints an empty line. Many long time awk programmers use `print' in BEGIN and END rules, to mean `print ""', relying on $0

being null. While you might generally get away with this in `BEGIN` rules, in `gawk` at least, it is a very bad idea in `END` rules. It is also poor style, since if you want an empty line in the output, you should say so explicitly in your program.

The Empty Pattern

An empty (i.e. non-existent) pattern is considered to match *every* input record. For example, the program:

```
awk '{ print $1 }' BBS-list
```

prints the first field of every record.

Overview of Actions

An `awk` program or script consists of a series of rules and function definitions, interspersed. (Functions are described later. See section User-defined Functions.)

A rule contains a pattern and an action, either of which (but not both) may be omitted. The purpose of the action is to tell `awk` what to do once a match for the pattern is found. Thus, in outline, an `awk` program generally looks like this:

```
[pattern] [{ action }]
[pattern] [{ action }]
...
function name(args) { ... }
...
```

An action consists of one or more `awk` statements, enclosed in curly braces (`{' and `}'). Each statement specifies one thing to be done. The statements are separated by newlines or semicolons.

The curly braces around an action must be used even if the action contains only one statement, or even if it contains no statements at all. However, if you omit the action entirely, omit the curly braces as well. An omitted action is equivalent to `{ print $0 }'.

```
/foo/  { }  # match foo, do nothing - empty action
/foo/       # match foo, print the record - omitted action
```

Here are the kinds of statements supported in `awk`:

- Expressions, which can call functions or assign values to variables (see section Expressions). Executing this kind of statement simply computes the value of the expression. This is useful when the expression has side effects (see section Assignment Expressions).
- Control statements, which specify the control flow of `awk` programs. The `awk` language gives you C-like constructs (`if`, `for`, `while`, and `do`) as well as a few special ones (see section Control Statements in Actions).

- Compound statements, which consist of one or more statements enclosed in curly braces. A compound statement is used in order to put several statements together in the body of an `if`, `while`, `do` or `for` statement.
- Input statements, using the `getline` command (see section Explicit Input with `getline`), the `next` statement (see section The `next` Statement), and the `nextfile` statement (see section The `nextfile` Statement).
- Output statements, `print` and `printf`. See section Printing Output.
- Deletion statements, for deleting array elements. See section The `delete` Statement.

The next chapter covers control statements in detail.

Go to the previous, next section.

Go to the previous, next section.

Control Statements in Actions

Control statements such as `if`, `while`, and so on control the flow of execution in `awk` programs. Most of the control statements in `awk` are patterned on similar statements in C.

All the control statements start with special keywords such as `if` and `while`, to distinguish them from simple expressions.

Many control statements contain other statements; for example, the `if` statement contains another statement which may or may not be executed. The contained statement is called the body. If you want to include more than one statement in the body, group them into a single compound statement with curly braces, separating them with newlines or semicolons.

The `if-else` Statement

The `if-else` statement is `awk`'s decision-making statement. It looks like this:

```
if (condition) then-body [else else-body]
```

The condition is an expression that controls what the rest of the statement will do. If condition is true, then-body is executed; otherwise, else-body is executed. The `else` part of the statement is optional. The condition is considered false if its value is zero or the null string, and true otherwise.

Here is an example:

```
if (x % 2 == 0)
    print "x is even"
else
    print "x is odd"
```

In this example, if the expression `x % 2 == 0' is true (that is, the value of `x` is evenly divisible by two), then the first `print` statement is executed, otherwise the second `print` statement is executed.

If the `else` appears on the same line as then-body, and then-body is not a compound statement (i.e. not surrounded by curly braces), then a semicolon must separate then-body from `else`. To illustrate this, let's rewrite the previous example:

```
if (x % 2 == 0) print "x is even"; else
        print "x is odd"
```

If you forget the `;', `awk` won't be able to interpret the statement, and you will get a syntax error.

We would not actually write this example this way, because a human reader might fail to see the `else` if

it were not the first thing on its line.

The `while` Statement

In programming, a loop means a part of a program that can be executed two or more times in succession.

The `while` statement is the simplest looping statement in `awk`. It repeatedly executes a statement as long as a condition is true. It looks like this:

```
while (condition)
  body
```

Here body is a statement that we call the body of the loop, and condition is an expression that controls how long the loop keeps running.

The first thing the `while` statement does is test condition. If condition is true, it executes the statement body. After body has been executed, condition is tested again, and if it is still true, body is executed again. This process repeats until condition is no longer true. If condition is initially false, the body of the loop is never executed, and `awk` continues with the statement following the loop.

This example prints the first three fields of each record, one per line.

```
awk '{ i = 1
       while (i <= 3) {
           print $i
           i++
       }
}' inventory-shipped
```

Here the body of the loop is a compound statement enclosed in braces, containing two statements.

The loop works like this: first, the value of `i` is set to one. Then, the `while` tests whether `i` is less than or equal to three. This is true when `i` equals one, so the `i`-th field is printed. Then the `i++' increments the value of `i` and the loop repeats. The loop terminates when `i` reaches four.

As you can see, a newline is not required between the condition and the body; but using one makes the program clearer unless the body is a compound statement or is very simple. The newline after the open-brace that begins the compound statement is not required either, but the program would be harder to read without it.

The `do-while` Statement

The `do` loop is a variation of the `while` looping statement. The `do` loop executes the body once, and then repeats body as long as condition is true. It looks like this:

```
do
```


```
  body
while (condition)
```

Even if condition is false at the start, body is executed at least once (and only once, unless executing body makes condition true). Contrast this with the corresponding `while` statement:

```
while (condition)
  body
```

This statement does not execute body even once if condition is false to begin with.

Here is an example of a `do` statement:

```
awk '{ i = 1
       do {
          print $0
          i++
       } while (i <= 10)
}'
```

This program prints each input record ten times. It isn't a very realistic example, since in this case an ordinary `while` would do just as well. But this reflects actual experience; there is only occasionally a real use for a `do` statement.

The `for` Statement

The `for` statement makes it more convenient to count iterations of a loop. The general form of the `for` statement looks like this:

```
for (initialization; condition; increment)
  body
```

The initialization, condition and increment parts are arbitrary `awk` expressions, and body stands for any `awk` statement.

The `for` statement starts by executing initialization. Then, as long as condition is true, it repeatedly executes body and then increment. Typically initialization sets a variable to either zero or one, increment adds one to it, and condition compares it against the desired number of iterations.

Here is an example of a `for` statement:

```
awk '{ for (i = 1; i <= 3; i++)
          print $i
}' inventory-shipped
```

This prints the first three fields of each input record, one field per line.

You cannot set more than one variable in the initialization part unless you use a multiple assignment statement such as `x = y = 0', which is possible only if all the initial values are equal. (But you can initialize additional variables by writing their assignments as separate statements preceding the `for` loop.)

The same is true of the increment part; to increment additional variables, you must write separate statements at the end of the loop. The C compound expression, using C's comma operator, would be useful in this context, but it is not supported in `awk`.

Most often, increment is an increment expression, as in the example above. But this is not required; it can be any expression whatever. For example, this statement prints all the powers of two between one and 100:

```
for (i = 1; i <= 100; i *= 2)
  print i
```

Any of the three expressions in the parentheses following the `for` may be omitted if there is nothing to be done there. Thus, `for (; x > 0;)' is equivalent to `while (x > 0)'. If the condition is omitted, it is treated as true, effectively yielding an infinite loop (i.e. a loop that will never terminate).

In most cases, a `for` loop is an abbreviation for a `while` loop, as shown here:

```
initialization
while (condition) {
  body
  increment
}
```

The only exception is when the `continue` statement (see section The `continue` Statement) is used inside the loop; changing a `for` statement to a `while` statement in this way can change the effect of the `continue` statement inside the loop.

There is an alternate version of the `for` loop, for iterating over all the indices of an array:

```
for (i in array)
    do something with array[i]
```

See section Scanning All Elements of an Array, for more information on this version of the `for` loop.

The `awk` language has a `for` statement in addition to a `while` statement because often a `for` loop is both less work to type and more natural to think of. Counting the number of iterations is very common in loops. It can be easier to think of this counting as part of looping rather than as something to do inside the loop.

The next section has more complicated examples of `for` loops.

The `break` Statement

The `break` statement jumps out of the innermost `for`, `while`, or `do` loop that encloses it. The following example finds the smallest divisor of any integer, and also identifies prime numbers:

```
awk '# find smallest divisor of num
     { num = $1
       for (div = 2; div*div <= num; div++)
         if (num % div == 0)
           break
       if (num % div == 0)
         printf "Smallest divisor of %d is %d\n", num, div
       else
         printf "%d is prime\n", num
     }'
```

When the remainder is zero in the first `if` statement, `awk` immediately breaks out of the containing `for` loop. This means that `awk` proceeds immediately to the statement following the loop and continues processing. (This is very different from the `exit` statement which stops the entire `awk` program. See section The `exit` Statement.)

Here is another program equivalent to the previous one. It illustrates how the condition of a `for` or `while` could just as well be replaced with a `break` inside an `if`:

```
awk '# find smallest divisor of num
     { num = $1
       for (div = 2; ; div++) {
         if (num % div == 0) {
           printf "Smallest divisor of %d is %d\n", num, div
           break
         }
         if (div*div > num) {
           printf "%d is prime\n", num
           break
         }
       }
}'
```

As described above, the `break` statement has no meaning when used outside the body of a loop. However, although it was never documented, historical implementations of `awk` have treated the `break` statement outside of a loop as if it were a `next` statement (see section The `next` Statement). Recent versions of Unix `awk` no longer allow this usage. `gawk` will support this use of `break` only if `--traditional' has been specified on the command line (see section Command Line Options). Otherwise, it will be treated as an error, since the POSIX standard specifies that `break` should only be used inside the body of a loop (d.c.).

The `continue` Statement

The `continue` statement, like `break`, is used only inside `for`, `while`, and `do` loops. It skips over the rest of the loop body, causing the next cycle around the loop to begin immediately. Contrast this with `break`, which jumps out of the loop altogether.

The `continue` statement in a `for` loop directs `awk` to skip the rest of the body of the loop, and resume execution with the increment-expression of the `for` statement. The following program illustrates this fact:

```
awk 'BEGIN {
     for (x = 0; x <= 20; x++) {
         if (x == 5)
             continue
         printf "%d ", x
     }
     print ""
}'
```

This program prints all the numbers from zero to 20, except for five, for which the `printf` is skipped. Since the increment `x++' is not skipped, `x` does not remain stuck at five. Contrast the `for` loop above with this `while` loop:

```
awk 'BEGIN {
     x = 0
     while (x <= 20) {
         if (x == 5)
             continue
         printf "%d ", x
         x++
     }
     print ""
}'
```

This program loops forever once `x` gets to five.

As described above, the `continue` statement has no meaning when used outside the body of a loop. However, although it was never documented, historical implementations of `awk` have treated the `continue` statement outside of a loop as if it were a `next` statement (see section The `next` Statement). Recent versions of Unix `awk` no longer allow this usage. `gawk` will support this use of `continue` only if `--traditional' has been specified on the command line (see section Command Line Options). Otherwise, it will be treated as an error, since the POSIX standard specifies that `continue` should only be used inside the body of a loop (d.c.).

The `next` Statement

The `next` statement forces `awk` to immediately stop processing the current record and go on to the next record. This means that no further rules are executed for the current record. The rest of the current rule's action is not executed either.

Contrast this with the effect of the `getline` function (see section Explicit Input with `getline`). That too causes `awk` to read the next record immediately, but it does not alter the flow of control in any way. So the rest of the current action executes with a new input record.

At the highest level, `awk` program execution is a loop that reads an input record and then tests each rule's pattern against it. If you think of this loop as a `for` statement whose body contains the rules, then the `next` statement is analogous to a `continue` statement: it skips to the end of the body of this implicit loop, and executes the increment (which reads another record).

For example, if your `awk` program works only on records with four fields, and you don't want it to fail when given bad input, you might use this rule near the beginning of the program:

```
NF != 4 {
  err = sprintf("%s:%d: skipped: NF != 4\n", FILENAME, FNR)
  print err > "/dev/stderr"
  next
}
```

so that the following rules will not see the bad record. The error message is redirected to the standard error output stream, as error messages should be. See section Special File Names in `gawk`.

According to the POSIX standard, the behavior is undefined if the `next` statement is used in a `BEGIN` or `END` rule. `gawk` will treat it as a syntax error. Although POSIX permits it, some other `awk` implementations don't allow the `next` statement inside function bodies (see section User-defined Functions). Just as any other `next` statement, a `next` inside a function body reads the next record and starts processing it with the first rule in the program.

If the `next` statement causes the end of the input to be reached, then the code in any `END` rules will be executed. See section The `BEGIN` and `END` Special Patterns.

The `nextfile` Statement

`gawk` provides the `nextfile` statement, which is similar to the `next` statement. However, instead of abandoning processing of the current record, the `nextfile` statement instructs `gawk` to stop processing the current data file.

Upon execution of the `nextfile` statement, `FILENAME` is updated to the name of the next data file listed on the command line, `FNR` is reset to one, `ARGIND` is incremented, and processing starts over with the first rule in the progam. See section Built-in Variables.

If the `nextfile` statement causes the end of the input to be reached, then the code in any `END` rules will be executed. See section The `BEGIN` and `END` Special Patterns.

The `nextfile` statement is a `gawk` extension; it is not (currently) available in any other `awk` implementation. See section Implementing `nextfile` as a Function, for a user-defined function you can use to simulate the `nextfile` statement.

The `nextfile` statement would be useful if you have many data files to process, and you expect that you would not want to process every record in every file. Normally, in order to move on to the next data file, you would have to continue scanning the unwanted records. The `nextfile` statement accomplishes this much more efficiently.

Caution: Versions of `gawk` prior to 3.0 used two words (`next file') for the `nextfile` statement. This was changed in 3.0 to one word, since the treatment of `file' was inconsistent. When it appeared after `next`, it was a keyword. Otherwise, it was a regular identifier. The old usage is still accepted. However, `gawk` will generate a warning message, and support for `next file` will eventually be discontinued in a future version of `gawk`.

The `exit` Statement

The `exit` statement causes `awk` to immediately stop executing the current rule and to stop processing input; any remaining input is ignored. It looks like this:

```
exit [return code]
```

If an `exit` statement is executed from a `BEGIN` rule the program stops processing everything immediately. No input records are read. However, if an `END` rule is present, it is executed (see section The `BEGIN` and `END` Special Patterns).

If `exit` is used as part of an `END` rule, it causes the program to stop immediately.

An `exit` statement that is not part of a `BEGIN` or `END` rule stops the execution of any further automatic rules for the current record, skips reading any remaining input records, and executes the `END` rule if there is one.

If you do not want the `END` rule to do its job in this case, you can set a variable to non-zero before the `exit` statement, and check that variable in the `END` rule. See section Assertions, for an example that does this.

If an argument is supplied to `exit`, its value is used as the exit status code for the `awk` process. If no argument is supplied, `exit` returns status zero (success). In the case where an argument is supplied to a first `exit` statement, and then `exit` is called a second time with no argument, the previously supplied exit value is used (d.c.).

For example, let's say you've discovered an error condition you really don't know how to handle. Conventionally, programs report this by exiting with a non-zero status. Your `awk` program can do this using an `exit` statement with a non-zero argument. Here is an example:


```
BEGIN {
       if (("date" | getline date_now) < 0) {
         print "Can't get system date" > "/dev/stderr"
         exit 1
       }
       print "current date is", date_now
       close("date")
}
```

Go to the previous, next section.

Go to the previous, next section.

Built-in Variables

Most `awk` variables are available for you to use for your own purposes; they never change except when your program assigns values to them, and never affect anything except when your program examines them. However, a few variables in `awk` have special built-in meanings. Some of them `awk` examines automatically, so that they enable you to tell `awk` how to do certain things. Others are set automatically by `awk`, so that they carry information from the internal workings of `awk` to your program.

This chapter documents all the built-in variables of `gawk`. Most of them are also documented in the chapters describing their areas of activity.

Built-in Variables that Control awk

This is an alphabetical list of the variables which you can change to control how `awk` does certain things. Those variables that are specific to `gawk` are marked with an asterisk, `*'.

`CONVFMT`

This string controls conversion of numbers to strings (see section Conversion of Strings and Numbers). It works by being passed, in effect, as the first argument to the `sprintf` function (see section Built-in Functions for String Manipulation). Its default value is `"%.6g"`. `CONVFMT` was introduced by the POSIX standard.

`FIELDWIDTHS *`

This is a space separated list of columns that tells `gawk` how to split input with fixed, columnar boundaries. It is an experimental feature. Assigning to `FIELDWIDTHS` overrides the use of `FS` for field splitting. See section Reading Fixed-width Data, for more information.

If `gawk` is in compatibility mode (see section Command Line Options), then `FIELDWIDTHS` has no special meaning, and field splitting operations are done based exclusively on the value of `FS`.

`FS`

`FS` is the input field separator (see section Specifying How Fields are Separated). The value is a single-character string or a multi-character regular expression that matches the separations between fields in an input record. If the value is the null string (`""`), then each character in the record becomes a separate field.

The default value is `" "`, a string consisting of a single space. As a special exception, this value means that any sequence of spaces and tabs is a single separator. It also causes spaces and tabs at the beginning and end of a record to be ignored.

You can set the value of `FS` on the command line using the `-F' option:

```
awk -F, 'program' input-files
```

If gawk is using FIELDWIDTHS for field-splitting, assigning a value to FS will cause gawk to return to the normal, FS-based, field splitting. An easy way to do this is to simply say `FS = FS', perhaps with an explanatory comment.

IGNORECASE *

If IGNORECASE is non-zero or non-null, then all string comparisons, and all regular expression matching are case-independent. Thus, regexp matching with `~' and `!~', and the gensub, gsub, index, match, split and sub functions, record termination with RS, and field splitting with FS all ignore case when doing their particular regexp operations. See section Case-sensitivity in Matching.

If gawk is in compatibility mode (see section Command Line Options), then IGNORECASE has no special meaning, and string and regexp operations are always case-sensitive.

OFMT

This string controls conversion of numbers to strings (see section Conversion of Strings and Numbers) for printing with the print statement. It works by being passed, in effect, as the first argument to the sprintf function (see section Built-in Functions for String Manipulation). Its default value is "%.6g". Earlier versions of awk also used OFMT to specify the format for converting numbers to strings in general expressions; this is now done by CONVFMT.

OFS

This is the output field separator (see section Output Separators). It is output between the fields output by a print statement. Its default value is " ", a string consisting of a single space.

ORS

This is the output record separator. It is output at the end of every print statement. Its default value is "\n". (See section Output Separators.)

RS

This is awk's input record separator. Its default value is a string containing a single newline character, which means that an input record consists of a single line of text. It can also be the null string, in which case records are separated by runs of blank lines, or a regexp, in which case records are separated by matches of the regexp in the input text. (See section How Input is Split into Records.)

SUBSEP

SUBSEP is the subscript separator. It has the default value of "\034", and is used to separate the parts of the indices of a multi-dimensional array. Thus, the expression foo["A", "B"] really accesses foo["A\034B"] (see section Multi-dimensional Arrays).

Built-in Variables that Convey Information

This is an alphabetical list of the variables that are set automatically by awk on certain occasions in order to provide information to your program. Those variables that are specific to gawk are marked with an asterisk, `*'.

ARGC

ARGV

The command-line arguments available to `awk` programs are stored in an array called `ARGV`. `ARGC` is the number of command-line arguments present. See section Other Command Line Arguments. Unlike most `awk` arrays, `ARGV` is indexed from zero to `ARGC` - 1. For example:

```
$ awk 'BEGIN {
>         for (i = 0; i < ARGC; i++)
>             print ARGV[i]
>       }' inventory-shipped BBS-list
-| awk
-| inventory-shipped
-| BBS-list
```

In this example, `ARGV[0]` contains `"awk"`, `ARGV[1]` contains `"inventory-shipped"`, and `ARGV[2]` contains `"BBS-list"`. The value of `ARGC` is three, one more than the index of the last element in `ARGV`, since the elements are numbered from zero.

The names `ARGC` and `ARGV`, as well as the convention of indexing the array from zero to `ARGC` - 1, are derived from the C language's method of accessing command line arguments. See section Using ARGC and ARGV, for information about how `awk` uses these variables.

ARGIND *

The index in `ARGV` of the current file being processed. Every time `gawk` opens a new data file for processing, it sets `ARGIND` to the index in `ARGV` of the file name. When `gawk` is processing the input files, it is always true that `FILENAME == ARGV[ARGIND]'.

This variable is useful in file processing; it allows you to tell how far along you are in the list of data files, and to distinguish between successive instances of the same filename on the command line.

While you can change the value of `ARGIND` within your `awk` program, `gawk` will automatically set it to a new value when the next file is opened.

This variable is a `gawk` extension. In other `awk` implementations, or if `gawk` is in compatibility mode (see section Command Line Options), it is not special.

ENVIRON

An associative array that contains the values of the environment. The array indices are the environment variable names; the values are the values of the particular environment variables. For example, `ENVIRON["HOME"]` might be `/home/arnold'. Changing this array does not affect the environment passed on to any programs that `awk` may spawn via redirection or the `system` function. (In a future version of `gawk`, it may do so.)

Some operating systems may not have environment variables. On such systems, the `ENVIRON` array is empty (except for `ENVIRON["AWKPATH"]`).

ERRNO *

If a system error occurs either doing a redirection for `getline`, during a read for `getline`, or during a `close` operation, then `ERRNO` will contain a string describing the error.

This variable is a `gawk` extension. In other `awk` implementations, or if `gawk` is in compatibility mode (see section Command Line Options), it is not special.

`FILENAME`

This is the name of the file that `awk` is currently reading. When no data files are listed on the command line, `awk` reads from the standard input, and `FILENAME` is set to `"-"`. `FILENAME` is changed each time a new file is read (see section Reading Input Files). Inside a `BEGIN` rule, the value of `FILENAME` is `""`, since there are no input files being processed yet.(7) (d.c.)

`FNR`

`FNR` is the current record number in the current file. `FNR` is incremented each time a new record is read (see section Explicit Input with `getline`). It is reinitialized to zero each time a new input file is started.

`NF`

`NF` is the number of fields in the current input record. `NF` is set each time a new record is read, when a new field is created, or when `$0` changes (see section Examining Fields).

`NR`

This is the number of input records `awk` has processed since the beginning of the program's execution (see section How Input is Split into Records). `NR` is set each time a new record is read.

`RLENGTH`

`RLENGTH` is the length of the substring matched by the `match` function (see section Built-in Functions for String Manipulation). `RLENGTH` is set by invoking the `match` function. Its value is the length of the matched string, or -1 if no match was found.

`RSTART`

`RSTART` is the start-index in characters of the substring matched by the `match` function (see section Built-in Functions for String Manipulation). `RSTART` is set by invoking the `match` function. Its value is the position of the string where the matched substring starts, or zero if no match was found.

`RT` *

`RT` is set each time a record is read. It contains the input text that matched the text denoted by `RS`, the record separator.

This variable is a `gawk` extension. In other `awk` implementations, or if `gawk` is in compatibility mode (see section Command Line Options), it is not special.

A side note about `NR` and `FNR`. `awk` simply increments both of these variables each time it reads a record, instead of setting them to the absolute value of the number of records read. This means that your program can change these variables, and their new values will be incremented for each record (d.c.). For example:

```
$ echo '1
```


```
> 2
> 3
> 4' | awk 'NR == 2 { NR = 17 }
> { print NR }'
-| 1
-| 17
-| 18
-| 19
```

Before FNR was added to the awk language (see section Major Changes between V7 and SVR3.1), many awk programs used this feature to track the number of records in a file by resetting NR to zero when FILENAME changed.

Using ARGC and ARGV

In section Built-in Variables that Convey Information, you saw this program describing the information contained in ARGC and ARGV:

```
$ awk 'BEGIN {
>         for (i = 0; i < ARGC; i++)
>             print ARGV[i]
>       }' inventory-shipped BBS-list
-| awk
-| inventory-shipped
-| BBS-list
```

In this example, ARGV[0] contains "awk", ARGV[1] contains "inventory-shipped", and ARGV[2] contains "BBS-list".

Notice that the awk program is not entered in ARGV. The other special command line options, with their arguments, are also not entered. But variable assignments on the command line *are* treated as arguments, and do show up in the ARGV array.

Your program can alter ARGC and the elements of ARGV. Each time awk reaches the end of an input file, it uses the next element of ARGV as the name of the next input file. By storing a different string there, your program can change which files are read. You can use "-" to represent the standard input. By storing additional elements and incrementing ARGC you can cause additional files to be read.

If you decrease the value of ARGC, that eliminates input files from the end of the list. By recording the old value of ARGC elsewhere, your program can treat the eliminated arguments as something other than file names.

To eliminate a file from the middle of the list, store the null string ("") into ARGV in place of the file's name. As a special feature, awk ignores file names that have been replaced with the null string. You may also use the delete statement to remove elements from ARGV (see section The delete Statement).

All of these actions are typically done from the BEGIN rule, before actual processing of the input begins.

See section Splitting a Large File Into Pieces, and see section Duplicating Output Into Multiple Files for an example of each way of removing elements from ARGV.

The following fragment processes ARGV in order to examine, and then remove, command line options.

```
BEGIN {
    for (i = 1; i < ARGC; i++) {
        if (ARGV[i] == "-v")
            verbose = 1
        else if (ARGV[i] == "-d")
            debug = 1
        else if (ARGV[i] ~ /^-?/) {
            e = sprintf("%s: unrecognized option -- %c",
                    ARGV[0], substr(ARGV[i], 1, ,1))
            print e > "/dev/stderr"
        } else
            break
        delete ARGV[i]
    }
}
```

Go to the previous, next section.

Go to the previous, next section.

Arrays in awk

An array is a table of values, called elements. The elements of an array are distinguished by their indices. Indices may be either numbers or strings. `awk` maintains a single set of names that may be used for naming variables, arrays and functions (see section User-defined Functions). Thus, you cannot have a variable and an array with the same name in the same `awk` program.

Introduction to Arrays

The `awk` language provides one-dimensional arrays for storing groups of related strings or numbers.

Every `awk` array must have a name. Array names have the same syntax as variable names; any valid variable name would also be a valid array name. But you cannot use one name in both ways (as an array and as a variable) in one `awk` program.

Arrays in `awk` superficially resemble arrays in other programming languages; but there are fundamental differences. In `awk`, you don't need to specify the size of an array before you start to use it. Additionally, any number or string in `awk` may be used as an array index, not just consecutive integers.

In most other languages, you have to declare an array and specify how many elements or components it contains. In such languages, the declaration causes a contiguous block of memory to be allocated for that many elements. An index in the array usually must be a positive integer; for example, the index zero specifies the first element in the array, which is actually stored at the beginning of the block of memory. Index one specifies the second element, which is stored in memory right after the first element, and so on. It is impossible to add more elements to the array, because it has room for only as many elements as you declared. (Some languages allow arbitrary starting and ending indices, e.g., `15 .. 27', but the size of the array is still fixed when the array is declared.)

A contiguous array of four elements might look like this, conceptually, if the element values are eight, `"foo"`, `""` and 30:

Only the values are stored; the indices are implicit from the order of the values. Eight is the value at index zero, because eight appears in the position with zero elements before it.

Arrays in `awk` are different: they are associative. This means that each array is a collection of pairs: an index, and its corresponding array element value:

```
Element 4     Value 30
Element 2     Value "foo"
Element 1     Value 8
Element 3     Value ""
```

We have shown the pairs in jumbled order because their order is irrelevant.

One advantage of associative arrays is that new pairs can be added at any time. For example, suppose we add to the above array a tenth element whose value is `"number ten"`. The result is this:

```
Element 10    Value "number ten"
Element 4     Value 30
Element 2     Value "foo"
Element 1     Value 8
Element 3     Value ""
```

Now the array is sparse, which just means some indices are missing: it has elements 1--4 and 10, but doesn't have elements 5, 6, 7, 8, or 9.

Another consequence of associative arrays is that the indices don't have to be positive integers. Any number, or even a string, can be an index. For example, here is an array which translates words from English into French:

```
Element "dog" Value "chien"
Element "cat" Value "chat"
Element "one" Value "un"
Element 1     Value "un"
```

Here we decided to translate the number one in both spelled-out and numeric form--thus illustrating that a single array can have both numbers and strings as indices. (In fact, array subscripts are always strings; this is discussed in more detail in section Using Numbers to Subscript Arrays.)

When `awk` creates an array for you, e.g., with the `split` built-in function, that array's indices are consecutive integers starting at one. (See section Built-in Functions for String Manipulation.)

Referring to an Array Element

The principal way of using an array is to refer to one of its elements. An array reference is an expression which looks like this:

```
array[index]
```

Here, array is the name of an array. The expression index is the index of the element of the array that you want.

The value of the array reference is the current value of that array element. For example, `foo[4.3]` is an expression for the element of array `foo` at index `4.3'.

If you refer to an array element that has no recorded value, the value of the reference is `""`, the null string. This includes elements to which you have not assigned any value, and elements that have been deleted (see section The `delete` Statement). Such a reference automatically creates that array element, with the null string as its value. (In some cases, this is unfortunate, because it might waste memory inside `awk`.)

You can find out if an element exists in an array at a certain index with the expression:

```
index in array
```

This expression tests whether or not the particular index exists, without the side effect of creating that element if it is not present. The expression has the value one (true) if `array[index]` exists, and zero (false) if it does not exist.

For example, to test whether the array `frequencies` contains the index `2', you could write this statement:

```
if (2 in frequencies)
    print "Subscript 2 is present."
```

Note that this is *not* a test of whether or not the array `frequencies` contains an element whose *value* is two. (There is no way to do that except to scan all the elements.) Also, this *does not* create `frequencies[2]`, while the following (incorrect) alternative would do so:

```
if (frequencies[2] != "")
    print "Subscript 2 is present."
```

Assigning Array Elements

Array elements are lvalues: they can be assigned values just like `awk` variables:

```
array[subscript] = value
```

Here array is the name of your array. The expression subscript is the index of the element of the array that you want to assign a value. The expression value is the value you are assigning to that element of the array.

Basic Array Example

The following program takes a list of lines, each beginning with a line number, and prints them out in order of line number. The line numbers are not in order, however, when they are first read: they are scrambled. This program sorts the lines by making an array using the line numbers as subscripts. It then prints out the lines in sorted order of their numbers. It is a very simple program, and gets confused if it encounters repeated numbers, gaps, or lines that don't begin with a number.

```
{
  if ($1 > max)
    max = $1
  arr[$1] = $0
}
```


```
END {
  for (x = 1; x <= max; x++)
    print arr[x]
}
```

The first rule keeps track of the largest line number seen so far; it also stores each line into the array `arr`, at an index that is the line's number.

The second rule runs after all the input has been read, to print out all the lines.

When this program is run with the following input:

```
5  I am the Five man
2  Who are you?  The new number two!
4  . . . And four on the floor
1  Who is number one?
3  I three you.
```

its output is this:

```
1  Who is number one?
2  Who are you?  The new number two!
3  I three you.
4  . . . And four on the floor
5  I am the Five man
```

If a line number is repeated, the last line with a given number overrides the others.

Gaps in the line numbers can be handled with an easy improvement to the program's `END` rule:

```
END {
  for (x = 1; x <= max; x++)
    if (x in arr)
      print arr[x]
}
```

Scanning All Elements of an Array

In programs that use arrays, you often need a loop that executes once for each element of an array. In other languages, where arrays are contiguous and indices are limited to positive integers, this is easy: you can find all the valid indices by counting from the lowest index up to the highest. This technique won't do the job in `awk`, since any number or string can be an array index. So `awk` has a special kind of `for` statement for scanning an array:

```
for (var in array)
```


```
body
```

This loop executes body once for each index in array that your program has previously used, with the variable var set to that index.

Here is a program that uses this form of the `for` statement. The first rule scans the input records and notes which words appear (at least once) in the input, by storing a one into the array `used` with the word as index. The second rule scans the elements of `used` to find all the distinct words that appear in the input. It prints each word that is more than 10 characters long, and also prints the number of such words. See section Built-in Functions for String Manipulation, for more information on the built-in function `length`.

```
# Record a 1 for each word that is used at least once.
{
    for (i = 1; i <= NF; i++)
        used[$i] = 1
}

# Find number of distinct words more than 10 characters long.
END {
    for (x in used)
        if (length(x) > 10) {
            ++num_long_words
            print x
        }
    print num_long_words, "words longer than 10 characters"
}
```

See section Generating Word Usage Counts, for a more detailed example of this type.

The order in which elements of the array are accessed by this statement is determined by the internal arrangement of the array elements within `awk` and cannot be controlled or changed. This can lead to problems if new elements are added to array by statements in the loop body; you cannot predict whether or not the `for` loop will reach them. Similarly, changing var inside the loop may produce strange results. It is best to avoid such things.

The `delete` Statement

You can remove an individual element of an array using the `delete` statement:

```
delete array[index]
```

Once you have deleted an array element, you can no longer obtain any value the element once had. It is as if you had never referred to it and had never given it any value.

Here is an example of deleting elements in an array:


```
for (i in frequencies)
  delete frequencies[i]
```

This example removes all the elements from the array `frequencies`.

If you delete an element, a subsequent `for` statement to scan the array will not report that element, and the `in` operator to check for the presence of that element will return zero (i.e. false):

```
delete foo[4]
if (4 in foo)
    print "This will never be printed"
```

It is important to note that deleting an element is *not* the same as assigning it a null value (the empty string, `""`).

```
foo[4] = ""
if (4 in foo)
  print "This is printed, even though foo[4] is empty"
```

It is not an error to delete an element that does not exist.

You can delete all the elements of an array with a single statement, by leaving off the subscript in the `delete` statement.

```
delete array
```

This ability is a `gawk` extension; it is not available in compatibility mode (see section Command Line Options).

Using this version of the `delete` statement is about three times more efficient than the equivalent loop that deletes each element one at a time.

The following statement provides a portable, but non-obvious way to clear out an array.

```
# thanks to Michael Brennan for pointing this out
split("", array)
```

The `split` function (see section Built-in Functions for String Manipulation) clears out the target array first. This call asks it to split apart the null string. Since there is no data to split out, the function simply clears the array and then returns.

Using Numbers to Subscript Arrays

An important aspect of arrays to remember is that *array subscripts are always strings*. If you use a numeric value as a subscript, it will be converted to a string value before it is used for subscripting (see section Conversion of Strings and Numbers).

This means that the value of the built-in variable `CONVFMT` can potentially affect how your program accesses elements of an array. For example:

```
xyz = 12.153
data[xyz] = 1
CONVFMT = "%2.2f"
if (xyz in data)
    printf "%s is in data\n", xyz
else
    printf "%s is not in data\n", xyz
```

This prints `12.15 is not in data'. The first statement gives `xyz` a numeric value. Assigning to `data[xyz]` subscripts `data` with the string value `"12.153"` (using the default conversion value of `CONVFMT`, `"%.6g"`), and assigns one to `data["12.153"]`. The program then changes the value of `CONVFMT`. The test `(xyz in data)' generates a new string value from `xyz`, this time `"12.15"`, since the value of `CONVFMT` only allows two significant digits. This test fails, since `"12.15"` is a different string from `"12.153"`.

According to the rules for conversions (see section Conversion of Strings and Numbers), integer values are always converted to strings as integers, no matter what the value of `CONVFMT` may happen to be. So the usual case of:

```
for (i = 1; i <= maxsub; i++)
    do something with array[i]
```

will work, no matter what the value of `CONVFMT`.

Like many things in `awk`, the majority of the time things work as you would expect them to work. But it is useful to have a precise knowledge of the actual rules, since sometimes they can have a subtle effect on your programs.

Using Uninitialized Variables as Subscripts

Suppose you want to print your input data in reverse order. A reasonable attempt at a program to do so (with some test data) might look like this:

```
$ echo 'line 1
> line 2
> line 3' | awk '{ l[lines] = $0; ++lines }
> END {
>     for (i = lines-1; i >= 0; --i)
>        print l[i]
> }'
-| line 3
-| line 2
```

Unfortunately, the very first line of input data did not come out in the output!

At first glance, this program should have worked. The variable `lines` is uninitialized, and uninitialized variables have the numeric value zero. So, the value of `l[0]` should have been printed.

The issue here is that subscripts for `awk` arrays are **always** strings. And uninitialized variables, when used as strings, have the value `""`, not zero. Thus, `line 1' ended up stored in `l[""]`.

The following version of the program works correctly:

```
{ l[lines++] = $0 }
END {
    for (i = lines - 1; i >= 0; --i)
       print l[i]
}
```

Here, the `++' forces `l` to be numeric, thus making the "old value" numeric zero, which is then converted to `"0"` as the array subscript.

As we have just seen, even though it is somewhat unusual, the null string (`""`) is a valid array subscript (d.c.). If `--lint' is provided on the command line (see section Command Line Options), `gawk` will warn about the use of the null string as a subscript.

Multi-dimensional Arrays

A multi-dimensional array is an array in which an element is identified by a sequence of indices, instead of a single index. For example, a two-dimensional array requires two indices. The usual way (in most languages, including `awk`) to refer to an element of a two-dimensional array named `grid` is with `grid[x,y]`.

Multi-dimensional arrays are supported in `awk` through concatenation of indices into one string. What happens is that `awk` converts the indices into strings (see section Conversion of Strings and Numbers) and concatenates them together, with a separator between them. This creates a single string that describes the values of the separate indices. The combined string is used as a single index into an ordinary, one-dimensional array. The separator used is the value of the built-in variable SUBSEP.

For example, suppose we evaluate the expression `foo[5,12] = "value"' when the value of SUBSEP is `"@"`. The numbers five and 12 are converted to strings and concatenated with an `@' between them, yielding `"5@12"`; thus, the array element `foo["5@12"]` is set to `"value"`.

Once the element's value is stored, `awk` has no record of whether it was stored with a single index or a sequence of indices. The two expressions `foo[5,12]' and `foo[5 SUBSEP 12]' are always equivalent.

The default value of SUBSEP is the string `"\034"`, which contains a non-printing character that is unlikely to appear in an `awk` program or in most input data.

The usefulness of choosing an unlikely character comes from the fact that index values that contain a

string matching SUBSEP lead to combined strings that are ambiguous. Suppose that SUBSEP were '@', then `foo["a@b", "c"]' and `foo["a", "b@c"]' would be indistinguishable because both would actually be stored as `foo["a@b@c"]'.

You can test whether a particular index-sequence exists in a "multi-dimensional" array with the same operator `in' used for single dimensional arrays. Instead of a single index as the left-hand operand, write the whole sequence of indices, separated by commas, in parentheses:

```
(subscript1, subscript2, ...) in array
```

The following example treats its input as a two-dimensional array of fields; it rotates this array 90 degrees clockwise and prints the result. It assumes that all lines have the same number of elements.

```
awk '{
     if (max_nf < NF)
          max_nf = NF
     max_nr = NR
     for (x = 1; x <= NF; x++)
          vector[x, NR] = $x
}

END {
     for (x = 1; x <= max_nf; x++) {
          for (y = max_nr; y >= 1; --y)
               printf("%s ", vector[x, y])
          printf("\n")
     }
}'
```

When given the input:

```
1 2 3 4 5 6
2 3 4 5 6 1
3 4 5 6 1 2
4 5 6 1 2 3
```

it produces:

```
4 3 2 1
5 4 3 2
6 5 4 3
1 6 5 4
2 1 6 5
3 2 1 6
```

Scanning Multi-dimensional Arrays

There is no special `for` statement for scanning a "multi-dimensional" array; there cannot be one, because in truth there are no multi-dimensional arrays or elements; there is only a multi-dimensional *way of accessing* an array.

However, if your program has an array that is always accessed as multi-dimensional, you can get the effect of scanning it by combining the scanning `for` statement (see section Scanning All Elements of an Array) with the `split` built-in function (see section Built-in Functions for String Manipulation). It works like this:

```
for (combined in array) {
  split(combined, separate, SUBSEP)
  ...
}
```

This sets `combined` to each concatenated, combined index in the array, and splits it into the individual indices by breaking it apart where the value of `SUBSEP` appears. The split-out indices become the elements of the array `separate`.

Thus, suppose you have previously stored a value in `array[1, "foo"]`; then an element with index `"1\034foo"` exists in `array`. (Recall that the default value of `SUBSEP` is the character with code 034.) Sooner or later the `for` statement will find that index and do an iteration with `combined` set to `"1\034foo"`. Then the `split` function is called as follows:

```
split("1\034foo", separate, "\034")
```

The result of this is to set `separate[1]` to `"1"` and `separate[2]` to `"foo"`. Presto, the original sequence of separate indices has been recovered.

Go to the previous, next section.

Go to the previous, next section.

Built-in Functions

Built-in functions are functions that are always available for your `awk` program to call. This chapter defines all the built-in functions in `awk`; some of them are mentioned in other sections, but they are summarized here for your convenience. (You can also define new functions yourself. See section User-defined Functions.)

Calling Built-in Functions

To call a built-in function, write the name of the function followed by arguments in parentheses. For example, `atan2(y + z, 1)' is a call to the function `atan2`, with two arguments.

Whitespace is ignored between the built-in function name and the open-parenthesis, but we recommend that you avoid using whitespace there. User-defined functions do not permit whitespace in this way, and you will find it easier to avoid mistakes by following a simple convention which always works: no whitespace after a function name.

Each built-in function accepts a certain number of arguments. In some cases, arguments can be omitted. The defaults for omitted arguments vary from function to function and are described under the individual functions. In some `awk` implementations, extra arguments given to built-in functions are ignored. However, in `gawk`, it is a fatal error to give extra arguments to a built-in function.

When a function is called, expressions that create the function's actual parameters are evaluated completely before the function call is performed. For example, in the code fragment:

```
i = 4
j = sqrt(i++)
```

the variable `i` is set to five before `sqrt` is called with a value of four for its actual parameter.

The order of evaluation of the expressions used for the function's parameters is undefined. Thus, you should not write programs that assume that parameters are evaluated from left to right or from right to left. For example,

```
i = 5
j = atan2(i++, i *= 2)
```

If the order of evaluation is left to right, then `i` first becomes six, and then 12, and `atan2` is called with the two arguments six and 12. But if the order of evaluation is right to left, `i` first becomes 10, and then 11, and `atan2` is called with the two arguments 11 and 10.

Numeric Built-in Functions

Here is a full list of built-in functions that work with numbers. Optional parameters are enclosed in square brackets ("[" and "]").

`int(x)`

This produces the nearest integer to x, located between x and zero, truncated toward zero.

For example, `int(3)` is three, `int(3.9)` is three, `int(-3.9)` is -3, and `int(-3)` is -3 as well.

`sqrt(x)`

This gives you the positive square root of x. It reports an error if x is negative. Thus, `sqrt(4)` is two.

`exp(x)`

This gives you the exponential of x (`e ^ x`), or reports an error if x is out of range. The range of values x can have depends on your machine's floating point representation.

`log(x)`

This gives you the natural logarithm of x, if x is positive; otherwise, it reports an error.

`sin(x)`

This gives you the sine of x, with x in radians.

`cos(x)`

This gives you the cosine of x, with x in radians.

`atan2(y, x)`

This gives you the arctangent of `y / x` in radians.

`rand()`

This gives you a random number. The values of `rand` are uniformly-distributed between zero and one. The value is never zero and never one.

Often you want random integers instead. Here is a user-defined function you can use to obtain a random non-negative integer less than n:

```
function randint(n) {
     return int(n * rand())
}
```

The multiplication produces a random real number greater than zero and less than `n`. We then make it an integer (using `int`) between zero and `n` - 1, inclusive.

Here is an example where a similar function is used to produce random integers between one and n. This program prints a new random number for each input record.

```
awk '
# Function to roll a simulated die.
```


```
function roll(n) { return 1 + int(rand() * n) }

# Roll 3 six-sided dice and
# print total number of points.
{
      printf("%d points\n",
             roll(6)+roll(6)+roll(6))
}'
```

Caution: In most awk implementations, including gawk, rand starts generating numbers from the same starting number, or seed, each time you run awk. Thus, a program will generate the same results each time you run it. The numbers are random within one awk run, but predictable from run to run. This is convenient for debugging, but if you want a program to do different things each time it is used, you must change the seed to a value that will be different in each run. To do this, use srand.

`srand([x])`

The function srand sets the starting point, or seed, for generating random numbers to the value x.

Each seed value leads to a particular sequence of random numbers.(8) Thus, if you set the seed to the same value a second time, you will get the same sequence of random numbers again.

If you omit the argument x, as in `srand()`, then the current date and time of day are used for a seed. This is the way to get random numbers that are truly unpredictable.

The return value of srand is the previous seed. This makes it easy to keep track of the seeds for use in consistently reproducing sequences of random numbers.

Built-in Functions for String Manipulation

The functions in this section look at or change the text of one or more strings. Optional parameters are enclosed in square brackets ("[" and "]").

`index(in, find)`

This searches the string in for the first occurrence of the string find, and returns the position in characters where that occurrence begins in the string in. For example:

```
$ awk 'BEGIN { print index("peanut", "an") }'
-| 3
```

If find is not found, index returns zero. (Remember that string indices in awk start at one.)

`length([string])`

This gives you the number of characters in string. If string is a number, the length of the digit string representing that number is returned. For example, `length("abcde")` is five. By contrast, `length(15 * 35)` works out to three. How? Well, 15 * 35 = 525, and 525 is then converted to the string `"525"`, which has three characters.

If no argument is supplied, `length` returns the length of `$0`.

In older versions of `awk`, you could call the `length` function without any parentheses. Doing so is marked as "deprecated" in the POSIX standard. This means that while you can do this in your programs, it is a feature that can eventually be removed from a future version of the standard. Therefore, for maximal portability of your `awk` programs, you should always supply the parentheses.

`match(string, regexp)`

The `match` function searches the string, string, for the longest, leftmost substring matched by the regular expression, regexp. It returns the character position, or index, of where that substring begins (one, if it starts at the beginning of string). If no match is found, it returns zero.

The `match` function sets the built-in variable `RSTART` to the index. It also sets the built-in variable `RLENGTH` to the length in characters of the matched substring. If no match is found, `RSTART` is set to zero, and `RLENGTH` to -1.

For example:

```
awk '{
       if ($1 == "FIND")
         regex = $2
       else {
         where = match($0, regex)
         if (where != 0)
           print "Match of", regex, "found at", \
                     where, "in", $0
       }
}'
```

This program looks for lines that match the regular expression stored in the variable `regex`. This regular expression can be changed. If the first word on a line is `FIND', `regex` is changed to be the second word on that line. Therefore, given:

```
FIND ru+n
My program runs
but not very quickly
FIND Melvin
JF+KM
This line is property of Reality Engineering Co.
Melvin was here.
```

`awk` prints:

```
Match of ru+n found at 12 in My program runs
Match of Melvin found at 1 in Melvin was here.
```

`split(string, array [, fieldsep])`

This divides string into pieces separated by fieldsep, and stores the pieces in array. The first pi is stored in `array[1]`, the second piece in `array[2]`, and so forth. The string value of the third argument, fieldsep, is a regexp describing where to split string (much as `FS` can be a regexp describing where to split input records). If the fieldsep is omitted, the value of `FS` is used. `split` returns the number of elements created.

The `split` function splits strings into pieces in a manner similar to the way input lines are split into fields. For example:

```
split("cul-de-sac", a, "-")
```

splits the string `cul-de-sac' into three fields using `-' as the separator. It sets the contents of the array `a` as follows:

```
a[1] = "cul"
a[2] = "de"
a[3] = "sac"
```

The value returned by this call to `split` is three.

As with input field-splitting, when the value of fieldsep is `" "`, leading and trailing whitespace is ignored, and the elements are separated by runs of whitespace.

Also as with input field-splitting, if fieldsep is the null string, each individual character in the string is split into its own array element. (This is a `gawk`-specific extension.)

Recent implementations of `awk`, including `gawk`, allow the third argument to be a regexp constant (`/abc/`), as well as a string (d.c.). The POSIX standard allows this as well.

Before splitting the string, `split` deletes any previously existing elements in the array array (d.c.).

`sprintf(format, expression1,...)`

This returns (without printing) the string that `printf` would have printed out with the same arguments (see section Using `printf` Statements for Fancier Printing). For example:

```
sprintf("pi = %.2f (approx.)", 22/7)
```

returns the string `"pi = 3.14 (approx.)"`.

`sub(regexp, replacement [, target])`

The `sub` function alters the value of target. It searches this value, which is treated as a string, for the leftmost longest substring matched by the regular expression, regexp, extending this match as far as possible. Then the entire string is changed by replacing the matched text with replacement. The modified string becomes the new value of target.

This function is peculiar because target is not simply used to compute a value, and not just any expression will do: it must be a variable, field or array element, so that `sub` can store a modified value there. If this argument is omitted, then the default is to use and alter `$0`.

For example:

```
str = "water, water, everywhere"
sub(/at/, "ith", str)
```

sets `str` to `"wither, water, everywhere"`, by replacing the leftmost, longest occurrence of `at' with `ith'.

The `sub` function returns the number of substitutions made (either one or zero).

If the special character `&' appears in replacement, it stands for the precise substring that was matched by regexp. (If the regexp can match more than one string, then this precise substring may vary.) For example:

```
awk '{ sub(/candidate/, "& and his wife"); print }'
```

changes the first occurrence of `candidate' to `candidate and his wife' on each input line.

Here is another example:

```
awk 'BEGIN {
        str = "daabaaa"
        sub(/a*/, "c&c", str)
        print str
}'
-| dcaacbaaa
```

This shows how `&' can represent a non-constant string, and also illustrates the "leftmost, longest" rule in regexp matching (see section How Much Text Matches?).

The effect of this special character (`&') can be turned off by putting a backslash before it in the string. As usual, to insert one backslash in the string, you must write two backslashes. Therefore, write `\\&' in a string constant to include a literal `&' in the replacement. For example, here is how to replace the first `|' on each line with an `&':

```
awk '{ sub(/\|/, "\\&"); print }'
```

Note: As mentioned above, the third argument to `sub` must be a variable, field or array reference. Some versions of `awk` allow the third argument to be an expression which is not an lvalue. In such a case, `sub` would still search for the pattern and return zero or one, but the result of the substitution (if any) would be thrown away because there is no place to put it. Such versions of `awk` accept expressions like this:

```
sub(/USA/, "United States", "the USA and Canada")
```

This is considered erroneous in `gawk`.

```
gsub(regexp, replacement [, target])
```

This is similar to the `sub` function, except `gsub` replaces *all* of the longest, leftmost, *non-overlapping* matching substrings it can find. The `g' in `gsub` stands for "global," which means replace everywhere. For example:

```
awk '{ gsub(/Britain/, "United Kingdom"); print }'
```

replaces all occurrences of the string `Britain' with `United Kingdom' for all input records.

The `gsub` function returns the number of substitutions made. If the variable to be searched and altered, target, is omitted, then the entire input record, `$0`, is used.

As in `sub`, the characters `&' and `\' are special, and the third argument must be an lvalue.

`gensub(regexp, replacement, how [, target])`

`gensub` is a general substitution function. Like `sub` and `gsub`, it searches the target string target for matches of the regular expression regexp. Unlike `sub` and `gsub`, the modified string is returned as the result of the function, and the original target string is *not* changed. If how is a string beginning with `g' or `G', then it replaces all matches of regexp with replacement. Otherwise, how is a number indicating which match of regexp to replace. If no target is supplied, `$0` is used instead.

`gensub` provides an additional feature that is not available in `sub` or `gsub`: the ability to specify components of a regexp in the replacement text. This is done by using parentheses in the regexp to mark the components, and then specifying `\n' in the replacement text, where n is a digit from one to nine. For example:

```
$ gawk '
> BEGIN {
>       a = "abc def"
>       b = gensub(/(.+) (.+)/, "\\2 \\1", "g", a)
>       print b
> }'
-| def abc
```

As described above for `sub`, you must type two backslashes in order to get one into the string.

In the replacement text, the sequence `\0' represents the entire matched text, as does the character `&'.

This example shows how you can use the third argument to control which match of the regexp should be changed.

```
$ echo a b c a b c |
> gawk '{ print gensub(/a/, "AA", 2) }'
-| a b c AA b c
```

In this case, `$0` is used as the default target string. `gensub` returns the new string as its result, which is passed directly to `print` for printing.

If the how argument is a string that does not begin with `g' or `G', or if it is a number that is less than zero, only one substitution is performed.

`gensub` is a `gawk` extension; it is not available in compatibility mode (see section Command Line Options).

`substr(string, start [, length])`

This returns a length-character-long substring of string, starting at character number start. The first character of a string is character number one. For example, `substr("washington", 5, 3)` returns `"ing"`.

If length is not present, this function returns the whole suffix of string that begins at character number start. For example, `substr("washington", 5)` returns `"ington"`. The whole suffix is also returned if length is greater than the number of characters remaining in the string, counting from character number start.

`tolower(string)`

This returns a copy of string, with each upper-case character in the string replaced with its corresponding lower-case character. Non-alphabetic characters are left unchanged. For example, `tolower("MiXeD cAsE 123")` returns `"mixed case 123"`.

`toupper(string)`

This returns a copy of string, with each lower-case character in the string replaced with its corresponding upper-case character. Non-alphabetic characters are left unchanged. For example, `toupper("MiXeD cAsE 123")` returns `"MIXED CASE 123"`.

More About `\' and `&' with sub, gsub and gensub

When using `sub`, `gsub` or `gensub`, and trying to get literal backslashes and ampersands into the replacement text, you need to remember that there are several levels of escape processing going on.

First, there is the lexical level, which is when `awk` reads your program, and builds an internal copy of your program that can be executed.

Then there is the run-time level, when `awk` actually scans the replacement string to determine what to generate.

At both levels, `awk` looks for a defined set of characters that can come after a backslash. At the lexical level, it looks for the escape sequences listed in section Escape Sequences. Thus, for every `\' that `awk` will process at the run-time level, you type two `\'s at the lexical level. When a character that is not valid for an escape sequence follows the `\', Unix `awk` and `gawk` both simply remove the initial `\', and put the following character into the string. Thus, for example, `"a\qb"` is treated as `"aqb"`.

At the run-time level, the various functions handle sequences of `\' and `&' differently. The situation is (sadly) somewhat complex.

Historically, the `sub` and `gsub` functions treated the two character sequence `\&' specially; this sequence was replaced in the generated text with a single `&'. Any other `\' within the replacement string that did not precede an `&' was passed through unchanged. To illustrate with a table:

This table shows both the lexical level processing, where an odd number of backslashes becomes an even number at the run time level, and the run-time processing done by `sub`. (For the sake of simplicity, the rest of the tables below only show the case of even numbers of `\'s entered at the lexical level.)

The problem with the historical approach is that there is no way to get a literal `\' followed by the matched text.

The 1992 POSIX standard attempted to fix this problem. The standard says that `sub` and `gsub` look for either a `\' or an `&' after the `\'. If either one follows a `\', that character is output literally. The interpretation of `\' and `&' then becomes like this:

This would appear to solve the problem. Unfortunately, the phrasing of the standard is unusual. It says, in effect, that `\' turns off the special meaning of any following character, but that for anything other than `\' and `&', such special meaning is undefined. This wording leads to two problems.

1. Backslashes must now be doubled in the replacement string, breaking historical `awk` programs.
2. To make sure that an `awk` program is portable, *every* character in the replacement string must be preceded with a backslash.(9)

The POSIX standard is under revision.(10) Because of the above problems, proposed text for the revised standard reverts to rules that correspond more closely to the original existing practice. The proposed rules have special cases that make it possible to produce a `\' preceding the matched text.

In a nutshell, at the run-time level, there are now three special sequences of characters, `\\\&', `\\&' and `\&', whereas historically, there was only one. However, as in the historical case, any `\' that is not part of one of these three sequences is not special, and appears in the output literally.

`gawk` 3.0 follows these proposed POSIX rules for `sub` and `gsub`. Whether these proposed rules will actually become codified into the standard is unknown at this point. Subsequent `gawk` releases will track the standard and implement whatever the final version specifies; this book will be updated as well.

The rules for `gensub` are considerably simpler. At the run-time level, whenever `gawk` sees a `\', if the following character is a digit, then the text that matched the corresponding parenthesized subexpression is placed in the generated output. Otherwise, no matter what the character after the `\' is, that character will appear in the generated text, and the `\' will not.

Because of the complexity of the lexical and run-time level processing, and the special cases for `sub` and `gsub`, we recommend the use of `gawk` and `gensub` for when you have to do substitutions.

Built-in Functions for Input/Output

The following functions are related to Input/Output (I/O). Optional parameters are enclosed in square brackets ("[" and "]").

`close(filename)`

Close the file filename, for input or output. The argument may alternatively be a shell command that was used for redirecting to or from a pipe; then the pipe is closed. See section Closing Input and Output Files and Pipes, for more information.

`fflush([filename])`

Flush any buffered output associated filename, which is either a file opened for writing, or a shell command for redirecting output to a pipe.

Many utility programs will buffer their output; they save information to be written to a disk file or terminal in memory, until there is enough for it to be worthwhile to send the data to the ouput device. This is often more efficient than writing every little bit of information as soon as it is ready. However, sometimes it is necessary to force a program to flush its buffers; that is, write the information to its destination, even if a buffer is not full. This is the purpose of the `fflush` function; `gawk` too buffers its output, and the `fflush` function can be used to force `gawk` to flush its buffers.

`fflush` is a recent (1994) addition to the Bell Labs research version of `awk`; it is not part of the POSIX standard, and will not be available if `--posix' has been specified on the command line (see section Command Line Options).

`gawk` extends the `fflush` function in two ways. This first is to allow no argument at all. In this case, the buffer for the standard output is flushed. The second way is to allow the null string (`""`) as the argument. In this case, the buffers for *all* open output files and pipes are flushed.

`fflush` returns zero if the buffer was successfully flushed, and nonzero otherwise.

`system(command)`

The system function allows the user to execute operating system commands and then return to the `awk` program. The `system` function executes the command given by the string command. It returns, as its value, the status returned by the command that was executed.

For example, if the following fragment of code is put in your `awk` program:

```
END {
     system("date | mail -s 'awk run done' root")
}
```

the system administrator will be sent mail when the `awk` program finishes processing input and begins its end-of-input processing.

Note that redirecting `print` or `printf` into a pipe is often enough to accomplish your task. However, if your `awk` program is interactive, `system` is useful for cranking up large self-contained programs, such as a shell or an editor.

Some operating systems cannot implement the `system` function. `system` causes a fatal error if it is not supported.

Controlling Output Buffering with `system`

The `fflush` function provides explicit control over output buffering for individual files and pipes. However, its use is not portable to many other `awk` implementations. An alternative method to flush output buffers is by calling `system` with a null string as its argument:


```
system("")   # flush output
```

gawk treats this use of the system function as a special case, and is smart enough not to run a shell (or other command interpreter) with the empty command. Therefore, with gawk, this idiom is not only useful, it is efficient. While this method should work with other awk implementations, it will not necessarily avoid starting an unnecessary shell. (Other implementations may only flush the buffer associated with the standard output, and not necessarily all buffered output.)

If you think about what a programmer expects, it makes sense that system should flush any pending output. The following program:

```
BEGIN {
     print "first print"
     system("echo system echo")
     print "second print"
}
```

must print

```
first print
system echo
second print
```

and not

```
system echo
first print
second print
```

If awk did not flush its buffers before calling system, the latter (undesirable) output is what you would see.

Functions for Dealing with Time Stamps

A common use for awk programs is the processing of log files containing time stamp information, indicating when a particular log record was written. Many programs log their time stamp in the form returned by the time system call, which is the number of seconds since a particular epoch. On POSIX systems, it is the number of seconds since Midnight, January 1, 1970, UTC.

In order to make it easier to process such log files, and to produce useful reports, gawk provides two functions for working with time stamps. Both of these are gawk extensions; they are not specified in the POSIX standard, nor are they in any other known version of awk.

Optional parameters are enclosed in square brackets ("[" and "]").

```
systime()
```

This function returns the current time as the number of seconds since the system epoch. On P[illegible] systems, this is the number of seconds since Midnight, January 1, 1970, UTC. It may be a diff[illegible] number on other systems.

`strftime([format [, timestamp]])`

This function returns a string. It is similar to the function of the same name in ANSI C. The time specified by timestamp is used to produce a string, based on the contents of the format string. The timestamp is in the same format as the value returned by the `systime` function. If no timestamp argument is supplied, `gawk` will use the current time of day as the time stamp. If no format argument is supplied, `strftime` uses `"%a %b %d %H:%M:%S %Z %Y"`. This format string produces output (almost) equivalent to that of the `date` utility. (Versions of `gawk` prior to 3.0 require the format argument.)

The `systime` function allows you to compare a time stamp from a log file with the current time of day. In particular, it is easy to determine how long ago a particular record was logged. It also allows you to produce log records using the "seconds since the epoch" format.

The `strftime` function allows you to easily turn a time stamp into human-readable information. It is similar in nature to the `sprintf` function (see section Built-in Functions for String Manipulation), in that it copies non-format specification characters verbatim to the returned string, while substituting date and time values for format specifications in the format string.

`strftime` is guaranteed by the ANSI C standard to support the following date format specifications:

`%a`

The locale's abbreviated weekday name.

`%A`

The locale's full weekday name.

`%b`

The locale's abbreviated month name.

`%B`

The locale's full month name.

`%c`

The locale's "appropriate" date and time representation.

`%d`

The day of the month as a decimal number (01--31).

`%H`

The hour (24-hour clock) as a decimal number (00--23).

`%I`

The hour (12-hour clock) as a decimal number (01--12).

`%j`

The day of the year as a decimal number (001--366).

`%m`

The month as a decimal number (01--12).

`%M`

The minute as a decimal number (00--59).

`%p`

The locale's equivalent of the AM/PM designations associated with a 12-hour clock.

`%S`

The second as a decimal number (00--61).(11)

`%U`

The week number of the year (the first Sunday as the first day of week one) as a decimal number (00--53).

`%w`

The weekday as a decimal number (0--6). Sunday is day zero.

`%W`

The week number of the year (the first Monday as the first day of week one) as a decimal number (00--53).

`%x`

The locale's "appropriate" date representation.

`%X`

The locale's "appropriate" time representation.

`%y`

The year without century as a decimal number (00--99).

`%Y`

The year with century as a decimal number (e.g., 1995).

`%Z`

The time zone name or abbreviation, or no characters if no time zone is determinable.

`%%`

A literal `%'.

If a conversion specifier is not one of the above, the behavior is undefined.(12)

Informally, a locale is the geographic place in which a program is meant to run. For example, a common way to abbreviate the date September 4, 1991 in the United States would be "9/4/91". In many countries in Europe, however, it would be abbreviated "4.9.91". Thus, the `%x' specification in a `"US"` locale might produce `9/4/91', while in a `"EUROPE"` locale, it might produce `4.9.91'. The ANSI C standard defines a default `"C"` locale, which is an environment that is typical of what most C programmers are used to.

A public-domain C version of `strftime` is supplied with `gawk` for systems that are not yet fully ANSI-compliant. If that version is used to compile `gawk` (see section Installing `gawk`), then the following additional format specifications are available:

`%D`

Equivalent to specifying `%m/%d/%y'.

`%e`

The day of the month, padded with a space if it is only one digit.

`%h`

Equivalent to `%b', above.

`%n`

A newline character (ASCII LF).

`%r`

Equivalent to specifying `%I:%M:%S %p'.

`%R`

Equivalent to specifying `%H:%M'.

`%T`

Equivalent to specifying `%H:%M:%S'.

`%t`

A tab character.

`%k`

The hour (24-hour clock) as a decimal number (0-23). Single digit numbers are padded with a space.

`%l`

The hour (12-hour clock) as a decimal number (1-12). Single digit numbers are padded with a space.

`%C`

The century, as a number between 00 and 99.

`%u`

The weekday as a decimal number [1 (Monday)--7].

`%V`

The week number of the year (the first Monday as the first day of week one) as a decimal number (01--53). The method for determining the week number is as specified by ISO 8601 (to wit: if the week containing January 1 has four or more days in the new year, then it is week one, otherwise it is week 53 of the previous year and the next week is week one).

`%G`

The year with century of the ISO week number, as a decimal number.

For example, January 1, 1993, is in week 53 of 1992. Thus, the year of its ISO week number is 1992, even though its year is 1993. Similarly, December 31, 1973, is in week 1 of 1974. Thus, the year of its ISO week number is 1974, even though its year is 1973.

`%g`

The year without century of the ISO week number, as a decimal number (00--99).

`%Ec %EC %Ex %Ey %EY %Od %Oe %OH %OI`

`%Om %OM %OS %Ou %OU %OV %Ow %OW %Oy`

These are "alternate representations" for the specifications that use only the second letter (`%c', `%C', and so on). They are recognized, but their normal representations are used.(13) (These facilitate compliance with the POSIX `date` utility.)

`%v`

The date in VMS format (e.g., 20-JUN-1991).

`%z`

The timezone offset in a +HHMM format (e.g., the format necessary to produce RFC-822/RFC-1036 date headers).

This example is an `awk` implementation of the POSIX `date` utility. Normally, the `date` utility prints the current date and time of day in a well known format. However, if you provide an argument to it that begins with a `+', `date` will copy non-format specifier characters to the standard output, and will interpret the current time according to the format specifiers in the string. For example:

```
$ date '+Today is %A, %B %d, %Y.'
-| Today is Thursday, July 11, 1991.
```

Here is the `gawk` version of the `date` utility. It has a shell "wrapper", to handle the `-u' option, which requires that `date` run as if the time zone was set to UTC.

```
#! /bin/sh
#
# date -- approximate the P1003.2 'date' command

case $1 in
-u)  TZ=GMT0     # use UTC
     export TZ
     shift ;;
esac

gawk 'BEGIN  {
    format = "%a %b %d %H:%M:%S %Z %Y"
    exitval = 0

    if (ARGC > 2)
        exitval = 1
    else if (ARGC == 2) {
        format = ARGV[1]
        if (format ~ /^\+/)
            format = substr(format, 2)   # remove leading +
    }
    print strftime(format)
```


```
    exit exitval
}' "$@"
```

Go to the previous, next section.

Go to the previous, next section.

User-defined Functions

Complicated `awk` programs can often be simplified by defining your own functions. User-defined functions can be called just like built-in ones (see section Function Calls), but it is up to you to define them--to tell `awk` what they should do.

Function Definition Syntax

Definitions of functions can appear anywhere between the rules of an `awk` program. Thus, the general form of an `awk` program is extended to include sequences of rules *and* user-defined function definitions. There is no need in `awk` to put the definition of a function before all uses of the function. This is because `awk` reads the entire program before starting to execute any of it.

The definition of a function named name looks like this:

```
function name(parameter-list)
{
     body-of-function
}
```

name is the name of the function to be defined. A valid function name is like a valid variable name: a sequence of letters, digits and underscores, not starting with a digit. Within a single `awk` program, any particular name can only be used as a variable, array or function.

parameter-list is a list of the function's arguments and local variable names, separated by commas. When the function is called, the argument names are used to hold the argument values given in the call. The local variables are initialized to the empty string. A function cannot have two parameters with the same name.

The body-of-function consists of `awk` statements. It is the most important part of the definition, because it says what the function should actually *do*. The argument names exist to give the body a way to talk about the arguments; local variables, to give the body places to keep temporary values.

Argument names are not distinguished syntactically from local variable names; instead, the number of arguments supplied when the function is called determines how many argument variables there are. Thus, if three argument values are given, the first three names in parameter-list are arguments, and the rest are local variables.

It follows that if the number of arguments is not the same in all calls to the function, some of the names in parameter-list may be arguments on some occasions and local variables on others. Another way to think of this is that omitted arguments default to the null string.

Usually when you write a function you know how many names you intend to use for arguments and how

many you intend to use as local variables. It is conventional to place some extra space between the arguments and the local variables, to document how your function is supposed to be used.

During execution of the function body, the arguments and local variable values hide or shadow any variables of the same names used in the rest of the program. The shadowed variables are not accessible in the function definition, because there is no way to name them while their names have been taken away for the local variables. All other variables used in the `awk` program can be referenced or set normally in the function's body.

The arguments and local variables last only as long as the function body is executing. Once the body finishes, you can once again access the variables that were shadowed while the function was running.

The function body can contain expressions which call functions. They can even call this function, either directly or by way of another function. When this happens, we say the function is recursive.

In many `awk` implementations, including `gawk`, the keyword `function` may be abbreviated `func`. However, POSIX only specifies the use of the keyword `function`. This actually has some practical implications. If `gawk` is in POSIX-compatibility mode (see section Command Line Options), then the following statement will *not* define a function:

```
func foo() { a = sqrt($1) ; print a }
```

Instead it defines a rule that, for each record, concatenates the value of the variable `func' with the return value of the function `foo'. If the resulting string is non-null, the action is executed. This is probably not what was desired. (`awk` accepts this input as syntactically valid, since functions may be used before they are defined in `awk` programs.)

To ensure that your `awk` programs are portable, always use the keyword `function` when defining a function.

Function Definition Examples

Here is an example of a user-defined function, called `myprint`, that takes a number and prints it in a specific format.

```
function myprint(num)
{
     printf "%6.3g\n", num
}
```

To illustrate, here is an `awk` rule which uses our `myprint` function:

```
$3 > 0     { myprint($3) }
```

This program prints, in our special format, all the third fields that contain a positive number in our input. Therefore, when given:


```
 1.2   3.4    5.6   7.8
 9.10 11.12 -13.14 15.16
17.18 19.20  21.22 23.24
```

this program, using our function to format the results, prints:

```
   5.6
  21.2
```

This function deletes all the elements in an array.

```
function delarray(a,    i)
{
    for (i in a)
       delete a[i]
}
```

When working with arrays, it is often necessary to delete all the elements in an array and start over with a new list of elements (see section The `delete` Statement). Instead of having to repeat this loop everywhere in your program that you need to clear out an array, your program can just call `delarray`.

Here is an example of a recursive function. It takes a string as an input parameter, and returns the string in backwards order.

```
function rev(str, start)
{
    if (start == 0)
        return ""

    return (substr(str, start, 1) rev(str, start - 1))
}
```

If this function is in a file named `` `rev.awk' ``, we can test it this way:

```
$ echo "Don't Panic!" |
> gawk --source '{ print rev($0, length($0)) }' -f rev.awk
-| !cinaP t'noD
```

Here is an example that uses the built-in function `strftime`. (See section Functions for Dealing with Time Stamps, for more information on `strftime`.) The C `ctime` function takes a timestamp and returns it in a string, formatted in a well known fashion. Here is an `awk` version:

```
# ctime.awk
#
# awk version of C ctime(3) function
```


```
function ctime(ts,    format)
{
    format = "%a %b %d %H:%M:%S %Z %Y"
    if (ts == 0)
        ts = systime()       # use current time as default
    return strftime(format, ts)
}
```

Calling User-defined Functions

Calling a function means causing the function to run and do its job. A function call is an expression, and its value is the value returned by the function.

A function call consists of the function name followed by the arguments in parentheses. What you write in the call for the arguments are `awk` expressions; each time the call is executed, these expressions are evaluated, and the values are the actual arguments. For example, here is a call to `foo` with three arguments (the first being a string concatenation):

```
foo(x y, "lose", 4 * z)
```

Caution: whitespace characters (spaces and tabs) are not allowed between the function name and the open-parenthesis of the argument list. If you write whitespace by mistake, `awk` might think that you mean to concatenate a variable with an expression in parentheses. However, it notices that you used a function name and not a variable name, and reports an error.

When a function is called, it is given a *copy* of the values of its arguments. This is known as call by value. The caller may use a variable as the expression for the argument, but the called function does not know this: it only knows what value the argument had. For example, if you write this code:

```
foo = "bar"
z = myfunc(foo)
```

then you should not think of the argument to `myfunc` as being "the variable `foo`." Instead, think of the argument as the string value, `"bar"`.

If the function `myfunc` alters the values of its local variables, this has no effect on any other variables. Thus, if `myfunc` does this:

```
function myfunc(str)
{
  print str
  str = "zzz"
  print str
}
```

to change its first argument variable `str`, this *does not* change the value of `foo` in the caller. The role of

`foo` in calling `myfunc` ended when its value, `"bar"`, was computed. If `str` also exists outside of `myfunc`, the function body cannot alter this outer value, because it is shadowed during the execution of `myfunc` and cannot be seen or changed from there.

However, when arrays are the parameters to functions, they are *not* copied. Instead, the array itself is made available for direct manipulation by the function. This is usually called call by reference. Changes made to an array parameter inside the body of a function *are* visible outside that function. *This can be very dangerous if you do not watch what you are doing.* For example:

```
function changeit(array, ind, nvalue)
{
     array[ind] = nvalue
}

BEGIN {
    a[1] = 1; a[2] = 2; a[3] = 3
    changeit(a, 2, "two")
    printf "a[1] = %s, a[2] = %s, a[3] = %s\n",
            a[1], a[2], a[3]
}
```

This program prints `a[1] = 1, a[2] = two, a[3] = 3', because `changeit` stores `"two"` in the second element of `a`.

Some `awk` implementations allow you to call a function that has not been defined, and only report a problem at run-time when the program actually tries to call the function. For example:

```
BEGIN {
    if (0)
        foo()
    else
        bar()
}
function bar() { ... }
# note that `foo' is not defined
```

Since the `if' statement will never be true, it is not really a problem that `foo` has not been defined. Usually though, it is a problem if a program calls an undefined function.

If `--lint' has been specified (see section Command Line Options), `gawk` will report about calls to undefined functions.

The `return` Statement

The body of a user-defined function can contain a `return` statement. This statement returns control to the rest of the `awk` program. It can also be used to return a value for use in the rest of the `awk` program. It looks like this:

```
return [expression]
```

The expression part is optional. If it is omitted, then the returned value is undefined and, therefore, unpredictable.

A `return` statement with no value expression is assumed at the end of every function definition. So if control reaches the end of the function body, then the function returns an unpredictable value. `awk` will *not* warn you if you use the return value of such a function.

Sometimes, you want to write a function for what it does, not for what it returns. Such a function corresponds to a `void` function in C or to a `procedure` in Pascal. Thus, it may be appropriate to not return any value; you should simply bear in mind that if you use the return value of such a function, you do so at your own risk.

Here is an example of a user-defined function that returns a value for the largest number among the elements of an array:

```
function maxelt(vec,   i, ret)
{
     for (i in vec) {
          if (ret == "" || vec[i] > ret)
               ret = vec[i]
     }
     return ret
}
```

You call `maxelt` with one argument, which is an array name. The local variables `i` and `ret` are not intended to be arguments; while there is nothing to stop you from passing two or three arguments to `maxelt`, the results would be strange. The extra space before `i` in the function parameter list indicates that `i` and `ret` are not supposed to be arguments. This is a convention that you should follow when you define functions.

Here is a program that uses our `maxelt` function. It loads an array, calls `maxelt`, and then reports the maximum number in that array:

```
awk '
function maxelt(vec,   i, ret)
{
     for (i in vec) {
          if (ret == "" || vec[i] > ret)
```


```
                ret = vec[i]
        }
        return ret
}

# Load all fields of each record into nums.
{
        for(i = 1; i <= NF; i++)
                nums[NR, i] = $i
}

END {
        print maxelt(nums)
}'
```

Given the following input:

```
 1 5 23 8 16
44 3 5 2 8 26
256 291 1396 2962 100
-6 467 998 1101
99385 11 0 225
```

our program tells us (predictably) that `99385` is the largest number in our array.

Go to the previous, next section.

Go to the previous, next section.

Running awk

There are two ways to run `awk`: with an explicit program, or with one or more program files. Here are templates for both of them; items enclosed in `[...]' in these templates are optional.

Besides traditional one-letter POSIX-style options, `gawk` also supports GNU long options.

```
awk [options] -f progfile [--] file ...
awk [options] [--] 'program' file ...
```

It is possible to invoke `awk` with an empty program:

```
$ awk " datafile1 datafile2
```

Doing so makes little sense though; `awk` will simply exit silently when given an empty program (d.c.). If `--lint' has been specified on the command line, `gawk` will issue a warning that the program is empty.

Command Line Options

Options begin with a dash, and consist of a single character. GNU style long options consist of two dashes and a keyword. The keyword can be abbreviated, as long the abbreviation allows the option to be uniquely identified. If the option takes an argument, then the keyword is either immediately followed by an equals sign (`=') and the argument's value, or the keyword and the argument's value are separated by whitespace. For brevity, the discussion below only refers to the traditional short options; however the long and short options are interchangeable in all contexts.

Each long option for `gawk` has a corresponding POSIX-style option. The options and their meanings are as follows:

`-F fs`

`--field-separator fs`

> Sets the `FS` variable to fs (see section Specifying How Fields are Separated).

`-f source-file`

`--file source-file`

> Indicates that the `awk` program is to be found in source-file instead of in the first non-option argument.

`-v var=val`

`--assign var=val`

> Sets the variable var to the value val **before** execution of the program begins. Such variable values are available inside the `BEGIN` rule (see section Other Command Line Arguments).

The `-v' option can only set one variable, but you can use it more than once, setting another variable each time, like this: `awk -v foo=1 -v bar=2 ...'.

`-mf=NNN`

`-mr=NNN`

Set various memory limits to the value NNN. The `f' flag sets the maximum number of fields, and the `r' flag sets the maximum record size. These two flags and the `-m' option are from the Bell Labs research version of Unix `awk`. They are provided for compatibility, but otherwise ignored by `gawk`, since `gawk` has no predefined limits.

`-W gawk-opt`

Following the POSIX standard, options that are implementation specific are supplied as arguments to the `-W' option. With `gawk`, these arguments may be separated by commas, or quoted and separated by whitespace. Case is ignored when processing these options. These options also have corresponding GNU style long options. See below.

`--`

Signals the end of the command line options. The following arguments are not treated as options even if they begin with `-'. This interpretation of `--' follows the POSIX argument parsing conventions.

This is useful if you have file names that start with `-', or in shell scripts, if you have file names that will be specified by the user which could start with `-'.

The following `gawk`-specific options are available:

`-W traditional`

`-W compat`

`--traditional`

`--compat`

Specifies compatibility mode, in which the GNU extensions to the `awk` language are disabled, so that `gawk` behaves just like the Bell Labs research version of Unix `awk`. `--traditional' is the preferred form of this option. See section Extensions in `gawk` Not in POSIX `awk`, which summarizes the extensions. Also see section Downward Compatibility and Debugging.

`-W copyleft`

`-W copyright`

`--copyleft`

`--copyright`

Print the short version of the General Public License. This option may disappear in a future version of `gawk`.

`-W help`

`-W usage`

`--help`

`--usage`

Print a "usage" message summarizing the short and long style options that `gawk` accepts, and exit.

`-W lint`

`--lint`

Warn about constructs that are dubious or non-portable to other `awk` implementations. Some warnings are issued when `gawk` first reads your program. Others are issued at run-time, as your program executes.

`-W lint-old`

`--lint-old`

Warn about constructs that are not available in the original Version 7 Unix version of `awk` (see section Major Changes between V7 and SVR3.1).

`-W posix`

`--posix`

Operate in strict POSIX mode. This disables all `gawk` extensions (just like `--traditional'), and adds the following additional restrictions:

`\x` escape sequences are not recognized (see section Escape Sequences).

The synonym `func` for the keyword `function` is not recognized (see section Function Definition Syntax).

The operators `**' and `**=' cannot be used in place of `^' and `^=' (see section Arithmetic Operators, and also see section Assignment Expressions).

Specifying `-Ft' on the command line does not set the value of `FS` to be a single tab character (see section Specifying How Fields are Separated).

The `fflush` built-in function is not supported (see section Built-in Functions for Input/Output).

If you supply both `--traditional' and `--posix' on the command line, `--posix' will take precedence. `gawk` will also issue a warning if both options are supplied.

`-W re-interval`

`--re-interval`

Allow interval expressions (see section Regular Expression Operators), in regexps. Because interval expressions were traditionally not available in `awk`, `gawk` does not provide them by default. This prevents old `awk` programs from breaking.

`-W source program-text`

`--source program-text`

Program source code is taken from the program-text. This option allows you to mix source code in files with source code that you enter on the command line. This is particularly useful when you have library functions that you wish to use from your command line programs (see section The `AWKPATH` Environment Variable).

`-W version`

`--version`

> Prints version information for this particular copy of `gawk`. This allows you to determine if your copy of `gawk` is up to date with respect to whatever the Free Software Foundation is currently distributing. It is also useful for bug reports (see section Reporting Problems and Bugs).

Any other options are flagged as invalid with a warning message, but are otherwise ignored.

In compatibility mode, as a special case, if the value of fs supplied to the `-F' option is `t', then `FS` is set to the tab character (`"\t"`). This is only true for `--traditional', and not for `--posix' (see section Specifying How Fields are Separated).

The `-f' option may be used more than once on the command line. If it is, `awk` reads its program source from all of the named files, as if they had been concatenated together into one big file. This is useful for creating libraries of `awk` functions. Useful functions can be written once, and then retrieved from a standard place, instead of having to be included into each individual program.

You can type in a program at the terminal and still use library functions, by specifying `-f /dev/tty'. `awk` will read a file from the terminal to use as part of the `awk` program. After typing your program, type Control-d (the end-of-file character) to terminate it. (You may also use `-f -' to read program source from the standard input, but then you will not be able to also use the standard input as a source of data.)

Because it is clumsy using the standard `awk` mechanisms to mix source file and command line `awk` programs, `gawk` provides the `--source' option. This does not require you to pre-empt the standard input for your source code, and allows you to easily mix command line and library source code (see section The `AWKPATH` Environment Variable).

If no `-f' or `--source' option is specified, then `gawk` will use the first non-option command line argument as the text of the program source code.

If the environment variable `POSIXLY_CORRECT` exists, then `gawk` will behave in strict POSIX mode, exactly as if you had supplied the `--posix' command line option. Many GNU programs look for this environment variable to turn on strict POSIX mode. If you supply `--lint' on the command line, and `gawk` turns on POSIX mode because of `POSIXLY_CORRECT`, then it will print a warning message indicating that POSIX mode is in effect.

You would typically set this variable in your shell's startup file. For a Bourne compatible shell (such as Bash), you would add these lines to the `.profile'` file in your home directory.

```
POSIXLY_CORRECT=true
export POSIXLY_CORRECT
```

For a `csh` compatible shell,(14) you would add this line to the `.login'` file in your home directory.

```
setenv POSIXLY_CORRECT true
```

Other Command Line Arguments

Any additional arguments on the command line are normally treated as input files to be processed in the order specified. However, an argument that has the form `var=value`, assigns the value value to the variable var---it does not specify a file at all.

All these arguments are made available to your `awk` program in the `ARGV` array (see section Built-in Variables). Command line options and the program text (if present) are omitted from `ARGV`. All other arguments, including variable assignments, are included. As each element of `ARGV` is processed, `gawk` sets the variable `ARGIND` to the index in `ARGV` of the current element.

The distinction between file name arguments and variable-assignment arguments is made when `awk` is about to open the next input file. At that point in execution, it checks the "file name" to see whether it is really a variable assignment; if so, `awk` sets the variable instead of reading a file.

Therefore, the variables actually receive the given values after all previously specified files have been read. In particular, the values of variables assigned in this fashion are *not* available inside a `BEGIN` rule (see section The `BEGIN` and `END` Special Patterns), since such rules are run before `awk` begins scanning the argument list.

The variable values given on the command line are processed for escape sequences (d.c.) (see section Escape Sequences).

In some earlier implementations of `awk`, when a variable assignment occurred before any file names, the assignment would happen *before* the `BEGIN` rule was executed. `awk`'s behavior was thus inconsistent; some command line assignments were available inside the `BEGIN` rule, while others were not. However, some applications came to depend upon this "feature." When `awk` was changed to be more consistent, the `-v' option was added to accommodate applications that depended upon the old behavior.

The variable assignment feature is most useful for assigning to variables such as `RS`, `OFS`, and `ORS`, which control input and output formats, before scanning the data files. It is also useful for controlling state if multiple passes are needed over a data file. For example:

```
awk 'pass == 1  { pass 1 stuff }
     pass == 2  { pass 2 stuff }' pass=1 mydata pass=2 mydata
```

Given the variable assignment feature, the `-F' option for setting the value of `FS` is not strictly necessary. It remains for historical compatibility.

The `AWKPATH` Environment Variable

The previous section described how `awk` program files can be named on the command line with the `-f' option. In most `awk` implementations, you must supply a precise path name for each program file, unless the file is in the current directory.

But in `gawk`, if the file name supplied to the `-f' option does not contain a `/', then `gawk` searches a list

of directories (called the search path), one by one, looking for a file with the specified name.

The search path is a string consisting of directory names separated by colons. `gawk` gets its search path from the `AWKPATH` environment variable. If that variable does not exist, `gawk` uses a default path, which is `.:/usr/local/share/awk'.(15) (Programs written for use by system administrators should use an `AWKPATH` variable that does not include the current directory, `.'.)

The search path feature is particularly useful for building up libraries of useful `awk` functions. The library files can be placed in a standard directory that is in the default path, and then specified on the command line with a short file name. Otherwise, the full file name would have to be typed for each file.

By using both the `--source' and `-f' options, your command line `awk` programs can use facilities in `awk` library files. See section A Library of awk Functions.

Path searching is not done if `gawk` is in compatibility mode. This is true for both `--traditional' and `--posix'. See section Command Line Options.

Note: if you want files in the current directory to be found, you must include the current directory in the path, either by including `.' explicitly in the path, or by writing a null entry in the path. (A null entry is indicated by starting or ending the path with a colon, or by placing two colons next to each other (`::').) If the current directory is not included in the path, then files cannot be found in the current directory. This path search mechanism is identical to the shell's.

Starting with version 3.0, if `AWKPATH` is not defined in the environment, `gawk` will place its default search path into `ENVIRON["AWKPATH"]`. This makes it easy to determine the actual search path `gawk` will use.

Obsolete Options and/or Features

This section describes features and/or command line options from previous releases of `gawk` that are either not available in the current version, or that are still supported but deprecated (meaning that they will *not* be in the next release).

For version 3.0 of `gawk`, there are no command line options or other deprecated features from the previous version of `gawk`. This section is thus essentially a place holder, in case some option becomes obsolete in a future version of `gawk`.

Undocumented Options and Features

This section intentionally left blank.

Known Bugs in gawk

- The `-F' option for changing the value of `FS` (see section Command Line Options) is not necessary given the command line variable assignment feature; it remains only for backwards compatibility.

- If your system actually has support for `/dev/fd' and the associated `/dev/stdin', `/dev/stdout', and `/dev/stderr' files, you may get different output from gawk than you would get on a system without those files. When gawk interprets these files internally, it synchronizes output to the standard output with output to `/dev/stdout', while on a system with those files, the output is actually to different open files (see section Special File Names in gawk).
- Syntactically invalid single character programs tend to overflow the parse stack, generating a rather unhelpful message. Such programs are surprisingly difficult to diagnose in the completely general case, and the effort to do so really is not worth it.
- The word "GNU" is incorrectly capitalized in at least one file in the source code.

Go to the previous, next section.

Go to the previous, next section.

A Library of awk Functions

This chapter presents a library of useful `awk` functions. The sample programs presented later (see section Practical `awk` Programs) use these functions. The functions are presented here in a progression from simple to complex.

section Extracting Programs from Texinfo Source Files, presents a program that you can use to extract the source code for these example library functions and programs from the Texinfo source for this book. (This has already been done as part of the `gawk` distribution.)

If you have written one or more useful, general purpose `awk` functions, and would like to contribute them for a subsequent edition of this book, please contact the author. See section Reporting Problems and Bugs, for information on doing this. Don't just send code, as you will be required to either place your code in the public domain, publish it under the GPL (see section GNU GENERAL PUBLIC LICENSE), or assign the copyright in it to the Free Software Foundation.

Simulating gawk-specific Features

The programs in this chapter and in section Practical `awk` Programs, freely use features that are specific to `gawk`. This section briefly discusses how you can rewrite these programs for different implementations of `awk`.

Diagnostic error messages are sent to `` `/dev/stderr' ``. Use `` `| "cat 1>&2"' `` instead of `` `> "/dev/stderr"' ``, if your system does not have a `` `/dev/stderr' ``, or if you cannot use `gawk`.

A number of programs use `nextfile` (see section The `nextfile` Statement), to skip any remaining input in the input file. section Implementing `nextfile` as a Function, shows you how to write a function that will do the same thing.

Finally, some of the programs choose to ignore upper-case and lower-case distinctions in their input. They do this by assigning one to `IGNORECASE`. You can achieve the same effect by adding the following rule to the beginning of the program:

```
# ignore case
{ $0 = tolower($0) }
```

Also, verify that all regexp and string constants used in comparisons only use lower-case letters.

Implementing `nextfile` as a Function

The `nextfile` statement presented in section The `nextfile` Statement, is a `gawk`-specific extension. It is not available in other implementations of `awk`. This section shows two versions of a `nextfile` function that you can use to simulate `gawk`'s `nextfile` statement if you cannot use `gawk`.

Here is a first attempt at writing a `nextfile` function.

```
# nextfile -- skip remaining records in current file

# this should be read in before the "main" awk program

function nextfile()    { _abandon_ = FILENAME; next }

_abandon_ == FILENAME  { next }
```

This file should be included before the main program, because it supplies a rule that must be executed first. This rule compares the current data file's name (which is always in the `FILENAME` variable) to a private variable named `_abandon_`. If the file name matches, then the action part of the rule executes a `next` statement, to go on to the next record. (The use of `_' in the variable name is a convention. It is discussed more fully in section Naming Library Function Global Variables.)

The use of the `next` statement effectively creates a loop that reads all the records from the current data file. Eventually, the end of the file is reached, and a new data file is opened, changing the value of `FILENAME`. Once this happens, the comparison of `_abandon_` to `FILENAME` fails, and execution continues with the first rule of the "real" program.

The `nextfile` function itself simply sets the value of `_abandon_` and then executes a `next` statement to start the loop going.(16)

This initial version has a subtle problem. What happens if the same data file is listed *twice* on the command line, one right after the other, or even with just a variable assignment between the two occurrences of the file name?

In such a case, this code will skip right through the file, a second time, even though it should stop when it gets to the end of the first occurrence. Here is a second version of `nextfile` that remedies this problem.

```
# nextfile -- skip remaining records in current file
# correctly handle successive occurrences of the same file
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# May, 1993

# this should be read in before the "main" awk program

function nextfile()   { _abandon_ = FILENAME; next }
```


```
_abandon_ == FILENAME {
      if (FNR == 1)
          _abandon_ = ""
      else
          next
}
```

The `nextfile` function has not changed. It sets `_abandon_` equal to the current file name and then executes a `next` satement. The `next` statement reads the next record and increments `FNR`, so `FNR` is guaranteed to have a value of at least two. However, if `nextfile` is called for the last record in the file, then `awk` will close the current data file and move on to the next one. Upon doing so, `FILENAME` will be set to the name of the new file, and `FNR` will be reset to one. If this next file is the same as the previous one, `_abandon_` will still be equal to `FILENAME`. However, `FNR` will be equal to one, telling us that this is a new occurrence of the file, and not the one we were reading when the `nextfile` function was executed. In that case, `_abandon_` is reset to the empty string, so that further executions of this rule will fail (until the next time that `nextfile` is called).

If `FNR` is not one, then we are still in the original data file, and the program executes a `next` statement to skip through it.

An important question to ask at this point is: "Given that the functionality of `nextfile` can be provided with a library file, why is it built into `gawk`?" This is an important question. Adding features for little reason leads to larger, slower programs that are harder to maintain.

The answer is that building `nextfile` into `gawk` provides significant gains in efficiency. If the `nextfile` function is executed at the beginning of a large data file, `awk` still has to scan the entire file, splitting it up into records, just to skip over it. The built-in `nextfile` can simply close the file immediately and proceed to the next one, saving a lot of time. This is particularly important in `awk`, since `awk` programs are generally I/O bound (i.e. they spend most of their time doing input and output, instead of performing computations).

Assertions

When writing large programs, it is often useful to be able to know that a condition or set of conditions is true. Before proceeding with a particular computation, you make a statement about what you believe to be the case. Such a statement is known as an "assertion." The C language provides an `<assert.h>` header file and corresponding `assert` macro that the programmer can use to make assertions. If an assertion fails, the `assert` macro arranges to print a diagnostic message describing the condition that should have been true but was not, and then it kills the program. In C, using `assert` looks this:

```
#include <assert.h>

int myfunc(int a, double b)
{
     assert(a <= 5 && b >= 17);
```


```
     ...
}
```

If the assertion failed, the program would print a message similar to this:

```
prog.c:5: assertion failed: a <= 5 && b >= 17
```

The ANSI C language makes it possible to turn the condition into a string for use in printing the diagnostic message. This is not possible in `awk`, so this `assert` function also requires a string version of the condition that is being tested.

```
# assert -- assert that a condition is true. Otherwise exit.
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# May, 1993

function assert(condition, string)
{
    if (! condition) {
        printf("%s:%d: assertion failed: %s\n",
            FILENAME, FNR, string) > "/dev/stderr"
        _assert_exit = 1
        exit 1
    }
}

END {
    if (_assert_exit)
        exit 1
}
```

The `assert` function tests the `condition` parameter. If it is false, it prints a message to standard error, using the `string` parameter to describe the failed condition. It then sets the variable `_assert_exit` to one, and executes the `exit` statement. The `exit` statement jumps to the `END` rule. If the `END` rules finds `_assert_exit` to be true, then it exits immediately.

The purpose of the `END` rule with its test is to keep any other `END` rules from running. When an assertion fails, the program should exit immediately. If no assertions fail, then `_assert_exit` will still be false when the `END` rule is run normally, and the rest of the program's `END` rules will execute. For all of this to work correctly, `` `assert.awk' `` must be the first source file read by `awk`.

You would use this function in your programs this way:

```
function myfunc(a, b)
{
     assert(a <= 5 && b >= 17, "a <= 5 && b >= 17")
     ...
}
```

If the assertion failed, you would see a message like this:

```
mydata:1357: assertion failed: a <= 5 && b >= 17
```

There is a problem with this version of `assert`, that it may not be possible to work around. An `END` rule is automatically added to the program calling `assert`. Normally, if a program consists of just a `BEGIN` rule, the input files and/or standard input are not read. However, now that the program has an `END` rule, `awk` will attempt to read the input data files, or standard input (see section Startup and Cleanup Actions), most likely causing the program to hang, waiting for input.

Just a note on programming style. You may have noticed that the `END` rule uses backslash continuation, with the open brace on a line by itself. This is so that it more closely resembles the way functions are written. Many of the examples in this chapter and the next one use this style. You can decide for yourself if you like writing your `BEGIN` and `END` rules this way, or not.

Translating Between Characters and Numbers

One commercial implementation of `awk` supplies a built-in function, `ord`, which takes a character and returns the numeric value for that character in the machine's character set. If the string passed to `ord` has more than one character, only the first one is used.

The inverse of this function is `chr` (from the function of the same name in Pascal), which takes a number and returns the corresponding character.

Both functions can be written very nicely in `awk`; there is no real reason to build them into the `awk` interpreter.

```
# ord.awk -- do ord and chr
#
# Global identifiers:
#    _ord_:        numerical values indexed by characters
#    _ord_init:    function to initialize _ord_
#
# Arnold Robbins
# arnold@gnu.ai.mit.edu
# Public Domain
# 16 January, 1992
# 20 July, 1992, revised

BEGIN    { _ord_init() }

function _ord_init(    low, high, i, t)
{
    low = sprintf("%c", 7) # BEL is ascii 7
    if (low == "\a") {    # regular ascii
```


```
        low = 0
        high = 127
    } else if (sprintf("%c", 128 + 7) == "\a") {
        # ascii, mark parity
        low = 128
        high = 255
    } else {        # ebcdic(!)
        low = 0
        high = 255
    }

    for (i = low; i <= high; i++) {
        t = sprintf("%c", i)
        _ord_[t] = i
    }
}
```

Some explanation of the numbers used by `chr` is worthwhile. The most prominent character set in use today is ASCII. Although an eight-bit byte can hold 256 distinct values (from zero to 255), ASCII only defines characters that use the values from zero to 127.(17) At least one computer manufacturer that we know of uses ASCII, but with mark parity, meaning that the leftmost bit in the byte is always one. What this means is that on those systems, characters have numeric values from 128 to 255. Finally, large mainframe systems use the EBCDIC character set, which uses all 256 values. While there are other character sets in use on some older systems, they are not really worth worrying about.

```
function ord(str,    c)
{
    # only first character is of interest
    c = substr(str, 1, 1)
    return _ord_[c]
}

function chr(c)
{
    # force c to be numeric by adding 0
    return sprintf("%c", c + 0)
}

#### test code ####
# BEGIN    \
# {
#    for (;;) {
#        printf("enter a character: ")
#        if (getline var <= 0)
#            break
```


```
#         printf("ord(%s) = %d\n", var, ord(var))
#     }
# }
```

An obvious improvement to these functions would be to move the code for the `_ord_init` function into the body of the `BEGIN` rule. It was written this way initially for ease of development.

There is a "test program" in a `BEGIN` rule, for testing the function. It is commented out for production use.

Merging an Array Into a String

When doing string processing, it is often useful to be able to join all the strings in an array into one long string. The following function, `join`, accomplishes this task. It is used later in several of the application programs (see section Practical `awk` Programs).

Good function design is important; this function needs to be general, but it should also have a reasonable default behavior. It is called with an array and the beginning and ending indices of the elements in the array to be merged. This assumes that the array indices are numeric--a reasonable assumption since the array was likely created with `split` (see section Built-in Functions for String Manipulation).

```
# join.awk -- join an array into a string
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# May 1993

function join(array, start, end, sep,    result, i)
{
    if (sep == "")
       sep = " "
    else if (sep == SUBSEP) # magic value
       sep = ""
    result = array[start]
    for (i = start + 1; i <= end; i++)
        result = result sep array[i]
    return result
}
```

An optional additional argument is the separator to use when joining the strings back together. If the caller supplies a non-empty value, `join` uses it. If it is not supplied, it will have a null value. In this case, `join` uses a single blank as a default separator for the strings. If the value is equal to `SUBSEP`, then `join` joins the strings with no separator between them. `SUBSEP` serves as a "magic" value to indicate that there should be no separation between the component strings.

It would be nice if `awk` had an assignment operator for concatenation. The lack of an explicit operator for concatenation makes string operations more difficult than they really need to be.

Turning Dates Into Timestamps

The `systime` function built in to `gawk` returns the current time of day as a timestamp in "seconds since the Epoch." This timestamp can be converted into a printable date of almost infinitely variable format using the built-in `strftime` function. (For more information on `systime` and `strftime`, see section Functions for Dealing with Time Stamps.)

An interesting but difficult problem is to convert a readable representation of a date back into a timestamp. The ANSI C library provides a `mktime` function that does the basic job, converting a canonical representation of a date into a timestamp.

It would appear at first glance that `gawk` would have to supply a `mktime` built-in function that was simply a "hook" to the C language version. In fact though, `mktime` can be implemented entirely in `awk`.

Here is a version of `mktime` for `awk`. It takes a simple representation of the date and time, and converts it into a timestamp.

The code is presented here intermixed with explanatory prose. In section Extracting Programs from Texinfo Source Files, you will see how the Texinfo source file for this book can be processed to extract the code into a single source file.

The program begins with a descriptive comment and a `BEGIN` rule that initializes a table `_tm_months`. This table is a two-dimensional array that has the lengths of the months. The first index is zero for regular years, and one for leap years. The values are the same for all the months in both kinds of years, except for February; thus the use of multiple assignment.

```
# mktime.awk -- convert a canonical date representation
#               into a timestamp
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# May 1993

BEGIN    \
{
    # Initialize table of month lengths
    _tm_months[0,1] = _tm_months[1,1] = 31
    _tm_months[0,2] = 28; _tm_months[1,2] = 29
    _tm_months[0,3] = _tm_months[1,3] = 31
    _tm_months[0,4] = _tm_months[1,4] = 30
    _tm_months[0,5] = _tm_months[1,5] = 31
    _tm_months[0,6] = _tm_months[1,6] = 30
    _tm_months[0,7] = _tm_months[1,7] = 31
    _tm_months[0,8] = _tm_months[1,8] = 31
    _tm_months[0,9] = _tm_months[1,9] = 30
    _tm_months[0,10] = _tm_months[1,10] = 31
    _tm_months[0,11] = _tm_months[1,11] = 30
    _tm_months[0,12] = _tm_months[1,12] = 31
```


```
}
```

The benefit of merging multiple `BEGIN` rules (see section The `BEGIN` and `END` Special Patterns) is particularly clear when writing library files. Functions in library files can cleanly initialize their own private data and also provide clean-up actions in private `END` rules.

The next function is a simple one that computes whether a given year is or is not a leap year. If a year is evenly divisible by four, but not evenly divisible by 100, or if it is evenly divisible by 400, then it is a leap year. Thus, 1904 was a leap year, 1900 was not, but 2000 will be.

```
# decide if a year is a leap year
function _tm_isleap(year,    ret)
{
    ret = (year % 4 == 0 && year % 100 != 0) ||
            (year % 400 == 0)

    return ret
}
```

This function is only used a few times in this file, and its computation could have been written in-line (at the point where it's used). Making it a separate function made the original development easier, and also avoids the possibility of typing errors when duplicating the code in multiple places.

The next function is more interesting. It does most of the work of generating a timestamp, which is converting a date and time into some number of seconds since the Epoch. The caller passes an array (rather imaginatively named `a`) containing six values: the year including century, the month as a number between one and 12, the day of the month, the hour as a number between zero and 23, the minute in the hour, and the seconds within the minute.

The function uses several local variables to precompute the number of seconds in an hour, seconds in a day, and seconds in a year. Often, similar C code simply writes out the expression in-line, expecting the compiler to do constant folding. E.g., most C compilers would turn `60 * 60' into `3600' at compile time, instead of recomputing it every time at run time. Precomputing these values makes the function more efficient.

```
# convert a date into seconds
function _tm_addup(a,    total, yearsecs, daysecs,
                         hoursecs, i, j)
{
    hoursecs = 60 * 60
    daysecs = 24 * hoursecs
    yearsecs = 365 * daysecs

    total = (a[1] - 1970) * yearsecs

    # extra day for leap years
    for (i = 1970; i < a[1]; i++)
```


```
        if (_tm_isleap(i))
            total += daysecs

    j = _tm_isleap(a[1])
    for (i = 1; i < a[2]; i++)
        total += _tm_months[j, i] * daysecs

    total += (a[3] - 1) * daysecs
    total += a[4] * hoursecs
    total += a[5] * 60
    total += a[6]

    return total
}
```

The function starts with a first approximation of all the seconds between Midnight, January 1, 1970,(18) and the beginning of the current year. It then goes through all those years, and for every leap year, adds an additional day's worth of seconds.

The variable `j` holds either one or zero, if the current year is or is not a leap year. For every month in the current year prior to the current month, it adds the number of seconds in the month, using the appropriate entry in the `_tm_months` array.

Finally, it adds in the seconds for the number of days prior to the current day, and the number of hours, minutes, and seconds in the current day.

The result is a count of seconds since January 1, 1970. This value is not yet what is needed though. The reason why is described shortly.

The main `mktime` function takes a single character string argument. This string is a representation of a date and time in a "canonical" (fixed) form. This string should be `"year month day hour minute second"`.

```
# mktime -- convert a date into seconds,
#           compensate for time zone

function mktime(str,    res1, res2, a, b, i, j, t, diff)
{
    i = split(str, a, " ")    # don't rely on FS

    if (i != 6)
        return -1

    # force numeric
    for (j in a)
        a[j] += 0
```


```
    # validate
    if (a[1] < 1970 ||
        a[2] < 1 || a[2] > 12 ||
        a[3] < 1 || a[3] > 31 ||
        a[4] < 0 || a[4] > 23 ||
        a[5] < 0 || a[5] > 59 ||
        a[6] < 0 || a[6] > 61 )
            return -1

    res1 = _tm_addup(a)
    t = strftime("%Y %m %d %H %M %S", res1)

    if (_tm_debug)
        printf("(%s) -> (%s)\n", str, t) > "/dev/stderr"

    split(t, b, " ")
    res2 = _tm_addup(b)

    diff = res1 - res2

    if (_tm_debug)
        printf("diff = %d seconds\n", diff) > "/dev/stderr"

    res1 += diff

    return res1
}
```

The function first splits the string into an array, using spaces and tabs as separators. If there are not six elements in the array, it returns an error, signaled as the value -1. Next, it forces each element of the array to be numeric, by adding zero to it. The following `if' statement then makes sure that each element is within an allowable range. (This checking could be extended further, e.g., to make sure that the day of the month is within the correct range for the particular month supplied.) All of this is essentially preliminary set-up and error checking.

Recall that `_tm_addup` generated a value in seconds since Midnight, January 1, 1970. This value is not directly usable as the result we want, *since the calculation does not account for the local timezone*. In other words, the value represents the count in seconds since the Epoch, but only for UTC (Universal Coordinated Time). If the local timezone is east or west of UTC, then some number of hours should be either added to, or subtracted from the resulting timestamp.

For example, 6:23 p.m. in Atlanta, Georgia (USA), is normally five hours west of (behind) UTC. It is only four hours behind UTC if daylight savings time is in effect. If you are calling `mktime` in Atlanta, with the argument `"1993 5 23 18 23 12"`, the result from `_tm_addup` will be for 6:23 p.m. UTC, which is only 2:23 p.m. in Atlanta. It is necessary to add another four hours worth of seconds to the result.

How can `mktime` determine how far away it is from UTC? This is surprisingly easy. The returned timestamp represents the time passed to `mktime` *as UTC*. This timestamp can be fed back to `strftime`, which will format it as a *local* time; i.e. as if it already had the UTC difference added in to it. This is done by giving `"%Y %m %d %H %M %S"` to `strftime` as the format argument. It returns the computed timestamp in the original string format. The result represents a time that accounts for the UTC difference. When the new time is converted back to a timestamp, the difference between the two timestamps is the difference (in seconds) between the local timezone and UTC. This difference is then added back to the original result. An example demonstrating this is presented below.

Finally, there is a "main" program for testing the function.

```
BEGIN  {
    if (_tm_test) {
        printf "Enter date as yyyy mm dd hh mm ss: "
        getline _tm_test_date

        t = mktime(_tm_test_date)
        r = strftime("%Y %m %d %H %M %S", t)
        printf "Got back (%s)\n", r
    }
}
```

The entire program uses two variables that can be set on the command line to control debugging output and to enable the test in the final `BEGIN` rule. Here is the result of a test run. (Note that debugging output is to standard error, and test output is to standard output.)

```
$ gawk -f mktime.awk -v _tm_test=1 -v _tm_debug=1
-| Enter date as yyyy mm dd hh mm ss: 1993 5 23 15 35 10
error--> (1993 5 23 15 35 10) -> (1993 05 23 11 35 10)
error--> diff = 14400 seconds
-| Got back (1993 05 23 15 35 10)
```

The time entered was 3:35 p.m. (15:35 on a 24-hour clock), on May 23, 1993. The first line of debugging output shows the resulting time as UTC--four hours ahead of the local time zone. The second line shows that the difference is 14400 seconds, which is four hours. (The difference is only four hours, since daylight savings time is in effect during May.) The final line of test output shows that the timezone compensation algorithm works; the returned time is the same as the entered time.

This program does not solve the general problem of turning an arbitrary date representation into a timestamp. That problem is very involved. However, the `mktime` function provides a foundation upon which to build. Other software can convert month names into numeric months, and AM/PM times into 24-hour clocks, to generate the "canonical" format that `mktime` requires.

TOC

Managing the Time of Day

The `systime` and `strftime` functions described in section Functions for Dealing with Time Stamps, provide the minimum functionality necessary for dealing with the time of day in human readable form. While `strftime` is extensive, the control formats are not necessarily easy to remember or intuitively obvious when reading a program.

The following function, `gettimeofday`, populates a user-supplied array with pre-formatted time information. It returns a string with the current time formatted in the same way as the `date` utility.

```
# gettimeofday -- get the time of day in a usable format
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain, May 1993
#
# Returns a string in the format of output of date(1)
# Populates the array argument time with individual values:
#    time["second"]       -- seconds (0 - 59)
#    time["minute"]       -- minutes (0 - 59)
#    time["hour"]         -- hours (0 - 23)
#    time["althour"]      -- hours (0 - 12)
#    time["monthday"]     -- day of month (1 - 31)
#    time["month"]        -- month of year (1 - 12)
#    time["monthname"]    -- name of the month
#    time["shortmonth"]   -- short name of the month
#    time["year"]         -- year within century (0 - 99)
#    time["fullyear"]     -- year with century (19xx or 20xx)
#    time["weekday"]      -- day of week (Sunday = 0)
#    time["altweekday"]   -- day of week (Monday = 0)
#    time["weeknum"]      -- week number, Sunday first day
#    time["altweeknum"]   -- week number, Monday first day
#    time["dayname"]      -- name of weekday
#    time["shortdayname"] -- short name of weekday
#    time["yearday"]      -- day of year (0 - 365)
#    time["timezone"]     -- abbreviation of timezone name
#    time["ampm"]         -- AM or PM designation

function gettimeofday(time,    ret, now, i)
{
    # get time once, avoids unnecessary system calls
    now = systime()

    # return date(1)-style output
    ret = strftime("%a %b %d %H:%M:%S %Z %Y", now)

    # clear out target array
    for (i in time)
```


```
        delete time[i]

    # fill in values, force numeric values to be
    # numeric by adding 0
    time["second"]       = strftime("%S", now) + 0
    time["minute"]       = strftime("%M", now) + 0
    time["hour"]         = strftime("%H", now) + 0
    time["althour"]      = strftime("%I", now) + 0
    time["monthday"]     = strftime("%d", now) + 0
    time["month"]        = strftime("%m", now) + 0
    time["monthname"]    = strftime("%B", now)
    time["shortmonth"]   = strftime("%b", now)
    time["year"]         = strftime("%y", now) + 0
    time["fullyear"]     = strftime("%Y", now) + 0
    time["weekday"]      = strftime("%w", now) + 0
    time["altweekday"]   = strftime("%u", now) + 0
    time["dayname"]      = strftime("%A", now)
    time["shortdayname"] = strftime("%a", now)
    time["yearday"]      = strftime("%j", now) + 0
    time["timezone"]     = strftime("%Z", now)
    time["ampm"]         = strftime("%p", now)
    time["weeknum"]      = strftime("%U", now) + 0
    time["altweeknum"]   = strftime("%W", now) + 0

    return ret
}
```

The string indices are easier to use and read than the various formats required by `strftime`. The `alarm` program presented in section An Alarm Clock Program, uses this function.

The `gettimeofday` function is presented above as it was written. A more general design for this function would have allowed the user to supply an optional timestamp value that would have been used instead of the current time.

Noting Data File Boundaries

The `BEGIN` and `END` rules are each executed exactly once, at the beginning and end respectively of your `awk` program (see section The `BEGIN` and `END` Special Patterns). We (the `gawk` authors) once had a user who mistakenly thought that the `BEGIN` rule was executed at the beginning of each data file and the `END` rule was executed at the end of each data file. When informed that this was not the case, the user requested that we add new special patterns to `gawk`, named `BEGIN_FILE` and `END_FILE`, that would have the desired behavior. He even supplied us the code to do so.

However, after a little thought, I came up with the following library program. It arranges to call two user-supplied functions, `beginfile` and `endfile`, at the beginning and end of each data file. Besides solving the problem in only nine(!) lines of code, it does so *portably*; this will work with any

implementation of awk.

```
# transfile.awk
#
# Give the user a hook for filename transitions
#
# The user must supply functions beginfile() and endfile()
# that each take the name of the file being started or
# finished, respectively.
#
# Arnold Robbins, arnold@gnu.ai.mit.edu, January 1992
# Public Domain

FILENAME != _oldfilename \
{
    if (_oldfilename != "")
        endfile(_oldfilename)
    _oldfilename = FILENAME
    beginfile(FILENAME)
}

END   { endfile(FILENAME) }
```

This file must be loaded before the user's "main" program, so that the rule it supplies will be executed first.

This rule relies on awk's FILENAME variable that automatically changes for each new data file. The current file name is saved in a private variable, _oldfilename. If FILENAME does not equal _oldfilename, then a new data file is being processed, and it is necessary to call endfile for the old file. Since endfile should only be called if a file has been processed, the program first checks to make sure that _oldfilename is not the null string. The program then assigns the current file name to _oldfilename, and calls beginfile for the file. Since, like all awk variables, _oldfilename will be initialized to the null string, this rule executes correctly even for the first data file.

The program also supplies an END rule, to do the final processing for the last file. Since this END rule comes before any END rules supplied in the "main" program, endfile will be called first. Once again the value of multiple BEGIN and END rules should be clear.

This version has same problem as the first version of nextfile (see section Implementing nextfile as a Function). If the same data file occurs twice in a row on command line, then endfile and beginfile will not be executed at the end of the first pass and at the beginning of the second pass. This version solves the problem.

```
# ftrans.awk -- handle data file transitions
#
# user supplies beginfile() and endfile() functions
```


```
#
# Arnold Robbins, arnold@gnu.ai.mit.edu. November 1992
# Public Domain

FNR == 1 {
    if (_filename_ != "")
        endfile(_filename_)
    _filename_ = FILENAME
    beginfile(FILENAME)
}

END  { endfile(_filename_) }
```

In section Counting Things, you will see how this library function can be used, and how it simplifies writing the main program.

Processing Command Line Options

Most utilities on POSIX compatible systems take options or "switches" on the command line that can be used to change the way a program behaves. `awk` is an example of such a program (see section Command Line Options). Often, options take arguments, data that the program needs to correctly obey the command line option. For example, `awk`'s `-F' option requires a string to use as the field separator. The first occurrence on the command line of either `--' or a string that does not begin with `-' ends the options.

Most Unix systems provide a C function named `getopt` for processing command line arguments. The programmer provides a string describing the one letter options. If an option requires an argument, it is followed in the string with a colon. `getopt` is also passed the count and values of the command line arguments, and is called in a loop. `getopt` processes the command line arguments for option letters. Each time around the loop, it returns a single character representing the next option letter that it found, or `?' if it found an invalid option. When it returns -1, there are no options left on the command line.

When using `getopt`, options that do not take arguments can be grouped together. Furthermore, options that take arguments require that the argument be present. The argument can immediately follow the option letter, or it can be a separate command line argument.

Given a hypothetical program that takes three command line options, `-a', `-b', and `-c', and `-b' requires an argument, all of the following are valid ways of invoking the program:

```
prog -a -b foo -c data1 data2 data3
prog -ac -bfoo -- data1 data2 data3
prog -acbfoo data1 data2 data3
```

Notice that when the argument is grouped with its option, the rest of the command line argument is considered to be the option's argument. In the above example, `-acbfoo' indicates that all of the `-a', `-b', and `-c' options were supplied, and that `foo' is the argument to the `-b' option.

`getopt` provides four external variables that the programmer can use.

`optind`

The index in the argument value array (`argv`) where the first non-option command line argument can be found.

`optarg`

The string value of the argument to an option.

`opterr`

Usually `getopt` prints an error message when it finds an invalid option. Setting `opterr` to zero disables this feature. (An application might wish to print its own error message.)

`optopt`

The letter representing the command line option. While not usually documented, most versions supply this variable.

The following C fragment shows how `getopt` might process command line arguments for `awk`.

```
int
main(int argc, char *argv[])
{
    ...
    /* print our own message */
    opterr = 0;
    while ((c = getopt(argc, argv, "v:f:F:W:")) != -1) {
        switch (c) {
        case 'f':    /* file */
            ...
            break;
        case 'F':    /* field separator */
            ...
            break;
        case 'v':    /* variable assignment */
            ...
            break;
        case 'W':    /* extension */
            ...
            break;
        case '?':
        default:
            usage();
            break;
        }
    }
    ...
}
```

As a side point, `gawk` actually uses the GNU `getopt_long` function to process both normal and GNU-style long options (see section Command Line Options).

The abstraction provided by `getopt` is very useful, and would be quite handy in `awk` programs as well. Here is an `awk` version of `getopt`. This function highlights one of the greatest weaknesses in `awk`, which is that it is very poor at manipulating single characters. Repeated calls to `substr` are necessary for accessing individual characters (see section Built-in Functions for String Manipulation).

The discussion walks through the code a bit at a time.

```
# getopt -- do C library getopt(3) function in awk
#
# arnold@gnu.ai.mit.edu
# Public domain
#
# Initial version: March, 1991
# Revised: May, 1993

# External variables:
#    Optind -- index of ARGV for first non-option argument
#    Optarg -- string value of argument to current option
#    Opterr -- if non-zero, print our own diagnostic
#    Optopt -- current option letter

# Returns
#    -1     at end of options
#    ?      for unrecognized option
#    <c>    a character representing the current option

# Private Data
#    _opti  index in multi-flag option, e.g., -abc
```

The function starts out with some documentation: who wrote the code, and when it was revised, followed by a list of the global variables it uses, what the return values are and what they mean, and any global variables that are "private" to this library function. Such documentation is essential for any program, and particularly for library functions.

```
function getopt(argc, argv, options,    optl, thisopt, i)
{
    optl = length(options)
    if (optl == 0)        # no options given
        return -1

    if (argv[Optind] == "--") {  # all done
        Optind++
        _opti = 0
```


```
        return -1
    } else if (argv[Optind] !~ /^-[^: \t\n\f\r\v\b]/) {
        _opti = 0
        return -1
    }
```

The function first checks that it was indeed called with a string of options (the `options` parameter). If `options` has a zero length, `getopt` immediately returns -1.

The next thing to check for is the end of the options. A `--' ends the command line options, as does any command line argument that does not begin with a `-'. `Optind` is used to step through the array of command line arguments; it retains its value across calls to `getopt`, since it is a global variable.

The regexp used, `/^-[^: \t\n\f\r\v\b]/`, is perhaps a bit of overkill; it checks for a `-' followed by anything that is not whitespace and not a colon. If the current command line argument does not match this pattern, it is not an option, and it ends option processing.

```
    if (_opti == 0)
        _opti = 2
    thisopt = substr(argv[Optind], _opti, 1)
    Optopt = thisopt
    i = index(options, thisopt)
    if (i == 0) {
        if (Opterr)
            printf("%c -- invalid option\n",
                                    thisopt) > "/dev/stderr"
        if (_opti >= length(argv[Optind])) {
            Optind++
            _opti = 0
        } else
            _opti++
        return "?"
    }
```

The `_opti` variable tracks the position in the current command line argument (`argv[Optind]`). In the case that multiple options were grouped together with one `-' (e.g., `-abx'), it is necessary to return them to the user one at a time.

If `_opti` is equal to zero, it is set to two, the index in the string of the next character to look at (we skip the `-', which is at position one). The variable `thisopt` holds the character, obtained with `substr`. It is saved in `Optopt` for the main program to use.

If `thisopt` is not in the `options` string, then it is an invalid option. If `Opterr` is non-zero, `getopt` prints an error message on the standard error that is similar to the message from the C version of `getopt`.

Since the option is invalid, it is necessary to skip it and move on to the next option character. If `_opti` is greater than or equal to the length of the current command line argument, then it is necessary to move on

to the next one, so `Optind` is incremented and `_opti` is reset to zero. Otherwise, `Optind` is left al and `_opti` is merely incremented.

In any case, since the option was invalid, `getopt` returns `?'. The main program can examine `Optopt` if it needs to know what the invalid option letter actually was.

```
    if (substr(options, i + 1, 1) == ":") {
        # get option argument
        if (length(substr(argv[Optind], _opti + 1)) > 0)
            Optarg = substr(argv[Optind], _opti + 1)
        else
            Optarg = argv[++Optind]
        _opti = 0
    } else
        Optarg = ""
```

If the option requires an argument, the option letter is followed by a colon in the `options` string. If there are remaining characters in the current command line argument (`argv[Optind]`), then the rest of that string is assigned to `Optarg`. Otherwise, the next command line argument is used (`-xFOO' vs. `-x FOO'). In either case, `_opti` is reset to zero, since there are no more characters left to examine in the current command line argument.

```
    if (_opti == 0 || _opti >= length(argv[Optind])) {
        Optind++
        _opti = 0
    } else
        _opti++
    return thisopt
}
```

Finally, if `_opti` is either zero or greater than the length of the current command line argument, it means this element in `argv` is through being processed, so `Optind` is incremented to point to the next element in `argv`. If neither condition is true, then only `_opti` is incremented, so that the next option letter can be processed on the next call to `getopt`.

```
BEGIN {
    Opterr = 1    # default is to diagnose
    Optind = 1    # skip ARGV[0]

    # test program
    if (_getopt_test) {
        while ((_go_c = getopt(ARGC, ARGV, "ab:cd")) != -1)
            printf("c = <%c>, optarg = <%s>\n",
                                        _go_c, Optarg)
        printf("non-option arguments:\n")
        for (; Optind < ARGC; Optind++)
```


```
            printf("\tARGV[%d] = <%s>\n",
                                    Optind, ARGV[Optind])
    }
}
```

The `BEGIN` rule initializes both `Opterr` and `Optind` to one. `Opterr` is set to one, since the default behavior is for `getopt` to print a diagnostic message upon seeing an invalid option. `Optind` is set to one, since there's no reason to look at the program name, which is in `ARGV[0]`.

The rest of the `BEGIN` rule is a simple test program. Here is the result of two sample runs of the test program.

```
$ awk -f getopt.awk -v _getopt_test=1 -- -a -cbARG bax -x
-| c = <a>, optarg = <>
-| c = <c>, optarg = <>
-| c = <b>, optarg = <ARG>
-| non-option arguments:
-|         ARGV[3] = <bax>
-|         ARGV[4] = <-x>

$ awk -f getopt.awk -v _getopt_test=1 -- -a -x -- xyz abc
-| c = <a>, optarg = <>
error--> x -- invalid option
-| c = <?>, optarg = <>
-| non-option arguments:
-|         ARGV[4] = <xyz>
-|         ARGV[5] = <abc>
```

The first `--' terminates the arguments to `awk`, so that it does not try to interpret the `-a' etc. as its own options.

Several of the sample programs presented in section Practical `awk` Programs, use `getopt` to process their arguments.

Reading the User Database

The `/dev/user' special file (see section Special File Names in `gawk`) provides access to the current user's real and effective user and group id numbers, and if available, the user's supplementary group set. However, since these are numbers, they do not provide very useful information to the average user. There needs to be some way to find the user information associated with the user and group numbers. This section presents a suite of functions for retrieving information from the user database. See section Reading the Group Database, for a similar suite that retrieves information from the group database.

The POSIX standard does not define the file where user information is kept. Instead, it provides the `<pwd.h>` header file and several C language subroutines for obtaining user information. The primary function is `getpwent`, for "get password entry." The "password" comes from the original user database

file, `/etc/passwd', which kept user information, along with the encrypted passwords (hence th name).

While an `awk` program could simply read `/etc/passwd' directly (the format is well known), because of the way password files are handled on networked systems, this file may not contain complete information about the system's set of users.

To be sure of being able to produce a readable, complete version of the user database, it is necessary to write a small C program that calls `getpwent`. `getpwent` is defined to return a pointer to a `struct passwd`. Each time it is called, it returns the next entry in the database. When there are no more entries, it returns `NULL`, the null pointer. When this happens, the C program should call `endpwent` to close the database. Here is `pwcat`, a C program that "cats" the password database.

```
/*
 * pwcat.c
 *
 * Generate a printable version of the password database
 *
 * Arnold Robbins
 * arnold@gnu.ai.mit.edu
 * May 1993
 * Public Domain
 */

#include <stdio.h>
#include <pwd.h>

int
main(argc, argv)
int argc;
char **argv;
{
    struct passwd *p;

    while ((p = getpwent()) != NULL)
        printf("%s:%s:%d:%d:%s:%s:%s\n",
            p->pw_name, p->pw_passwd, p->pw_uid,
            p->pw_gid, p->pw_gecos, p->pw_dir, p->pw_shell);

    endpwent();
    exit(0);
}
```

If you don't understand C, don't worry about it. The output from `pwcat` is the user database, in the traditional `/etc/passwd' format of colon-separated fields. The fields are:

Login name

The user's login name.

Encrypted password

The user's encrypted password. This may not be available on some systems.

User-ID

The user's numeric user-id number.

Group-ID

The user's numeric group-id number.

Full name

The user's full name, and perhaps other information associated with the user.

Home directory

The user's login, or "home" directory (familiar to shell programmers as `$HOME`).

Login shell

The program that will be run when the user logs in. This is usually a shell, such as Bash (the Gnu Bourne-Again shell).

Here are a few lines representative of `pwcat`'s output.

```
$ pwcat
-| root:3Ov02d5VaUPB6:0:1:Operator:/:/bin/sh
-| nobody:*:65534:65534::/:
-| daemon:*:1:1::/:
-| sys:*:2:2::/:/bin/csh
-| bin:*:3:3::/bin:
-| arnold:xyzzy:2076:10:Arnold Robbins:/home/arnold:/bin/sh
-| miriam:yxaay:112:10:Miriam Robbins:/home/miriam:/bin/sh
...
```

With that introduction, here is a group of functions for getting user information. There are several functions here, corresponding to the C functions of the same name.

```
# passwd.awk -- access password file information
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# May 1993

BEGIN {
    # tailor this to suit your system
    _pw_awklib = "/usr/local/libexec/awk/"
}

function _pw_init(    oldfs, oldrs, olddol0, pwcat)
{
    if (_pw_inited)
        return
```


```
    oldfs = FS
    oldrs = RS
    olddol0 = $0
    FS = ":"
    RS = "\n"
    pwcat = _pw_awklib "pwcat"
    while ((pwcat | getline) > 0) {
        _pw_byname[$1] = $0
        _pw_byuid[$3] = $0
        _pw_bycount[++_pw_total] = $0
    }
    close(pwcat)
    _pw_count = 0
    _pw_inited = 1
    FS = oldfs
    RS = oldrs
    $0 = olddol0
}
```

The `BEGIN` rule sets a private variable to the directory where `pwcat` is stored. Since it is used to help out an `awk` library routine, we have chosen to put it in `` `/usr/local/libexec/awk' ``. You might want it to be in a different directory on your system.

The function `_pw_init` keeps three copies of the user information in three associative arrays. The arrays are indexed by user name (`_pw_byname`), by user-id number (`_pw_byuid`), and by order of occurrence (`_pw_bycount`).

The variable `_pw_inited` is used for efficiency; `_pw_init` only needs to be called once.

Since this function uses `getline` to read information from `pwcat`, it first saves the values of `FS`, `RS`, and `$0`. Doing so is necessary, since these functions could be called from anywhere within a user's program, and the user may have his or her own values for `FS` and `RS`.

The main part of the function uses a loop to read database lines, split the line into fields, and then store the line into each array as necessary. When the loop is done, `_pw_init` cleans up by closing the pipeline, setting `_pw_inited` to one, and restoring `FS`, `RS`, and `$0`. The use of `_pw_count` will be explained below.

```
function getpwnam(name)
{
    _pw_init()
    if (name in _pw_byname)
        return _pw_byname[name]
    return ""
}
```

The `getpwnam` function takes a user name as a string argument. If that user is in the database, it returns

the appropriate line. Otherwise it returns the null string.

```
function getpwuid(uid)
{
    _pw_init()
    if (uid in _pw_byuid)
        return _pw_byuid[uid]
    return ""
}
```

Similarly, the `getpwuid` function takes a user-id number argument. If that user number is in the database, it returns the appropriate line. Otherwise it returns the null string.

```
function getpwent()
{
    _pw_init()
    if (_pw_count < _pw_total)
        return _pw_bycount[++_pw_count]
    return ""
}
```

The `getpwent` function simply steps through the database, one entry at a time. It uses `_pw_count` to track its current position in the `_pw_bycount` array.

```
function endpwent()
{
    _pw_count = 0
}
```

The `endpwent` function resets `_pw_count` to zero, so that subsequent calls to `getpwent` will start over again.

A conscious design decision in this suite is that each subroutine calls `_pw_init` to initialize the database arrays. The overhead of running a separate process to generate the user database, and the I/O to scan it, will only be incurred if the user's main program actually calls one of these functions. If this library file is loaded along with a user's program, but none of the routines are ever called, then there is no extra run-time overhead. (The alternative would be to move the body of `_pw_init` into a `BEGIN` rule, which would always run `pwcat`. This simplifies the code but runs an extra process that may never be needed.)

In turn, calling `_pw_init` is not too expensive, since the `_pw_inited` variable keeps the program from reading the data more than once. If you are worried about squeezing every last cycle out of your `awk` program, the check of `_pw_inited` could be moved out of `_pw_init` and duplicated in all the other functions. In practice, this is not necessary, since most `awk` programs are I/O bound, and it would clutter up the code.

The `id` program in section Printing Out User Information, uses these functions.

Reading the Group Database

Much of the discussion presented in section Reading the User Database, applies to the group database as well. Although there has traditionally been a well known file, `/etc/group', in a well known format, the POSIX standard only provides a set of C library routines (<grp.h> and getgrent) for accessing the information. Even though this file may exist, it likely does not have complete information. Therefore, as with the user database, it is necessary to have a small C program that generates the group database as its output.

Here is grcat, a C program that "cats" the group database.

```
/*
 * grcat.c
 *
 * Generate a printable version of the group database
 *
 * Arnold Robbins, arnold@gnu.ai.mit.edu
 * May 1993
 * Public Domain
 */

#include <stdio.h>
#include <grp.h>

int
main(argc, argv)
int argc;
char **argv;
{
    struct group *g;
    int i;

    while ((g = getgrent()) != NULL) {
        printf("%s:%s:%d:", g->gr_name, g->gr_passwd,
                                            g->gr_gid);
        for (i = 0; g->gr_mem[i] != NULL; i++) {
            printf("%s", g->gr_mem[i]);
            if (g->gr_mem[i+1] != NULL)
                putchar(',');
        }
        putchar('\n');
    }
    endgrent();
    exit(0);
```


```
}
```

Each line in the group database represent one group. The fields are separated with colons, and represent the following information.

Group Name

The name of the group.

Group Password

The encrypted group password. In practice, this field is never used. It is usually empty, or set to `*'.

Group ID Number

The numeric group-id number. This number should be unique within the file.

Group Member List

A comma-separated list of user names. These users are members of the group. Most Unix systems allow users to be members of several groups simultaneously. If your system does, then reading `/dev/user' will return those group-id numbers in $5 through $NF. (Note that `/dev/user' is a gawk extension; see section Special File Names in gawk.)

Here is what running grcat might produce:

```
$ grcat
-| wheel:*:0:arnold
-| nogroup:*:65534:
-| daemon:*:1:
-| kmem:*:2:
-| staff:*:10:arnold,miriam,andy
-| other:*:20:
...
```

Here are the functions for obtaining information from the group database. There are several, modeled after the C library functions of the same names.

```
# group.awk -- functions for dealing with the group file
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# May 1993

BEGIN    \
{
    # Change to suit your system
    _gr_awklib = "/usr/local/libexec/awk/"
}

function _gr_init(    oldfs, oldrs, olddol0, grcat, n, a, i)
{
    if (_gr_inited)
```


```
        return

    oldfs = FS
    oldrs = RS
    olddol0 = $0
    FS = ":"
    RS = "\n"

    grcat = _gr_awklib "grcat"
    while ((grcat | getline) > 0) {
        if ($1 in _gr_byname)
            _gr_byname[$1] = _gr_byname[$1] "," $4
        else
            _gr_byname[$1] = $0
        if ($3 in _gr_bygid)
            _gr_bygid[$3] = _gr_bygid[$3] "," $4
        else
            _gr_bygid[$3] = $0

        n = split($4, a, "[ \t]*,[ \t]*")
        for (i = 1; i <= n; i++)
            if (a[i] in _gr_groupsbyuser)
                _gr_groupsbyuser[a[i]] = \
                    _gr_groupsbyuser[a[i]] " " $1
            else
                _gr_groupsbyuser[a[i]] = $1

        _gr_bycount[++_gr_count] = $0
    }
    close(grcat)
    _gr_count = 0
    _gr_inited++
    FS = oldfs
    RS = oldrs
    $0 = olddol0
}
```

The `BEGIN` rule sets a private variable to the directory where `grcat` is stored. Since it is used to help out an `awk` library routine, we have chosen to put it in `` `/usr/local/libexec/awk' ``. You might want it to be in a different directory on your system.

These routines follow the same general outline as the user database routines (see section Reading the User Database). The `_gr_inited` variable is used to ensure that the database is scanned no more than once. The `_gr_init` function first saves `FS`, `RS`, and `$0`, and then sets `FS` and `RS` to the correct values for scanning the group information.

The group information is stored is several associative arrays. The arrays are indexed by group name (`_gr_byname`), by group-id number (`_gr_bygid`), and by position in the database (`_gr_bycou`). There is an additional array indexed by user name (`_gr_groupsbyuser`), that is a space separated list of groups that each user belongs to.

Unlike the user database, it is possible to have multiple records in the database for the same group. This is common when a group has a large number of members. Such a pair of entries might look like:

```
tvpeople:*:101:johny,jay,arsenio
tvpeople:*:101:david,conan,tom,joan
```

For this reason, `_gr_init` looks to see if a group name or group-id number has already been seen. If it has, then the user names are simply concatenated onto the previous list of users. (There is actually a subtle problem with the code presented above. Suppose that the first time there were no names. This code adds the names with a leading comma. It also doesn't check that there is a `$4`.)

Finally, `_gr_init` closes the pipeline to `grcat`, restores `FS`, `RS`, and `$0`, initializes `_gr_count` to zero (it is used later), and makes `_gr_inited` non-zero.

```
function getgrnam(group)
{
    _gr_init()
    if (group in _gr_byname)
        return _gr_byname[group]
    return ""
}
```

The `getgrnam` function takes a group name as its argument, and if that group exists, it is returned. Otherwise, `getgrnam` returns the null string.

```
function getgrgid(gid)
{
    _gr_init()
    if (gid in _gr_bygid)
        return _gr_bygid[gid]
    return ""
}
```

The `getgrgid` function is similar, it takes a numeric group-id, and looks up the information associated with that group-id.

```
function getgruser(user)
{
    _gr_init()
    if (user in _gr_groupsbyuser)
        return _gr_groupsbyuser[user]
```


```
    return ""
}
```

The `getgruser` function does not have a C counterpart. It takes a user name, and returns the list of groups that have the user as a member.

```
function getgrent()
{
    _gr_init()
    if (++gr_count in _gr_bycount)
        return _gr_bycount[_gr_count]
    return ""
}
```

The `getgrent` function steps through the database one entry at a time. It uses `_gr_count` to track its position in the list.

```
function endgrent()
{
    _gr_count = 0
}
```

`endgrent` resets `_gr_count` to zero so that `getgrent` can start over again.

As with the user database routines, each function calls `_gr_init` to initialize the arrays. Doing so only incurs the extra overhead of running `grcat` if these functions are used (as opposed to moving the body of `_gr_init` into a `BEGIN` rule).

Most of the work is in scanning the database and building the various associative arrays. The functions that the user calls are themselves very simple, relying on `awk`'s associative arrays to do work.

The `id` program in section Printing Out User Information, uses these functions.

Naming Library Function Global Variables

Due to the way the `awk` language evolved, variables are either global (usable by the entire program), or local (usable just by a specific function). There is no intermediate state analogous to `static` variables in C.

Library functions often need to have global variables that they can use to preserve state information between calls to the function. For example, `getopt`'s variable `_opti` (see section Processing Command Line Options), and the `_tm_months` array used by `mktime` (see section Turning Dates Into Timestamps). Such variables are called private, since the only functions that need to use them are the ones in the library.

When writing a library function, you should try to choose names for your private variables so that they will not conflict with any variables used by either another library function or a user's main program. For

example, a name like `i' or `j' is not a good choice, since user programs often use variable names like these for their own purposes.

The example programs shown in this chapter all start the names of their private variables with an underscore (`_'). Users generally don't use leading underscores in their variable names, so this convention immediately decreases the chances that the variable name will be accidentally shared with the user's program.

In addition, several of the library functions use a prefix that helps indicate what function or set of functions uses the variables. For example, `_tm_months` in `mktime` (see section Turning Dates Into Timestamps), and `_pw_byname` in the user data base routines (see section Reading the User Database). This convention is recommended, since it even further decreases the chance of inadvertent conflict among variable names. Note that this convention can be used equally well both for variable names and for private function names too.

While I could have re-written all the library routines to use this convention, I did not do so, in order to show how my own `awk` programming style has evolved, and to provide some basis for this discussion.

As a final note on variable naming, if a function makes global variables available for use by a main program, it is a good convention to start that variable's name with a capital letter. For example, `getopt`'s `Opterr` and `Optind` variables (see section Processing Command Line Options). The leading capital letter indicates that it is global, while the fact that the variable name is not all capital letters indicates that the variable is not one of `awk`'s built-in variables, like `FS`.

It is also important that *all* variables in library functions that do not need to save state are in fact declared local. If this is not done, the variable could accidentally be used in the user's program, leading to bugs that are very difficult to track down.

```
function lib_func(x, y,    l1, l2)
{
    ...
    use variable some_var  # some_var could be local
    ...                    # but is not by oversight
}
```

A different convention, common in the Tcl community, is to use a single associative array to hold the values needed by the library function(s), or "package." This significantly decreases the number of actual global names in use. For example, the functions described in section Reading the User Database, might have used `PW_data["inited"]`, `PW_data["total"]`, `PW_data["count"]` and `PW_data["awklib"]`, instead of `_pw_inited`, `_pw_awklib`, `_pw_total`, and `_pw_count`.

The conventions presented in this section are exactly that, conventions. You are not required to write your programs this way, we merely recommend that you do so.

Go to the previous, next section.

Go to the previous, next section.

Practical awk Programs

This chapter presents a potpourri of `awk` programs for your reading enjoyment. There are two sections. The first presents `awk` versions of several common POSIX utilities. The second is a grab-bag of interesting programs.

Many of these programs use the library functions presented in section A Library of `awk` Functions.

Re-inventing Wheels for Fun and Profit

This section presents a number of POSIX utilities that are implemented in `awk`. Re-inventing these programs in `awk` is often enjoyable, since the algorithms can be very clearly expressed, and usually the code is very concise and simple. This is true because `awk` does so much for you.

It should be noted that these programs are not necessarily intended to replace the installed versions on your system. Instead, their purpose is to illustrate `awk` language programming for "real world" tasks.

The programs are presented in alphabetical order.

Cutting Out Fields and Columns

The `cut` utility selects, or "cuts," either characters or fields from its standard input and sends them to its standard output. `cut` can cut out either a list of characters, or a list of fields. By default, fields are separated by tabs, but you may supply a command line option to change the field delimiter, i.e. the field separator character. `cut`'s definition of fields is less general than `awk`'s.

A common use of `cut` might be to pull out just the login name of logged-on users from the output of `who`. For example, the following pipeline generates a sorted, unique list of the logged on users:

```
who | cut -c1-8 | sort | uniq
```

The options for `cut` are:

`-c list`
: Use list as the list of characters to cut out. Items within the list may be separated by commas, and ranges of characters can be separated with dashes. The list `1-8,15,22-35' specifies characters one through eight, 15, and 22 through 35.

`-f list`
: Use list as the list of fields to cut out.

`-d delim`
: Use delim as the field separator character instead of the tab character.

`-s`

TOC

Suppress printing of lines that do not contain the field delimiter.

The `awk` implementation of `cut` uses the `getopt` library function (see section Processing Command Line Options), and the `join` library function (see section Merging an Array Into a String).

The program begins with a comment describing the options and a `usage` function which prints out a usage message and exits. `usage` is called if invalid arguments are supplied.

```
# cut.awk -- implement cut in awk
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# May 1993

# Options:
#    -f list        Cut fields
#    -d c           Field delimiter character
#    -c list        Cut characters
#
#    -s        Suppress lines without the delimiter character

function usage(    e1, e2)
{
    e1 = "usage: cut [-f list] [-d c] [-s] [files...]"
    e2 = "usage: cut [-c list] [files...]"
    print e1 > "/dev/stderr"
    print e2 > "/dev/stderr"
    exit 1
}
```

The variables `e1` and `e2` are used so that the function fits nicely on the page.

Next comes a `BEGIN` rule that parses the command line options. It sets `FS` to a single tab character, since that is `cut`'s default field separator. The output field separator is also set to be the same as the input field separator. Then `getopt` is used to step through the command line options. One or the other of the variables `by_fields` or `by_chars` is set to true, to indicate that processing should be done by fields or by characters respectively. When cutting by characters, the output field separator is set to the null string.

```
BEGIN    \
{
    FS = "\t"    # default
    OFS = FS
    while ((c = getopt(ARGC, ARGV, "sf:c:d:")) != -1) {
        if (c == "f") {
            by_fields = 1
            fieldlist = Optarg
        } else if (c == "c") {
```


```
            by_chars = 1
            fieldlist = Optarg
            OFS = ""
        } else if (c == "d") {
            if (length(Optarg) > 1) {
                printf("Using first character of %s" \
                " for delimiter\n", Optarg) > "/dev/stderr"
                Optarg = substr(Optarg, 1, 1)
            }
            FS = Optarg
            OFS = FS
            if (FS == " ")    # defeat awk semantics
                FS = "[ ]"
        } else if (c == "s")
            suppress++
        else
            usage()
    }

    for (i = 1; i < Optind; i++)
        ARGV[i] = ""
```

Special care is taken when the field delimiter is a space. Using `" "` (a single space) for the value of `FS` is incorrect---`awk` would separate fields with runs of spaces and/or tabs, and we want them to be separated with individual spaces. Also, note that after `getopt` is through, we have to clear out all the elements of `ARGV` from one to `Optind`, so that `awk` will not try to process the command line options as file names.

After dealing with the command line options, the program verifies that the options make sense. Only one or the other of `-c' and `-f' should be used, and both require a field list. Then either `set_fieldlist` or `set_charlist` is called to pull apart the list of fields or characters.

```
    if (by_fields && by_chars)
        usage()

    if (by_fields == 0 && by_chars == 0)
        by_fields = 1    # default

    if (fieldlist == "") {
        print "cut: needs list for -c or -f" > "/dev/stderr"
        exit 1
    }

    if (by_fields)
        set_fieldlist()
    else
        set_charlist()
```


```
}
```

Here is `set_fieldlist`. It first splits the field list apart at the commas, into an array. Then, for each element of the array, it looks to see if it is actually a range, and if so splits it apart. The range is verified to make sure the first number is smaller than the second. Each number in the list is added to the `flist` array, which simply lists the fields that will be printed. Normal field splitting is used. The program lets `awk` handle the job of doing the field splitting.

```
function set_fieldlist(        n, m, i, j, k, f, g)
{
    n = split(fieldlist, f, ",")
    j = 1    # index in flist
    for (i = 1; i <= n; i++) {
        if (index(f[i], "-") != 0) { # a range
            m = split(f[i], g, "-")
            if (m != 2 || g[1] >= g[2]) {
                printf("bad field list: %s\n",
                                    f[i]) > "/dev/stderr"
                exit 1
            }
            for (k = g[1]; k <= g[2]; k++)
                flist[j++] = k
        } else
            flist[j++] = f[i]
    }
    nfields = j - 1
}
```

The `set_charlist` function is more complicated than `set_fieldlist`. The idea here is to use `gawk`'s `FIELDWIDTHS` variable (see section Reading Fixed-width Data), which describes constant width input. When using a character list, that is exactly what we have.

Setting up `FIELDWIDTHS` is more complicated than simply listing the fields that need to be printed. We have to keep track of the fields to be printed, and also the intervening characters that have to be skipped. For example, suppose you wanted characters one through eight, 15, and 22 through 35. You would use `-c 1-8,15,22-35'. The necessary value for `FIELDWIDTHS` would be `"8 6 1 6 14"`. This gives us five fields, and what should be printed are `$1`, `$3`, and `$5`. The intermediate fields are "filler," stuff in between the desired data.

`flist` lists the fields to be printed, and `t` tracks the complete field list, including filler fields.

```
function set_charlist(    field, i, j, f, g, t,
                          filler, last, len)
{
    field = 1   # count total fields
    n = split(fieldlist, f, ",")
```


```
    j = 1       # index in flist
    for (i = 1; i <= n; i++) {
        if (index(f[i], "-") != 0) { # range
            m = split(f[i], g, "-")
            if (m != 2 || g[1] >= g[2]) {
                printf(bad character list: %s\n",
                                f[i]) > "/dev/stderr"
                exit 1
            }
            len = g[2] - g[1] + 1
            if (g[1] > 1)  # compute length of filler
                filler = g[1] - last - 1
            else
                filler = 0
            if (filler)
                t[field++] = filler
            t[field++] = len  # length of field
            last = g[2]
            flist[j++] = field - 1
        } else {
            if (f[i] > 1)
                filler = f[i] - last - 1
            else
                filler = 0
            if (filler)
                t[field++] = filler
            t[field++] = 1
            last = f[i]
            flist[j++] = field - 1
        }
    }
    FIELDWIDTHS = join(t, 1, field - 1)
    nfields = j - 1
}
```

Here is the rule that actually processes the data. If the `-s' option was given, then `suppress` will be true. The first `if` statement makes sure that the input record does have the field separator. If `cut` is processing fields, `suppress` is true, and the field separator character is not in the record, then the record is skipped.

If the record is valid, then at this point, `gawk` has split the data into fields, either using the character in `FS` or using fixed-length fields and `FIELDWIDTHS`. The loop goes through the list of fields that should be printed. If the corresponding field has data in it, it is printed. If the next field also has data, then the separator character is written out in between the fields.

```
{
```


```
    if (by_fields && suppress && $0 !~ FS)
        next

    for (i = 1; i <= nfields; i++) {
        if ($flist[i] != "") {
            printf "%s", $flist[i]
            if (i < nfields && $flist[i+1] != "")
                printf "%s", OFS
        }
    }
    print ""
}
```

This version of `cut` relies on `gawk`'s `FIELDWIDTHS` variable to do the character-based cutting. While it would be possible in other `awk` implementations to use `substr` (see section Built-in Functions for String Manipulation), it would also be extremely painful to do so. The `FIELDWIDTHS` variable supplies an elegant solution to the problem of picking the input line apart by characters.

Searching for Regular Expressions in Files

The `egrep` utility searches files for patterns. It uses regular expressions that are almost identical to those available in `awk` (see section Regular Expression Constants). It is used this way:

```
egrep [ options ] 'pattern' files ...
```

The pattern is a regexp. In typical usage, the regexp is quoted to prevent the shell from expanding any of the special characters as file name wildcards. Normally, `egrep` prints the lines that matched. If multiple file names are provided on the command line, each output line is preceded by the name of the file and a colon.

The options are:

`-c`
: Print out a count of the lines that matched the pattern, instead of the lines themselves.

`-s`
: Be silent. No output is produced, and the exit value indicates whether or not the pattern was matched.

`-v`
: Invert the sense of the test. `egrep` prints the lines that do *not* match the pattern, and exits successfully if the pattern was not matched.

`-i`
: Ignore case distinctions in both the pattern and the input data.

`-l`
: Only print the names of the files that matched, not the lines that matched.

`-e pattern`

Use pattern as the regexp to match. The purpose of the `-e' option is to allow patterns that start with a `-'.

This version uses the `getopt` library function (see section Processing Command Line Options), and the file transition library program (see section Noting Data File Boundaries).

The program begins with a descriptive comment, and then a `BEGIN` rule that processes the command line arguments with `getopt`. The `-i' (ignore case) option is particularly easy with `gawk`; we just use the `IGNORECASE` built in variable (see section Built-in Variables).

```
# egrep.awk -- simulate egrep in awk
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# May 1993

# Options:
#    -c    count of lines
#    -s    silent - use exit value
#    -v    invert test, success if no match
#    -i    ignore case
#    -l    print filenames only
#    -e    argument is pattern

BEGIN {
    while ((c = getopt(ARGC, ARGV, "ce:svil")) != -1) {
        if (c == "c")
            count_only++
        else if (c == "s")
            no_print++
        else if (c == "v")
            invert++
        else if (c == "i")
            IGNORECASE = 1
        else if (c == "l")
            filenames_only++
        else if (c == "e")
            pattern = Optarg
        else
            usage()
    }
```

Next comes the code that handles the `egrep` specific behavior. If no pattern was supplied with `-e', the first non-option on the command line is used. The `awk` command line arguments up to `ARGV[Optind]` are cleared, so that `awk` won't try to process them as files. If no files were specified, the standard input is used, and if multiple files were specified, we make sure to note this so that the file names can precede the matched lines in the output.

The last two lines are commented out, since they are not needed in `gawk`. They should be uncommented if you have to use another version of `awk`.

```
    if (pattern == "")
        pattern = ARGV[Optind++]

    for (i = 1; i < Optind; i++)
        ARGV[i] = ""
    if (Optind >= ARGC) {
        ARGV[1] = "-"
        ARGC = 2
    } else if (ARGC - Optind > 1)
        do_filenames++

#    if (IGNORECASE)
#        pattern = tolower(pattern)
}
```

The next set of lines should be uncommented if you are not using `gawk`. This rule translates all the characters in the input line into lower-case if the `-i' option was specified. The rule is commented out since it is not necessary with `gawk`.

```
#{
#    if (IGNORECASE)
#        $0 = tolower($0)
#}
```

The `beginfile` function is called by the rule in `` `ftrans.awk' `` when each new file is processed. In this case, it is very simple; all it does is initialize a variable `fcount` to zero. `fcount` tracks how many lines in the current file matched the pattern.

```
function beginfile(junk)
{
    fcount = 0
}
```

The `endfile` function is called after each file has been processed. It is used only when the user wants a count of the number of lines that matched. `no_print` will be true only if the exit status is desired. `count_only` will be true if line counts are desired. `egrep` will therefore only print line counts if printing and counting are enabled. The output format must be adjusted depending upon the number of files to be processed. Finally, `fcount` is added to `total`, so that we know how many lines altogether matched the pattern.

```
function endfile(file)
{
```


```
    if (! no_print && count_only)
        if (do_filenames)
            print file ":" fcount
        else
            print fcount

    total += fcount
}
```

This rule does most of the work of matching lines. The variable `matches` will be true if the line matched the pattern. If the user wants lines that did not match, the sense of the `matches` is inverted using the `!' operator. `fcount` is incremented with the value of `matches`, which will be either one or zero, depending upon a successful or unsuccessful match. If the line did not match, the `next` statement just moves on to the next record.

There are several optimizations for performance in the following few lines of code. If the user only wants exit status (`no_print` is true), and we don't have to count lines, then it is enough to know that one line in this file matched, and we can skip on to the next file with `nextfile`. Along similar lines, if we are only printing file names, and we don't need to count lines, we can print the file name, and then skip to the next file with `nextfile`.

Finally, each line is printed, with a leading filename and colon if necessary.

```
{
    matches = ($0 ~ pattern)
    if (invert)
        matches = ! matches

    fcount += matches    # 1 or 0

    if (! matches)
        next

    if (no_print && ! count_only)
        nextfile

    if (filenames_only && ! count_only) {
        print FILENAME
        nextfile
    }

    if (do_filenames && ! count_only)
        print FILENAME ":" $0
    else if (! count_only)
        print
}
```

The END rule takes care of producing the correct exit status. If there were no matches, the exit status is one, otherwise it is zero.

```
END    \
{
    if (total == 0)
        exit 1
    exit 0
}
```

The usage function prints a usage message in case of invalid options and then exits.

```
function usage(    e)
{
    e = "Usage: egrep [-csvil] [-e pat] [files ...]"
    print e > "/dev/stderr"
    exit 1
}
```

The variable e is used so that the function fits nicely on the printed page.

Printing Out User Information

The id utility lists a user's real and effective user-id numbers, real and effective group-id numbers, and the user's group set, if any. id will only print the effective user-id and group-id if they are different from the real ones. If possible, id will also supply the corresponding user and group names. The output might look like this:

```
$ id
-| uid=2076(arnold) gid=10(staff) groups=10(staff),4(tty)
```

This information is exactly what is provided by gawk's `/dev/user' special file (see section Special File Names in gawk). However, the id utility provides a more palatable output than just a string of numbers.

Here is a simple version of id written in awk. It uses the user database library functions (see section Reading the User Database), and the group database library functions (see section Reading the Group Database).

The program is fairly straightforward. All the work is done in the BEGIN rule. The user and group id numbers are obtained from `/dev/user'. If there is no support for `/dev/user', the program gives up.

The code is repetitive. The entry in the user database for the real user-id number is split into parts at the `:'. The name is the first field. Similar code is used for the effective user-id number, and the group numbers.


```
# id.awk -- implement id in awk
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# May 1993

# output is:
# uid=12(foo) euid=34(bar) gid=3(baz) \
#             egid=5(blat) groups=9(nine),2(two),1(one)

BEGIN    \
{
    if ((getline < "/dev/user") < 0) {
        err = "id: no /dev/user support - cannot run"
        print err > "/dev/stderr"
        exit 1
    }
    close("/dev/user")

    uid = $1
    euid = $2
    gid = $3
    egid = $4

    printf("uid=%d", uid)
    pw = getpwuid(uid)
    if (pw != "") {
        split(pw, a, ":")
        printf("(%s)", a[1])
    }

    if (euid != uid) {
        printf(" euid=%d", euid)
        pw = getpwuid(euid)
        if (pw != "") {
            split(pw, a, ":")
            printf("(%s)", a[1])
        }
    }

    printf(" gid=%d", gid)
    pw = getgrgid(gid)
    if (pw != "") {
        split(pw, a, ":")
        printf("(%s)", a[1])
    }
```


```
    if (egid != gid) {
        printf(" egid=%d", egid)
        pw = getgrgid(egid)
        if (pw != "") {
            split(pw, a, ":")
            printf("(%s)", a[1])
        }
    }

    if (NF > 4) {
        printf(" groups=");
        for (i = 5; i <= NF; i++) {
            printf("%d", $i)
            pw = getgrgid($i)
            if (pw != "") {
                split(pw, a, ":")
                printf("(%s)", a[1])
            }
            if (i < NF)
                printf(",")
        }
    }
    print ""
}
```

Splitting a Large File Into Pieces

The `split` program splits large text files into smaller pieces. By default, the output files are named `` `xaa' ``, `` `xab' ``, and so on. Each file has 1000 lines in it, with the likely exception of the last file. To change the number of lines in each file, you supply a number on the command line preceded with a minus, e.g., `-500' for files with 500 lines in them instead of 1000. To change the name of the output files to something like `` `myfileaa' ``, `` `myfileab' ``, and so on, you supply an additional argument that specifies the filename.

Here is a version of `split` in `awk`. It uses the `ord` and `chr` functions presented in section Translating Between Characters and Numbers.

The program first sets its defaults, and then tests to make sure there are not too many arguments. It then looks at each argument in turn. The first argument could be a minus followed by a number. If it is, this happens to look like a negative number, so it is made positive, and that is the count of lines. The data file name is skipped over, and the final argument is used as the prefix for the output file names.

```
# split.awk -- do split in awk
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# May 1993
```


```
# usage: split [-num] [file] [outname]

BEGIN    \
{
    outfile = "x"    # default
    count = 1000
    if (ARGC > 4)
        usage()

    i = 1
    if (ARGV[i] ~ /^-[0-9]+$/) {
        count = -ARGV[i]
        ARGV[i] = ""
        i++
    }
    # test argv in case reading from stdin instead of file
    if (i in ARGV)
        i++    # skip data file name
    if (i in ARGV) {
        outfile = ARGV[i]
        ARGV[i] = ""
    }

    s1 = s2 = "a"
    out = (outfile s1 s2)
}
```

The next rule does most of the work. `tcount` (temporary count) tracks how many lines have been printed to the output file so far. If it is greater than `count`, it is time to close the current file and start a new one. `s1` and `s2` track the current suffixes for the file name. If they are both `z', the file is just too big. Otherwise, `s1` moves to the next letter in the alphabet and `s2` starts over again at `a'.

```
{
    if (++tcount > count) {
        close(out)
        if (s2 == "z") {
            if (s1 == "z") {
                printf("split: %s is too large to split\n", \
                       FILENAME) > "/dev/stderr"
                exit 1
            }
            s1 = chr(ord(s1) + 1)
            s2 = "a"
        } else
            s2 = chr(ord(s2) + 1)
        out = (outfile s1 s2)
```


```
        tcount = 1
    }
    print > out
}
```

The `usage` function simply prints an error message and exits.

```
function usage(   e)
{
    e = "usage: split [-num] [file] [outname]"
    print e > "/dev/stderr"
    exit 1
}
```

The variable `e` is used so that the function fits nicely on the page.

This program is a bit sloppy; it relies on `awk` to close the last file for it automatically, instead of doing it in an `END` rule.

Duplicating Output Into Multiple Files

The `tee` program is known as a "pipe fitting." `tee` copies its standard input to its standard output, and also duplicates it to the files named on the command line. Its usage is:

```
tee [-a] file ...
```

The `-a' option tells `tee` to append to the named files, instead of truncating them and starting over.

The `BEGIN` rule first makes a copy of all the command line arguments, into an array named `copy`. `ARGV[0]` is not copied, since it is not needed. `tee` cannot use `ARGV` directly, since `awk` will attempt to process each file named in `ARGV` as input data.

If the first argument is `-a', then the flag variable `append` is set to true, and both `ARGV[1]` and `copy[1]` are deleted. If `ARGC` is less than two, then no file names were supplied, and `tee` prints a usage message and exits. Finally, `awk` is forced to read the standard input by setting `ARGV[1]` to `"-"`, and `ARGC` to two.

```
# tee.awk -- tee in awk
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# May 1993
# Revised December 1995

BEGIN    \
{
    for (i = 1; i < ARGC; i++)
        copy[i] = ARGV[i]
```


```
    if (ARGV[1] == "-a") {
        append = 1
        delete ARGV[1]
        delete copy[1]
        ARGC--
    }
    if (ARGC < 2) {
        print "usage: tee [-a] file ..." > "/dev/stderr"
        exit 1
    }
    ARGV[1] = "-"
    ARGC = 2
}
```

The single rule does all the work. Since there is no pattern, it is executed for each line of input. The body of the rule simply prints the line into each file on the command line, and then to the standard output.

```
{
    # moving the if outside the loop makes it run faster
    if (append)
        for (i in copy)
            print >> copy[i]
    else
        for (i in copy)
            print > copy[i]
    print
}
```

It would have been possible to code the loop this way:

```
for (i in copy)
    if (append)
        print >> copy[i]
    else
        print > copy[i]
```

This is more concise, but it is also less efficient. The `if' is tested for each record and for each output file. By duplicating the loop body, the `if' is only tested once for each input record. If there are N input records and M input files, the first method only executes N `if' statements, while the second would execute N*M `if' statements.

Finally, the `END' rule cleans up, by closing all the output files.

```
END    \
{
    for (i in copy)
```


```
        close(copy[i])
}
```

Printing Non-duplicated Lines of Text

The `uniq` utility reads sorted lines of data on its standard input, and (by default) removes duplicate lines. In other words, only unique lines are printed, hence the name. `uniq` has a number of options. The usage is:

```
uniq [-udc [-n]] [+n] [ input file [ output file ]]
```

The option meanings are:

`-d`
: Only print repeated lines.

`-u`
: Only print non-repeated lines.

`-c`
: Count lines. This option overrides `-d' and `-u'. Both repeated and non-repeated lines are counted.

`-n`
: Skip n fields before comparing lines. The definition of fields is the same as `awk`'s default: non-whitespace characters separated by runs of spaces and/or tabs.

`+n`
: Skip n characters before comparing lines. Any fields specified with `-n' are skipped first.

`input file`
: Data is read from the input file named on the command line, instead of from the standard input.

`output file`
: The generated output is sent to the named output file, instead of to the standard output.

Normally `uniq` behaves as if both the `-d' and `-u' options had been provided.

Here is an `awk` implementation of `uniq`. It uses the `getopt` library function (see section Processing Command Line Options), and the `join` library function (see section Merging an Array Into a String).

The program begins with a `usage` function and then a brief outline of the options and their meanings in a comment.

The `BEGIN` rule deals with the command line arguments and options. It uses a trick to get `getopt` to handle options of the form `-25', treating such an option as the option letter `2' with an argument of `5'. If indeed two or more digits were supplied (`Optarg` looks like a number), `Optarg` is concatenated with the option digit, and then result is added to zero to make it into a number. If there is only one digit in the option, then `Optarg` is not needed, and `Optind` must be decremented so that `getopt` will process it next time. This code is admittedly a bit tricky.

If no options were supplied, then the default is taken, to print both repeated and non-repeated lines. The

output file, if provided, is assigned to `outputfile`. Earlier, `outputfile` was initialized to the standard output, `` `/dev/stdout' ``.

```
# uniq.awk -- do uniq in awk
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# May 1993

function usage(    e)
{
    e = "Usage: uniq [-udc [-n]] [+n] [ in [ out ]]"
    print e > "/dev/stderr"
    exit 1
}

# -c    count lines. overrides -d and -u
# -d    only repeated lines
# -u    only non-repeated lines
# -n    skip n fields
# +n    skip n characters, skip fields first

BEGIN    \
{
    count = 1
    outputfile = "/dev/stdout"
    opts = "udc0:1:2:3:4:5:6:7:8:9:"
    while ((c = getopt(ARGC, ARGV, opts)) != -1) {
        if (c == "u")
            non_repeated_only++
        else if (c == "d")
            repeated_only++
        else if (c == "c")
            do_count++
        else if (index("0123456789", c) != 0) {
            # getopt requires args to options
            # this messes us up for things like -5
            if (Optarg ~ /^[0-9]+$/)
                fcount = (c Optarg) + 0
            else {
                fcount = c + 0
                Optind--
            }
        } else
            usage()
    }
```


```
    if (ARGV[Optind] ~ /^\+[0-9]+$/) {
        charcount = substr(ARGV[Optind], 2) + 0
        Optind++
    }

    for (i = 1; i < Optind; i++)
        ARGV[i] = ""

    if (repeated_only == 0 && non_repeated_only == 0)
        repeated_only = non_repeated_only = 1

    if (ARGC - Optind == 2) {
        outputfile = ARGV[ARGC - 1]
        ARGV[ARGC - 1] = ""
    }
}
```

The following function, `are_equal`, compares the current line, `$0`, to the previous line, `last`. It handles skipping fields and characters.

If no field count and no character count were specified, `are_equal` simply returns one or zero depending upon the result of a simple string comparison of `last` and `$0`. Otherwise, things get more complicated.

If fields have to be skipped, each line is broken into an array using `split` (see section Built-in Functions for String Manipulation), and then the desired fields are joined back into a line using `join`. The joined lines are stored in `clast` and `cline`. If no fields are skipped, `clast` and `cline` are set to `last` and `$0` respectively.

Finally, if characters are skipped, `substr` is used to strip off the leading `charcount` characters in `clast` and `cline`. The two strings are then compared, and `are_equal` returns the result.

```
function are_equal(    n, m, clast, cline, alast, aline)
{
    if (fcount == 0 && charcount == 0)
        return (last == $0)

    if (fcount > 0) {
        n = split(last, alast)
        m = split($0, aline)
        clast = join(alast, fcount+1, n)
        cline = join(aline, fcount+1, m)
    } else {
        clast = last
        cline = $0
    }
```


```
    if (charcount) {
        clast = substr(clast, charcount + 1)
        cline = substr(cline, charcount + 1)
    }

    return (clast == cline)
}
```

The following two rules are the body of the program. The first one is executed only for the very first line of data. It sets `last` equal to `$0`, so that subsequent lines of text have something to be compared to.

The second rule does the work. The variable `equal` will be one or zero depending upon the results of `are_equal`'s comparison. If `uniq` is counting repeated lines, then the `count` variable is incremented if the lines are equal. Otherwise the line is printed and `count` is reset, since the two lines are not equal.

If `uniq` is not counting, `count` is incremented if the lines are equal. Otherwise, if `uniq` is counting repeated lines, and more than one line has been seen, or if `uniq` is counting non-repeated lines, and only one line has been seen, then the line is printed, and `count` is reset.

Finally, similar logic is used in the `END` rule to print the final line of input data.

```
NR == 1 {
    last = $0
    next
}

{
    equal = are_equal()

    if (do_count) {    # overrides -d and -u
        if (equal)
            count++
        else {
            printf("%4d %s\n", count, last) > outputfile
            last = $0
            count = 1    # reset
        }
        next
    }

    if (equal)
        count++
    else {
        if ((repeated_only && count > 1) ||
            (non_repeated_only && count == 1))
                print last > outputfile
        last = $0
```


```
        count = 1
    }
}

END {
    if (do_count)
        printf("%4d %s\n", count, last) > outputfile
    else if ((repeated_only && count > 1) ||
             (non_repeated_only && count == 1))
        print last > outputfile
}
```

Counting Things

The `wc` (word count) utility counts lines, words, and characters in one or more input files. Its usage is:

```
wc [-lwc] [ files ... ]
```

If no files are specified on the command line, `wc` reads its standard input. If there are multiple files, it will also print total counts for all the files. The options and their meanings are:

`-l`

Only count lines.

`-w`

Only count words. A "word" is a contiguous sequence of non-whitespace characters, separated by spaces and/or tabs. Happily, this is the normal way `awk` separates fields in its input data.

`-c`

Only count characters.

Implementing `wc` in `awk` is particularly elegant, since `awk` does a lot of the work for us; it splits lines into words (i.e. fields) and counts them, it counts lines (i.e. records) for us, and it can easily tell us how long a line is.

This version uses the `getopt` library function (see section Processing Command Line Options), and the file transition functions (see section Noting Data File Boundaries).

This version has one major difference from traditional versions of `wc`. Our version always prints the counts in the order lines, words, and characters. Traditional versions note the order of the `-l', `-w', and `-c' options on the command line, and print the counts in that order.

The `BEGIN` rule does the argument processing. The variable `print_total` will be true if more than one file was named on the command line.

```
# wc.awk -- count lines, words, characters
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# May 1993
```


```
# Options:
#    -l    only count lines
#    -w    only count words
#    -c    only count characters
#
# Default is to count lines, words, characters

BEGIN {
    # let getopt print a message about
    # invalid options. we ignore them
    while ((c = getopt(ARGC, ARGV, "lwc")) != -1) {
        if (c == "l")
            do_lines = 1
        else if (c == "w")
            do_words = 1
        else if (c == "c")
            do_chars = 1
    }
    for (i = 1; i < Optind; i++)
        ARGV[i] = ""

    # if no options, do all
    if (! do_lines && ! do_words && ! do_chars)
        do_lines = do_words = do_chars = 1

    print_total = (ARC - i > 2)
}
```

The `beginfile` function is simple; it just resets the counts of lines, words, and characters to zero, and saves the current file name in `fname`.

The `endfile` function adds the current file's numbers to the running totals of lines, words, and characters. It then prints out those numbers for the file that was just read. It relies on `beginfile` to reset the numbers for the following data file.

```
function beginfile(file)
{
    chars = lines = words = 0
    fname = FILENAME
}

function endfile(file)
{
    tchars += chars
    tlines += lines
```


```
    twords += words
    if (do_lines)
        printf "\t%d", lines
    if (do_words)
        printf "\t%d", words
    if (do_chars)
        printf "\t%d", chars
    printf "\t%s\n", fname
}
```

There is one rule that is executed for each line. It adds the length of the record to `chars`. It has to add one, since the newline character separating records (the value of `RS`) is not part of the record itself. `lines` is incremented for each line read, and `words` is incremented by the value of `NF`, the number of "words" on this line.(19)

Finally, the `END` rule simply prints the totals for all the files.

```
# do per line
{
    chars += length($0) + 1    # get newline
    lines++
    words += NF
}

END {
    if (print_total) {
        if (do_lines)
            printf "\t%d", tlines
        if (do_words)
            printf "\t%d", twords
        if (do_chars)
            printf "\t%d", tchars
        print "\ttotal"
    }
}
```

A Grab Bag of awk Programs

This section is a large "grab bag" of miscellaneous programs. We hope you find them both interesting and enjoyable.

Finding Duplicated Words in a Document

A common error when writing large amounts of prose is to accidentally duplicate words. Often you will see this in text as something like "the the program does the following ...." When the text is on-line, often the duplicated words occur at the end of one line and the beginning of another, making them very difficult to spot.

This program, `dupword.awk', scans through a file one line at a time, and looks for adjacent occurrences of the same word. It also saves the last word on a line (in the variable `prev`) for comparison with the first word on the next line.

The first two statements make sure that the line is all lower-case, so that, for example, "The" and "the" compare equal to each other. The second statement removes all non-alphanumeric and non-whitespace characters from the line, so that punctuation does not affect the comparison either. This sometimes leads to reports of duplicated words that really are different, but this is unusual.

```
# dupword -- find duplicate words in text
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# December 1991

{
    $0 = tolower($0)
    gsub(/[^A-Za-z0-9 \t]/, "");
    if ($1 == prev)
        printf("%s:%d: duplicate %s\n",
            FILENAME, FNR, $1)
    for (i = 2; i <= NF; i++)
        if ($i == $(i-1))
            printf("%s:%d: duplicate %s\n",
                FILENAME, FNR, $i)
    prev = $NF
}
```

An Alarm Clock Program

The following program is a simple "alarm clock" program. You give it a time of day, and an optional message. At the given time, it prints the message on the standard output. In addition, you can give it the number of times to repeat the message, and also a delay between repetitions.

This program uses the `gettimeofday` function from section Managing the Time of Day.

All the work is done in the `BEGIN` rule. The first part is argument checking and setting of defaults; the delay, the count, and the message to print. If the user supplied a message, but it does not contain the ASCII BEL character (known as the "alert" character, `\a'), then it is added to the message. (On many systems, printing the ASCII BEL generates some sort of audible alert. Thus, when the alarm goes off, the system calls attention to itself, in case the user is not looking at their computer or terminal.)


```
# alarm -- set an alarm
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# May 1993

# usage: alarm time [ "message" [ count [ delay ] ] ]

BEGIN    \
{
    # Initial argument sanity checking
    usage1 = "usage: alarm time ['message' [count [delay]]]"
    usage2 = sprintf("\t(%s) time ::= hh:mm", ARGV[1])

    if (ARGC < 2) {
        print usage > "/dev/stderr"
        exit 1
    } else if (ARGC == 5) {
        delay = ARGV[4] + 0
        count = ARGV[3] + 0
        message = ARGV[2]
    } else if (ARGC == 4) {
        count = ARGV[3] + 0
        message = ARGV[2]
    } else if (ARGC == 3) {
        message = ARGV[2]
    } else if (ARGV[1] !~ /[0-9]?[0-9]:[0-9][0-9]/) {
        print usage1 > "/dev/stderr"
        print usage2 > "/dev/stderr"
        exit 1
    }

    # set defaults for once we reach the desired time
    if (delay == 0)
        delay = 180    # 3 minutes
    if (count == 0)
        count = 5
    if (message == "")
        message = sprintf("\aIt is now %s!\a", ARGV[1])
    else if (index(message, "\a") == 0)
        message = "\a" message "\a"
```

The next section of code turns the alarm time into hours and minutes, and converts it if necessary to a 24-hour clock. Then it turns that time into a count of the seconds since midnight. Next it turns the current time into a count of seconds since midnight. The difference between the two is how long to wait before setting off the alarm.


```
    # split up dest time
    split(ARGV[1], atime, ":")
    hour = atime[1] + 0    # force numeric
    minute = atime[2] + 0  # force numeric

    # get current broken down time
    gettimeofday(now)

    # if time given is 12-hour hours and it's after that
    # hour, e.g., `alarm 5:30' at 9 a.m. means 5:30 p.m.,
    # then add 12 to real hour
    if (hour < 12 && now["hour"] > hour)
        hour += 12

    # set target time in seconds since midnight
    target = (hour * 60 * 60) + (minute * 60)

    # get current time in seconds since midnight
    current = (now["hour"] * 60 * 60) + \
              (now["minute"] * 60) + now["second"]

    # how long to sleep for
    naptime = target - current
    if (naptime <= 0) {
        print "time is in the past!" > "/dev/stderr"
        exit 1
    }
```

Finally, the program uses the `system` function (see section Built-in Functions for Input/Output) to call the `sleep` utility. The `sleep` utility simply pauses for the given number of seconds. If the exit status is not zero, the program assumes that `sleep` was interrupted, and exits. If `sleep` exited with an OK status (zero), then the program prints the message in a loop, again using `sleep` to delay for however many seconds are necessary.

```
    # zzzzzz..... go away if interrupted
    if (system(sprintf("sleep %d", naptime)) != 0)
        exit 1

    # time to notify!
    command = sprintf("sleep %d", delay)
    for (i = 1; i <= count; i++) {
        print message
        # if sleep command interrupted, go away
        if (system(command) != 0)
            break
```


```
    }

    exit 0
}
```

Transliterating Characters

The system `tr` utility transliterates characters. For example, it is often used to map upper-case letters into lower-case, for further processing.

```
generate data | tr '[A-Z]' '[a-z]' | process data ...
```

You give `tr` two lists of characters enclosed in square brackets. Usually, the lists are quoted to keep the shell from attempting to do a filename expansion.(20) When processing the input, the first character in the first list is replaced with the first character in the second list, the second character in the first list is replaced with the second character in the second list, and so on. If there are more characters in the "from" list than in the "to" list, the last character of the "to" list is used for the remaining characters in the "from" list.

Some time ago, a user proposed to us that we add a transliteration function to `gawk`. Being opposed to "creeping featurism," I wrote the following program to prove that character transliteration could be done with a user-level function. This program is not as complete as the system `tr` utility, but it will do most of the job.

The `translate` program demonstrates one of the few weaknesses of standard `awk`: dealing with individual characters is very painful, requiring repeated use of the `substr`, `index`, and `gsub` built-in functions (see section Built-in Functions for String Manipulation).(21)

There are two functions. The first, `stranslate`, takes three arguments.

`from`
: A list of characters to translate from.

`to`
: A list of characters to translate to.

`target`
: The string to do the translation on.

Associative arrays make the translation part fairly easy. `t_ar` holds the "to" characters, indexed by the "from" characters. Then a simple loop goes through `from`, one character at a time. For each character in `from`, if the character appears in `target`, `gsub` is used to change it to the corresponding `to` character.

The `translate` function simply calls `stranslate` using `$0` as the target. The main program sets two global variables, `FROM` and `TO`, from the command line, and then changes `ARGV` so that `awk` will read from the standard input.

Finally, the processing rule simply calls `translate` for each record.


```
# translate -- do tr like stuff
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# August 1989

# bugs: does not handle things like: tr A-Z a-z, it has
# to be spelled out. However, if `to' is shorter than `from',
# the last character in `to' is used for the rest of `from'.

function stranslate(from, to, target,     lf, lt, t_ar, i, c)
{
    lf = length(from)
    lt = length(to)
    for (i = 1; i <= lt; i++)
        t_ar[substr(from, i, 1)] = substr(to, i, 1)
    if (lt < lf)
        for (; i <= lf; i++)
            t_ar[substr(from, i, 1)] = substr(to, lt, 1)
    for (i = 1; i <= lf; i++) {
        c = substr(from, i, 1)
        if (index(target, c) > 0)
            gsub(c, t_ar[c], target)
    }
    return target
}

function translate(from, to)
{
    return $0 = stranslate(from, to, $0)
}

# main program
BEGIN {
    if (ARGC < 3) {
        print "usage: translate from to" > "/dev/stderr"
        exit
    }
    FROM = ARGV[1]
    TO = ARGV[2]
    ARGC = 2
    ARGV[1] = "-"
}

{
    translate(FROM, TO)
    print
}
```

While it is possible to do character transliteration in a user-level function, it is not necessarily efficie and we started to consider adding a built-in function. However, shortly after writing this program, w learned that the System V Release 4 `awk` had added the `toupper` and `tolower` functions. These functions handle the vast majority of the cases where character transliteration is necessary, and so we chose to simply add those functions to `gawk` as well, and then leave well enough alone.

An obvious improvement to this program would be to set up the `t_ar` array only once, in a `BEGIN` rule. However, this assumes that the "from" and "to" lists will never change throughout the lifetime of the program.

Printing Mailing Labels

Here is a "real world"(22) program. This script reads lists of names and addresses, and generates mailing labels. Each page of labels has 20 labels on it, two across and ten down. The addresses are guaranteed to be no more than five lines of data. Each address is separated from the next by a blank line.

The basic idea is to read 20 labels worth of data. Each line of each label is stored in the `line` array. The single rule takes care of filling the `line` array and printing the page when 20 labels have been read.

The `BEGIN` rule simply sets `RS` to the empty string, so that `awk` will split records at blank lines (see section How Input is Split into Records). It sets `MAXLINES` to 100, since `MAXLINE` is the maximum number of lines on the page (20 * 5 = 100).

Most of the work is done in the `printpage` function. The label lines are stored sequentially in the `line` array. But they have to be printed horizontally; `line[1]` next to `line[6]`, `line[2]` next to `line[7]`, and so on. Two loops are used to accomplish this. The outer loop, controlled by `i`, steps through every 10 lines of data; this is each row of labels. The inner loop, controlled by `j`, goes through the lines within the row. As `j` goes from zero to four, `i+j' is the `j`'th line in the row, and `i+j+5' is the entry next to it. The output ends up looking something like this:

```
line 1          line 6
line 2          line 7
line 3          line 8
line 4          line 9
line 5          line 10
```

As a final note, at lines 21 and 61, an extra blank line is printed, to keep the output lined up on the labels. This is dependent on the particular brand of labels in use when the program was written. You will also note that there are two blank lines at the top and two blank lines at the bottom.

The `END` rule arranges to flush the final page of labels; there may not have been an even multiple of 20 labels in the data.

```
# labels.awk
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# June 1992
```

TOC

```
# Program to print labels.  Each label is 5 lines of data
# that may have blank lines.  The label sheets have 2
# blank lines at the top and 2 at the bottom.

BEGIN    { RS = "" ; MAXLINES = 100 }

function printpage(    i, j)
{
    if (Nlines <= 0)
        return

    printf "\n\n"        # header

    for (i = 1; i <= Nlines; i += 10) {
        if (i == 21 || i == 61)
            print ""
        for (j = 0; j < 5; j++) {
            if (i + j > MAXLINES)
                break
            printf "   %-41s %s\n", line[i+j], line[i+j+5]
        }
        print ""
    }

    printf "\n\n"        # footer

    for (i in line)
        line[i] = ""
}

# main rule
{
    if (Count >= 20) {
        printpage()
        Count = 0
        Nlines = 0
    }
    n = split($0, a, "\n")
    for (i = 1; i <= n; i++)
        line[++Nlines] = a[i]
    for (; i <= 5; i++)
        line[++Nlines] = ""
    Count++
}

END    \
```


```
{
    printpage()
}
```

Generating Word Usage Counts

The following `awk` program prints the number of occurrences of each word in its input. It illustrates the associative nature of `awk` arrays by using strings as subscripts. It also demonstrates the `for x in array' construction. Finally, it shows how `awk` can be used in conjunction with other utility programs to do a useful task of some complexity with a minimum of effort. Some explanations follow the program listing.

```
awk '
# Print list of word frequencies
{
    for (i = 1; i <= NF; i++)
        freq[$i]++
}

END {
    for (word in freq)
        printf "%s\t%d\n", word, freq[word]
}'
```

The first thing to notice about this program is that it has two rules. The first rule, because it has an empty pattern, is executed on every line of the input. It uses `awk`'s field-accessing mechanism (see section Examining Fields) to pick out the individual words from the line, and the built-in variable `NF` (see section Built-in Variables) to know how many fields are available.

For each input word, an element of the array `freq` is incremented to reflect that the word has been seen an additional time.

The second rule, because it has the pattern `END`, is not executed until the input has been exhausted. It prints out the contents of the `freq` table that has been built up inside the first action.

This program has several problems that would prevent it from being useful by itself on real text files:

- Words are detected using the `awk` convention that fields are separated by whitespace and that other characters in the input (except newlines) don't have any special meaning to `awk`. This means that punctuation characters count as part of words.
- The `awk` language considers upper- and lower-case characters to be distinct. Therefore, `bartender' and `Bartender' are not treated as the same word. This is undesirable since, in normal text, words are capitalized if they begin sentences, and a frequency analyzer should not be sensitive to capitalization.
- The output does not come out in any useful order. You're more likely to be interested in which words occur most frequently, or having an alphabetized table of how frequently each word occurs.

The way to solve these problems is to use some of the more advanced features of the `awk` language.

First, we use `tolower` to remove case distinctions. Next, we use `gsub` to remove punctuation characters. Finally, we use the system `sort` utility to process the output of the `awk` script. Here is the new version of the program:

```
# Print list of word frequencies
{
    $0 = tolower($0)    # remove case distinctions
    gsub(/[^a-z0-9_ \t]/, "", $0)  # remove punctuation
    for (i = 1; i <= NF; i++)
        freq[$i]++
}

END {
    for (word in freq)
        printf "%s\t%d\n", word, freq[word]
}
```

Assuming we have saved this program in a file named `` `wordfreq.awk' ``, and that the data is in `` `file1' ``, the following pipeline

```
awk -f wordfreq.awk file1 | sort +1 -nr
```

produces a table of the words appearing in `` `file1' `` in order of decreasing frequency.

The `awk` program suitably massages the data and produces a word frequency table, which is not ordered.

The `awk` script's output is then sorted by the `sort` utility and printed on the terminal. The options given to `sort` in this example specify to sort using the second field of each input line (skipping one field), that the sort keys should be treated as numeric quantities (otherwise `15' would come before `5'), and that the sorting should be done in descending (reverse) order.

We could have even done the `sort` from within the program, by changing the `END` action to:

```
END {
    sort = "sort +1 -nr"
    for (word in freq)
        printf "%s\t%d\n", word, freq[word] | sort
    close(sort)
}
```

You would have to use this way of sorting on systems that do not have true pipes.

See the general operating system documentation for more information on how to use the `sort` program.

Removing Duplicates from Unsorted Text

The `uniq` program (see section Printing Non-duplicated Lines of Text), removes duplicate lines from *sorted* data.

Suppose, however, you need to remove duplicate lines from a data file, but that you wish to preserve the order the lines are in? A good example of this might be a shell history file. The history file keeps a copy of all the commands you have entered, and it is not unusual to repeat a command several times in a row. Occasionally you might wish to compact the history by removing duplicate entries. Yet it is desirable to maintain the order of the original commands.

This simple program does the job. It uses two arrays. The `data` array is indexed by the text of each line. For each line, `data[$0]` is incremented.

If a particular line has not been seen before, then `data[$0]` will be zero. In that case, the text of the line is stored in `lines[count]`. Each element of `lines` is a unique command, and the indices of `lines` indicate the order in which those lines were encountered. The `END` rule simply prints out the lines, in order.

```
# histsort.awk -- compact a shell history file
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# May 1993

# Thanks to Byron Rakitzis for the general idea
{
    if (data[$0]++ == 0)
        lines[++count] = $0
}

END {
    for (i = 1; i <= count; i++)
        print lines[i]
}
```

This program also provides a foundation for generating other useful information. For example, using the following `print` satement in the `END` rule would indicate how often a particular command was used.

```
print data[lines[i]], lines[i]
```

This works because `data[$0]` was incremented each time a line was seen.

Extracting Programs from Texinfo Source Files

Both this chapter and the previous chapter (section A Library of `awk` Functions), present a large number of `awk` programs. If you wish to experiment with these programs, it is tedious to have to type them in by hand. Here we present a program that can extract parts of a Texinfo input file into separate files.

This book is written in Texinfo, the GNU project's document formatting language. A single Texinfo source file can be used to produce both printed and on-line documentation. Texinfo is fully documented in Texinfo--The GNU Documentation Format, available from the Free Software Foundation.

For our purposes, it is enough to know three things about Texinfo input files.

- The "at" symbol, `@', is special in Texinfo, much like `\' in C or `awk'. Literal `@' symbols are represented in Texinfo source files as `@@'.
- Comments start with either `@c' or `@comment'. The file extraction program will work by using special comments that start at the beginning of a line.
- Example text that should not be split across a page boundary is bracketed between lines containing `@group' and `@end group' commands.

The following program, `extract.awk', reads through a Texinfo source file, and does two things, based on the special comments. Upon seeing `@c system ...', it runs a command, by extracting the command text from the control line and passing it on to the `system' function (see section Built-in Functions for Input/Output). Upon seeing `@c file filename', each subsequent line is sent to the file filename, until `@c endfile' is encountered. The rules in `extract.awk' will match either `@c' or `@comment' by letting the `omment' part be optional. Lines containing `@group' and `@end group' are simply removed. `extract.awk' uses the `join' library function (see section Merging an Array Into a String).

The example programs in the on-line Texinfo source for AWK Language Programming (`gawk.texi') have all been bracketed inside `file', and `endfile' lines. The `gawk' distribution uses a copy of `extract.awk' to extract the sample programs and install many of them in a standard directory, where `gawk' can find them.

`extract.awk' begins by setting `IGNORECASE' to one, so that mixed upper-case and lower-case letters in the directives won't matter.

The first rule handles calling `system', checking that a command was given (`NF' is at least three), and also checking that the command exited with a zero exit status, signifying OK.

```
# extract.awk -- extract files and run programs
#                from texinfo files
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# May 1993

BEGIN    { IGNORECASE = 1 }

/^@c(omment)?[ \t]+system/    \
{
    if (NF < 3) {
        e = (FILENAME ":" FNR)
        e = (e  ": badly formed `system' line")
        print e > "/dev/stderr"
        next
```


```
    }
    $1 = ""
    $2 = ""
    stat = system($0)
    if (stat != 0) {
        e = (FILENAME ":" FNR)
        e = (e ": warning: system returned " stat)
        print e > "/dev/stderr"
    }
}
```

The variable `e` is used so that the function fits nicely on the page.

The second rule handles moving data into files. It verifies that a file name was given in the directive. If the file named is not the current file, then the current file is closed. This means that an `@c endfile' was not given for that file. (We should probably print a diagnostic in this case, although at the moment we do not.)

The `for' loop does the work. It reads lines using `getline` (see section Explicit Input with `getline`). For an unexpected end of file, it calls the `unexpected_eof` function. If the line is an "endfile" line, then it breaks out of the loop. If the line is an `@group' or `@end group' line, then it ignores it, and goes on to the next line.

Most of the work is in the following few lines. If the line has no `@' symbols, it can be printed directly. Otherwise, each leading `@' must be stripped off.

To remove the `@' symbols, the line is split into separate elements of the array `a`, using the `split` function (see section Built-in Functions for String Manipulation). Each element of `a` that is empty indicates two successive `@' symbols in the original line. For each two empty elements (`@@' in the original file), we have to add back in a single `@' symbol.

When the processing of the array is finished, `join` is called with the value of SUBSEP, to rejoin the pieces back into a single line. That line is then printed to the output file.

```
/^@c(omment)?[ \t]+file/    \
{
    if (NF != 3) {
        e = (FILENAME ":" FNR ": badly formed `file' line")
        print e > "/dev/stderr"
        next
    }
    if ($3 != curfile) {
        if (curfile != "")
            close(curfile)
        curfile = $3
    }
```


```
    for (;;) {
        if ((getline line) <= 0)
            unexpected_eof()
        if (line ~ /^@c(omment)?[ \t]+endfile/)
            break
        else if (line ~ /^@(end[ \t]+)?group/)
            continue
        if (index(line, "@") == 0) {
            print line > curfile
            continue
        }
        n = split(line, a, "@")
        # if a[1] == "", means leading @,
        # don't add one back in.
        for (i = 2; i <= n; i++) {
            if (a[i] == "") { # was an @@
                a[i] = "@"
                if (a[i+1] == "")
                    i++
            }
        }
        print join(a, 1, n, SUBSEP) > curfile
    }
}
```

An important thing to note is the use of the `>' redirection. Output done with `>' only opens the file once; it stays open and subsequent output is appended to the file (see section Redirecting Output of `print' and `printf'). This allows us to easily mix program text and explanatory prose for the same sample source file (as has been done here!) without any hassle. The file is only closed when a new data file name is encountered, or at the end of the input file.

Finally, the function `unexpected_eof' prints an appropriate error message and then exits.

The `END' rule handles the final cleanup, closing the open file.

```
function unexpected_eof()
{
    printf("%s:%d: unexpected EOF or error\n", \
        FILENAME, FNR) > "/dev/stderr"
    exit 1
}

END {
    if (curfile)
        close(curfile)
}
```

A Simple Stream Editor

The `sed` utility is a "stream editor," a program that reads a stream of data, makes changes to it, and passes the modified data on. It is often used to make global changes to a large file, or to a stream of data generated by a pipeline of commands.

While `sed` is a complicated program in its own right, its most common use is to perform global substitutions in the middle of a pipeline:

```
command1 < orig.data | sed 's/old/new/g' | command2 > result
```

Here, the `s/old/new/g' tells `sed` to look for the regexp `old' on each input line, and replace it with the text `new', globally (i.e. all the occurrences on a line). This is similar to `awk`'s `gsub` function (see section Built-in Functions for String Manipulation).

The following program, `` `awksed.awk' ``, accepts at least two command line arguments; the pattern to look for and the text to replace it with. Any additional arguments are treated as data file names to process. If none are provided, the standard input is used.

```
# awksed.awk -- do s/foo/bar/g using just print
#    Thanks to Michael Brennan for the idea

# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# August 1995

function usage()
{
    print "usage: awksed pat repl [files...]" > "/dev/stderr"
    exit 1
}

BEGIN {
    # validate arguments
    if (ARGC < 3)
        usage()

    RS = ARGV[1]
    ORS = ARGV[2]

    # don't use arguments as files
    ARGV[1] = ARGV[2] = ""
}

# look ma, no hands!
{
```


```
    if (RT == "")
        printf "%s", $0
    else
        print
}
```

The program relies on `gawk`'s ability to have `RS` be a regexp and on the setting of `RT` to the actual text that terminated the record (see section How Input is Split into Records).

The idea is to have `RS` be the pattern to look for. `gawk` will automatically set `$0` to the text between matches of the pattern. This is text that we wish to keep, unmodified. Then, by setting `ORS` to the replacement text, a simple `print` statement will output the text we wish to keep, followed by the replacement text.

There is one wrinkle to this scheme, which is what to do if the last record doesn't end with text that matches `RS`? Using a `print` statement unconditionally prints the replacement text, which is not correct.

However, if the file did not end in text that matches `RS`, `RT` will be set to the null string. In this case, we can print `$0` using `printf` (see section Using `printf` Statements for Fancier Printing).

The `BEGIN` rule handles the setup, checking for the right number of arguments, and calling `usage` if there is a problem. Then it sets `RS` and `ORS` from the command line arguments, and sets `ARGV[1]` and `ARGV[2]` to the null string, so that they will not be treated as file names (see section Using ARGC and ARGV).

The `usage` function prints an error message and exits.

Finally, the single rule handles the printing scheme outlined above, using `print` or `printf` as appropriate, depending upon the value of `RT`.

An Easy Way to Use Library Functions

Using library functions in `awk` can be very beneficial. It encourages code re-use and the writing of general functions. Programs are smaller, and therefore clearer. However, using library functions is only easy when writing `awk` programs; it is painful when running them, requiring multiple `-f' options. If `gawk` is unavailable, then so too is the `AWKPATH` environment variable and the ability to put `awk` functions into a library directory (see section Command Line Options).

It would be nice to be able to write programs like so:

```
# library functions
@include getopt.awk
@include join.awk
...

# main program
BEGIN {
```


```
    while ((c = getopt(ARGC, ARGV, "a:b:cde")) != -1)
        ...
    ...
}
```

The following program, `igawk.sh', provides this service. It simulates `gawk`'s searching of the `AWKPATH` variable, and also allows nested includes; i.e. a file that has been included with `@include' can contain further `@include' statements. `igawk` will make an effort to only include files once, so that nested includes don't accidentally include a library function twice.

`igawk` should behave externally just like `gawk`. This means it should accept all of `gawk`'s command line arguments, including the ability to have multiple source files specified via `-f', and the ability to mix command line and library source files.

The program is written using the POSIX Shell (`sh`) command language. The way the program works is as follows:

1. Loop through the arguments, saving anything that doesn't represent `awk` source code for later, when the expanded program is run.
2. For any arguments that do represent `awk` text, put the arguments into a temporary file that will be expanded. There are two cases.
    1. Literal text, provided with `--source' or `--source='. This text is just echoed directly. The `echo` program will automatically supply a trailing newline.
    2. File names provided with `-f'. We use a neat trick, and echo `@include filename' into the temporary file. Since the file inclusion program will work the way `gawk` does, this will get the text of the file included into the program at the correct point.
3. Run an `awk` program (naturally) over the temporary file to expand `@include' statements. The expanded program is placed in a second temporary file.
4. Run the expanded program with `gawk` and any other original command line arguments that the user supplied (such as the data file names).

The initial part of the program turns on shell tracing if the first argument was `debug'. Otherwise, a shell `trap` statement arranges to clean up any temporary files on program exit or upon an interrupt.

The next part loops through all the command line arguments. There are several cases of interest.

`--`

This ends the arguments to `igawk`. Anything else should be passed on to the user's `awk` program without being evaluated.

`-W`

This indicates that the next option is specific to `gawk`. To make argument processing easier, the `-W' is appended to the front of the remaining arguments and the loop continues. (This is an `sh` programming trick. Don't worry about it if you are not familiar with `sh`.)

`-v`

`-F`

These are saved and passed on to `gawk`.

`-f`
`--file`
`--file=`
`-Wfile=`

The file name is saved to the temporary file `` `/tmp/ig.s.$$' `` with an `@include' statement. The `sed` utility is used to remove the leading option part of the argument (e.g., `--file=').

`--source`
`--source=`
`-Wsource=`

The source text is echoed into `` `/tmp/ig.s.$$' ``.

`--version`
`--version`
`-Wversion`

`igawk` prints its version number, and runs `gawk --version' to get the `gawk` version information, and then exits.

If none of `-f', `--file', `-Wfile', `--source', or `-Wsource', were supplied, then the first non-option argument should be the `awk` program. If there are no command line arguments left, `igawk` prints an error message and exits. Otherwise, the first argument is echoed into `` `/tmp/ig.s.$$' ``.

In any case, after the arguments have been processed, `` `/tmp/ig.s.$$' `` contains the complete text of the original `awk` program.

The `$$' in `sh` represents the current process ID number. It is often used in shell programs to generate unique temporary file names. This allows multiple users to run `igawk` without worrying that the temporary file names will clash.

Here's the program:

```
#! /bin/sh

# igawk -- like gawk but do @include processing
# Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
# July 1993

if [ "$1" = debug ]
then
    set -x
    shift
else
    # cleanup on exit, hangup, interrupt, quit, termination
    trap 'rm -f /tmp/ig.[se].$$' 0 1 2 3 15
fi
```


```
while [ $# -ne 0 ] # loop over arguments
do
    case $1 in
    --)     shift; break;;

    -W)     shift
            set -- -W"$@"
            continue;;

    -[vF])  opts="$opts $1 '$2'"
            shift;;

    -[vF]*) opts="$opts '$1'" ;;

    -f)     echo @include "$2" >> /tmp/ig.s.$$
            shift;;

    -f*)    f=`echo "$1" | sed 's/-f//'`
            echo @include "$f" >> /tmp/ig.s.$$ ;;

    -?file=*)    # -Wfile or --file
            f=`echo "$1" | sed 's/-.file=//'`
            echo @include "$f" >> /tmp/ig.s.$$ ;;

    -?file)    # get arg, $2
            echo @include "$2" >> /tmp/ig.s.$$
            shift;;

    -?source=*)    # -Wsource or --source
            t=`echo "$1" | sed 's/-.source=//'`
            echo "$t" >> /tmp/ig.s.$$ ;;

    -?source)  # get arg, $2
            echo "$2" >> /tmp/ig.s.$$
            shift;;

    -?version)
            echo igawk: version 1.0 1>&2
            gawk --version
            exit 0 ;;

    -[W-]*)    opts="$opts '$1'" ;;

    *)      break;;
    esac
    shift
```


```
done

if [ ! -s /tmp/ig.s.$$ ]
then
    if [ -z "$1" ]
    then
         echo igawk: no program! 1>&2
         exit 1
    else
        echo "$1" > /tmp/ig.s.$$
        shift
    fi
fi

# at this point, /tmp/ig.s.$$ has the program
```

The `awk` program to process `@include' directives reads through the program, one line at a time using `getline` (see section Explicit Input with `getline`). The input file names and `@include' statements are managed using a stack. As each `@include' is encountered, the current file name is "pushed" onto the stack, and the file named in the `@include' directive becomes the current file name. As each file is finished, the stack is "popped," and the previous input file becomes the current input file again. The process is started by making the original file the first one on the stack.

The `pathto` function does the work of finding the full path to a file. It simulates `gawk`'s behavior when searching the `AWKPATH` environment variable (see section The `AWKPATH` Environment Variable). If a file name has a `/' in it, no path search is done. Otherwise, the file name is concatenated with the name of each directory in the path, and an attempt is made to open the generated file name. The only way in `awk` to test if a file can be read is to go ahead and try to read it with `getline`; that is what `pathto` does. If the file can be read, it is closed, and the file name is returned.

```
gawk -- '
# process @include directives

function pathto(file,    i, t, junk)
{
    if (index(file, "/") != 0)
        return file

    for (i = 1; i <= ndirs; i++) {
        t = (pathlist[i] "/" file)
        if ((getline junk < t) > 0) {
            # found it
            close(t)
            return t
        }
    }
```


```
    return ""
}
```

The main program is contained inside one `BEGIN` rule. The first thing it does is set up the `pathlist` array that `pathto` uses. After splitting the path on `:', null elements are replaced with `"."', which represents the current directory.

```
BEGIN {
    path = ENVIRON["AWKPATH"]
    ndirs = split(path, pathlist, ":")
    for (i = 1; i <= ndirs; i++) {
        if (pathlist[i] == "")
            pathlist[i] = "."
    }
```

The stack is initialized with `ARGV[1]`, which will be `` `/tmp/ig.s.$$' ``. The main loop comes next. Input lines are read in succession. Lines that do not start with `@include' are printed verbatim.

If the line does start with `@include', the file name is in `$2`. `pathto` is called to generate the full path. If it could not, then we print an error message and continue.

The next thing to check is if the file has been included already. The `processed` array is indexed by the full file name of each included file, and it tracks this information for us. If the file has been seen, a warning message is printed. Otherwise, the new file name is pushed onto the stack and processing continues.

Finally, when `getline` encounters the end of the input file, the file is closed and the stack is popped. When `stackptr` is less than zero, the program is done.

```
    stackptr = 0
    input[stackptr] = ARGV[1] # ARGV[1] is first file

    for (; stackptr >= 0; stackptr--) {
        while ((getline < input[stackptr]) > 0) {
            if (tolower($1) != "@include") {
                print
                continue
            }
            fpath = pathto($2)
            if (fpath == "") {
                printf("igawk:%s:%d: cannot find %s\n", \
                    input[stackptr], FNR, $2) > "/dev/stderr"
                continue
            }
            if (! (fpath in processed)) {
                processed[fpath] = input[stackptr]
                input[++stackptr] = fpath
```


```
            } else
                print $2, "included in", input[stackptr], \
                    "already included in", \
                    processed[fpath] > "/dev/stderr"
        }
        close(input[stackptr])
    }
}' /tmp/ig.s.$$ > /tmp/ig.e.$$
```

The last step is to call `gawk` with the expanded program and the original options and command line arguments that the user supplied. `gawk`'s exit status is passed back on to `igawk`'s calling program.

```
eval gawk -f /tmp/ig.e.$$ $opts -- "$@"

exit $?
```

This version of `igawk` represents my third attempt at this program. There are three key simplifications that made the program work better.

1. Using `@include' even for the files named with `-f' makes building the initial collected `awk` program much simpler; all the `@include' processing can be done once.
2. The `pathto` function doesn't try to save the line read with `getline` when testing for the file's accessibility. Trying to save this line for use with the main program complicates things considerably.
3. Using a `getline` loop in the `BEGIN` rule does it all in one place. It is not necessary to call out to a separate loop for processing nested `@include' statements.

Also, this program illustrates that it is often worthwhile to combine `sh` and `awk` programming together. You can usually accomplish quite a lot, without having to resort to low-level programming in C or C++, and it is frequently easier to do certain kinds of string and argument manipulation using the shell than it is in `awk`.

Finally, `igawk` shows that it is not always necessary to add new features to a program; they can often be layered on top. With `igawk`, there is no real reason to build `@include' processing into `gawk` itself.

As an additional example of this, consider the idea of having two files in a directory in the search path.

`default.awk'`
: This file would contain a set of default library functions, such as `getopt` and `assert`.

`site.awk'`
: This file would contain library functions that are specific to a site or installation, i.e. locally developed functions. Having a separate file allows `default.awk' to change with new `gawk` releases, without requiring the system administrator to update it each time by adding the local functions.

One user suggested that `gawk` be modified to automatically read these files upon startup. Instead, it would be very simple to modify `igawk` to do this. Since `igawk` can process nested `@include' directives, `default.awk' could simply contain `@include' statements for the desired library

TOC

functions.

Go to the previous, next section.

Go to the previous, next section.

The Evolution of the awk Language

This book describes the GNU implementation of `awk`, which follows the POSIX specification. Many `awk` users are only familiar with the original `awk` implementation in Version 7 Unix. (This implementation was the basis for `awk` in Berkeley Unix, through 4.3--Reno. The 4.4 release of Berkeley Unix uses `gawk` 2.15.2 for its version of `awk`.) This chapter briefly describes the evolution of the `awk` language, with cross references to other parts of the book where you can find more information.

Major Changes between V7 and SVR3.1

The `awk` language evolved considerably between the release of Version 7 Unix (1978) and the new version first made generally available in System V Release 3.1 (1987). This section summarizes the changes, with cross-references to further details.

- The requirement for `;' to separate rules on a line (see section `awk` Statements Versus Lines).
- User-defined functions, and the `return` statement (see section User-defined Functions).
- The `delete` statement (see section The `delete` Statement).
- The `do-while` statement (see section The `do-while` Statement).
- The built-in functions `atan2`, `cos`, `sin`, `rand` and `srand` (see section Numeric Built-in Functions).
- The built-in functions `gsub`, `sub`, and `match` (see section Built-in Functions for String Manipulation).
- The built-in functions `close`, and `system` (see section Built-in Functions for Input/Output).
- The `ARGC`, `ARGV`, `FNR`, `RLENGTH`, `RSTART`, and `SUBSEP` built-in variables (see section Built-in Variables).
- The conditional expression using the ternary operator `?:' (see section Conditional Expressions).
- The exponentiation operator `^' (see section Arithmetic Operators) and its assignment operator form `^=' (see section Assignment Expressions).
- C-compatible operator precedence, which breaks some old `awk` programs (see section Operator Precedence (How Operators Nest)).
- Regexps as the value of `FS` (see section Specifying How Fields are Separated), and as the third argument to the `split` function (see section Built-in Functions for String Manipulation).
- Dynamic regexps as operands of the `~' and `!~' operators (see section How to Use Regular Expressions).
- The escape sequences `\b', `\f', and `\r' (see section Escape Sequences). (Some vendors have updated their old versions of `awk` to recognize `\r', `\b', and `\f', but this is not something you can

rely on.)

- Redirection of input for the `getline` function (see section Explicit Input with `getline`).
- Multiple `BEGIN` and `END` rules (see section The `BEGIN` and `END` Special Patterns).
- Multi-dimensional arrays (see section Multi-dimensional Arrays).

Changes between SVR3.1 and SVR4

The System V Release 4 version of Unix `awk` added these features (some of which originated in `gawk`):

- The `ENVIRON` variable (see section Built-in Variables).
- Multiple `-f' options on the command line (see section Command Line Options).
- The `-v' option for assigning variables before program execution begins (see section Command Line Options).
- The `--' option for terminating command line options.
- The `\a', `\v', and `\x' escape sequences (see section Escape Sequences).
- A defined return value for the `srand` built-in function (see section Numeric Built-in Functions).
- The `toupper` and `tolower` built-in string functions for case translation (see section Built-in Functions for String Manipulation).
- A cleaner specification for the `%c' format-control letter in the `printf` function (see section Format-Control Letters).
- The ability to dynamically pass the field width and precision (`"%*.*d"`) in the argument list of the `printf` function (see section Format-Control Letters).
- The use of regexp constants such as `/foo/` as expressions, where they are equivalent to using the matching operator, as in `$0 ~ /foo/' (see section Using Regular Expression Constants).

Changes between SVR4 and POSIX `awk`

The POSIX Command Language and Utilities standard for `awk` introduced the following changes into the language:

- The use of `-W' for implementation-specific options.
- The use of `CONVFMT` for controlling the conversion of numbers to strings (see section Conversion of Strings and Numbers).
- The concept of a numeric string, and tighter comparison rules to go with it (see section Variable Typing and Comparison Expressions).
- More complete documentation of many of the previously undocumented features of the language.

The following common extensions are not permitted by the POSIX standard:

- `\x` escape sequences are not recognized (see section Escape Sequences).

- The synonym `func` for the keyword `function` is not recognized (see section Function Definition Syntax).
- The operators `**' and `**=' cannot be used in place of `^' and `^=' (see section Arithmetic Operators, and also see section Assignment Expressions).
- Specifying `-Ft' on the command line does not set the value of `FS` to be a single tab character (see section Specifying How Fields are Separated).
- The `fflush` built-in function is not supported (see section Built-in Functions for Input/Output).

Extensions in the AT&T Bell Laboratories awk

Brian Kernighan, one of the original designers of Unix `awk`, has made his version available via anonymous `ftp` (see section Other Freely Available `awk` Implementations). This section describes extensions in his version of `awk` that are not in POSIX `awk`.

- The `-mf=NNN' and `-mr=NNN' command line options to set the maximum number of fields, and the maximum record size, respectively (see section Command Line Options).
- The `fflush` built-in function for flushing buffered output (see section Built-in Functions for Input/Output).

Extensions in gawk Not in POSIX awk

The GNU implementation, `gawk`, adds a number of features. This sections lists them in the order they were added to `gawk`. They can all be disabled with either the `--traditional' or `--posix' options (see section Command Line Options).

Version 2.10 of `gawk` introduced these features:

- The `AWKPATH` environment variable for specifying a path search for the `-f' command line option (see section Command Line Options).
- The `IGNORECASE` variable and its effects (see section Case-sensitivity in Matching).
- The `/dev/stdin', `/dev/stdout', `/dev/stderr', and `/dev/fd/n' file name interpretation (see section Special File Names in `gawk`).

Version 2.13 of `gawk` introduced these features:

- The `FIELDWIDTHS` variable and its effects (see section Reading Fixed-width Data).
- The `systime` and `strftime` built-in functions for obtaining and printing time stamps (see section Functions for Dealing with Time Stamps).
- The `-W lint' option to provide source code and run time error and portability checking (see section Command Line Options).
- The `-W compat' option to turn off these extensions (see section Command Line Options).
- The `-W posix' option for full POSIX compliance (see section Command Line Options).

Version 2.14 of `gawk` introduced these features:

- The `next file` statement for skipping to the next data file (see section The `nextfile` Statement).

Version 2.15 of `gawk` introduced these features:

- The `ARGIND` variable, that tracks the movement of `FILENAME` through `ARGV` (see section Built-in Variables).
- The `ERRNO` variable, that contains the system error message when `getline` returns -1, or when `close` fails (see section Built-in Variables).
- The ability to use GNU-style long named options that start with `--' (see section Command Line Options).
- The `--source' option for mixing command line and library file source code (see section Command Line Options).
- The `/dev/pid', `/dev/ppid', `/dev/pgrpid', and `/dev/user' file name interpretation (see section Special File Names in `gawk`).

Version 3.0 of `gawk` introduced these features:

- The `next file` statement became `nextfile` (see section The `nextfile` Statement).
- The `--lint-old' option to warn about constructs that are not available in the original Version 7 Unix version of `awk` (see section Major Changes between V7 and SVR3.1).
- The `--traditional' option was added as a better name for `--compat' (see section Command Line Options).
- The ability for `FS` to be a null string, and for the third argument to `split` to be the null string (see section Making Each Character a Separate Field).
- The ability for `RS` to be a regexp (see section How Input is Split into Records).
- The `RT` variable (see section How Input is Split into Records).
- The `gensub` function for more powerful text manipulation (see section Built-in Functions for String Manipulation).
- The `strftime` function acquired a default time format, allowing it to be called with no arguments (see section Functions for Dealing with Time Stamps).
- Full support for both POSIX and GNU regexps (see section Regular Expressions).
- The `--re-interval' option to provide interval expressions in regexps (see section Regular Expression Operators).
- `IGNORECASE` changed, now applying to string comparison as well as regexp operations (see section Case-sensitivity in Matching).
- The `-m' option and the `fflush` function from the Bell Labs research version of `awk` (see section Command Line Options; also see section Built-in Functions for Input/Output).
- The use of GNU Autoconf to control the configuration process (see section Compiling `gawk` for

Unix).

- Amiga support (see section Installing gawk on an Amiga).

Go to the previous, next section.

Go to the previous, next section.

gawk Summary

This appendix provides a brief summary of the `gawk` command line and the `awk` language. It is designed to serve as "quick reference." It is therefore terse, but complete.

Command Line Options Summary

The command line consists of options to `gawk` itself, the `awk` program text (if not supplied via the `-f' option), and values to be made available in the `ARGC` and `ARGV` predefined `awk` variables:

```
gawk [POSIX or GNU style options] -f source-file [--] file ...
gawk [POSIX or GNU style options] [--] 'program' file ...
```

The options that `gawk` accepts are:

`-F fs`

`--field-separator fs`

Use fs for the input field separator (the value of the `FS` predefined variable).

`-f program-file`

`--file program-file`

Read the `awk` program source from the file program-file, instead of from the first command line argument.

`-mf=NNN`

`-mr=NNN`

The `f' flag sets the maximum number of fields, and the `r' flag sets the maximum record size. These options are ignored by `gawk`, since `gawk` has no predefined limits; they are only for compatibility with the Bell Labs research version of Unix `awk`.

`-v var=val`

`--assign var=val`

Assign the variable var the value val before program execution begins.

`-W traditional`

`-W compat`

`--traditional`

`--compat`

Use compatibility mode, in which `gawk` extensions are turned off.

`-W copyleft`

`-W copyright`

`--copyleft`

`--copyright`

Print the short version of the General Public License on the error output. This option may disappear in a future version of `gawk`.

`-W help`

`-W usage`

`--help`

`--usage`

Print a relatively short summary of the available options on the error output.

`-W lint`

`--lint`

Give warnings about dubious or non-portable `awk` constructs.

`-W lint-old`

`--lint-old`

Warn about constructs that are not available in the original Version 7 Unix version of `awk`.

`-W posix`

`--posix`

Use POSIX compatibility mode, in which `gawk` extensions are turned off and additional restrictions apply.

`-W re-interval`

`--re-interval`

Allow interval expressions (see section Regular Expression Operators), in regexps.

`-W source=program-text`

`--source program-text`

Use program-text as `awk` program source code. This option allows mixing command line source code with source code from files, and is particularly useful for mixing command line programs with library functions.

`-W version`

`--version`

Print version information for this particular copy of `gawk` on the error output.

`--`

Signal the end of options. This is useful to allow further arguments to the `awk` program itself to start with a `-'. This is mainly for consistency with POSIX argument parsing conventions.

Any other options are flagged as invalid, but are otherwise ignored. See section Command Line Options, for more details.

Language Summary

An awk program consists of a sequence of zero or more pattern-action statements and optional function definitions. One or the other of the pattern and action may be omitted.

```
pattern    { action statements }
pattern
          { action statements }

function name(parameter list)     { action statements }
```

gawk first reads the program source from the program-file(s), if specified, or from the first non-option argument on the command line. The `-f' option may be used multiple times on the command line. gawk reads the program text from all the program-file files, effectively concatenating them in the order they are specified. This is useful for building libraries of awk functions, without having to include them in each new awk program that uses them. To use a library function in a file from a program typed in on the command line, specify `--source 'program", and type your program in between the single quotes. See section Command Line Options.

The environment variable AWKPATH specifies a search path to use when finding source files named with the `-f' option. The default path, which is `.:/usr/local/share/awk'(23) is used if AWKPATH is not set. If a file name given to the `-f' option contains a `/' character, no path search is performed. See section The AWKPATH Environment Variable.

gawk compiles the program into an internal form, and then proceeds to read each file named in the ARGV array. The initial values of ARGV come from the command line arguments. If there are no files named on the command line, gawk reads the standard input.

If a "file" named on the command line has the form `var=val', it is treated as a variable assignment: the variable var is assigned the value val. If any of the files have a value that is the null string, that element in the list is skipped.

For each record in the input, gawk tests to see if it matches any pattern in the awk program. For each pattern that the record matches, the associated action is executed.

Variables and Fields

awk variables are not declared; they come into existence when they are first used. Their values are either floating-point numbers or strings. awk also has one-dimensional arrays; multiple-dimensional arrays may be simulated. There are several predefined variables that awk sets as a program runs; these are summarized below.

Fields

As each input line is read, `gawk` splits the line into fields, using the value of the `FS` variable as the field separator. If `FS` is a single character, fields are separated by that character. Otherwise, `FS` is expected to be a full regular expression. In the special case that `FS` is a single space, fields are separated by runs of spaces and/or tabs. If `FS` is the null string (`""`), then each individual character in the record becomes a separate field. Note that the value of `IGNORECASE` (see section Case-sensitivity in Matching) also affects how fields are split when `FS` is a regular expression.

Each field in the input line may be referenced by its position, `$1`, `$2`, and so on. `$0` is the whole line. The value of a field may be assigned to as well. Field numbers need not be constants:

```
n = 5
print $n
```

prints the fifth field in the input line. The variable `NF` is set to the total number of fields in the input line.

References to non-existent fields (i.e. fields after `$NF`) return the null string. However, assigning to a non-existent field (e.g., `$(NF+2) = 5`) increases the value of `NF`, creates any intervening fields with the null string as their value, and causes the value of `$0` to be recomputed, with the fields being separated by the value of `OFS`. See section Reading Input Files.

Built-in Variables

`gawk`'s built-in variables are:

`ARGC`
: The number of elements in `ARGV`. See below for what is actually included in `ARGV`.

`ARGIND`
: The index in `ARGV` of the current file being processed. When `gawk` is processing the input data files, it is always true that `FILENAME == ARGV[ARGIND]'.

`ARGV`
: The array of command line arguments. The array is indexed from zero to `ARGC - 1`. Dynamically changing `ARGC` and the contents of `ARGV` can control the files used for data. A null-valued element in `ARGV` is ignored. `ARGV` does not include the options to `awk` or the text of the `awk` program itself.

`CONVFMT`
: The conversion format to use when converting numbers to strings.

`FIELDWIDTHS`
: A space separated list of numbers describing the fixed-width input data.

`ENVIRON`
: An array of environment variable values. The array is indexed by variable name, each element being the value of that variable. Thus, the environment variable `HOME` is `ENVIRON["HOME"]`. One possible value might be `` `/home/arnold' ``.

Changing this array does not affect the environment seen by programs which `gawk` spawns via redirection or the `system` function. (This may change in a future version of `gawk`.)

Some operating systems do not have environment variables. The `ENVIRON` array is empty when running on these systems.

`ERRNO`

The system error message when an error occurs using `getline` or `close`.

`FILENAME`

The name of the current input file. If no files are specified on the command line, the value of `FILENAME` is the null string.

`FNR`

The input record number in the current input file.

`FS`

The input field separator, a space by default.

`IGNORECASE`

The case-sensitivity flag for string comparisons and regular expression operations. If `IGNORECASE` has a non-zero value, then pattern matching in rules, record separating with `RS`, field splitting with `FS`, regular expression matching with `~' and `!~', and the `gensub`, `gsub`, `index`, `match`, `split` and `sub` built-in functions all ignore case when doing regular expression operations, and all string comparisons are done ignoring case.

`NF`

The number of fields in the current input record.

`NR`

The total number of input records seen so far.

`OFMT`

The output format for numbers for the `print` statement, `"%.6g"` by default.

`OFS`

The output field separator, a space by default.

`ORS`

The output record separator, by default a newline.

`RS`

The input record separator, by default a newline. If `RS` is set to the null string, then records are separated by blank lines. When `RS` is set to the null string, then the newline character always acts as a field separator, in addition to whatever value `FS` may have. If `RS` is set to a multi-character string, it denotes a regexp; input text matching the regexp separates records.

`RT`

The input text that matched the text denoted by `RS`, the record separator.

`RSTART`

The index of the first character last matched by `match`; zero if no match.

RLENGTH
: The length of the string last matched by `match`; -1 if no match.

SUBSEP
: The string used to separate multiple subscripts in array elements, by default `"\034"`.

See section Built-in Variables, for more information.

Arrays

Arrays are subscripted with an expression between square brackets (`[' and `]'). Array subscripts are *always* strings; numbers are converted to strings as necessary, following the standard conversion rules (see section Conversion of Strings and Numbers).

If you use multiple expressions separated by commas inside the square brackets, then the array subscript is a string consisting of the concatenation of the individual subscript values, converted to strings, separated by the subscript separator (the value of SUBSEP).

The special operator `in` may be used in a conditional context to see if an array has an index consisting of a particular value.

```
if (val in array)
        print array[val]
```

If the array has multiple subscripts, use `(i, j, ...) in array' to test for existence of an element.

The `in` construct may also be used in a `for` loop to iterate over all the elements of an array. See section Scanning All Elements of an Array.

You can remove an element from an array using the `delete` statement.

You can clear an entire array using `delete array'.

See section Arrays in `awk`.

Data Types

The value of an `awk` expression is always either a number or a string.

Some contexts (such as arithmetic operators) require numeric values. They convert strings to numbers by interpreting the text of the string as a number. If the string does not look like a number, it converts to zero.

Other contexts (such as concatenation) require string values. They convert numbers to strings by effectively printing them with `sprintf`. See section Conversion of Strings and Numbers, for the details.

To force conversion of a string value to a number, simply add zero to it. If the value you start with is already a number, this does not change it.

To force conversion of a numeric value to a string, concatenate it with the null string.

Comparisons are done numerically if both operands are numeric, or if one is numeric and the other is a numeric string. Otherwise one or both operands are converted to strings and a string comparison is performed. Fields, `getline` input, `FILENAME`, `ARGV` elements, `ENVIRON` elements and the elements of an array created by `split` are the only items that can be numeric strings. String constants, such as `"3.1415927"` are not numeric strings, they are string constants. The full rules for comparisons are described in section Variable Typing and Comparison Expressions.

Uninitialized variables have the string value `""` (the null, or empty, string). In contexts where a number is required, this is equivalent to zero.

See section Variables, for more information on variable naming and initialization; see section Conversion of Strings and Numbers, for more information on how variable values are interpreted.

Patterns

An `awk` program is mostly composed of rules, each consisting of a pattern followed by an action. The action is enclosed in `{' and `}'. Either the pattern may be missing, or the action may be missing, but not both. If the pattern is missing, the action is executed for every input record. A missing action is equivalent to `{ print }', which prints the entire line.

Comments begin with the `#' character, and continue until the end of the line. Blank lines may be used to separate statements. Statements normally end with a newline; however, this is not the case for lines ending in a `,', `{', `?', `:', `&&', or `||'. Lines ending in `do` or `else` also have their statements automatically continued on the following line. In other cases, a line can be continued by ending it with a `\', in which case the newline is ignored.

Multiple statements may be put on one line by separating each one with a `;'. This applies to both the statements within the action part of a rule (the usual case), and to the rule statements.

See section Comments in `awk` Programs, for information on `awk`'s commenting convention; see section `awk` Statements Versus Lines, for a description of the line continuation mechanism in `awk`.

Pattern Summary

`awk` patterns may be one of the following:

```
/regular expression/
relational expression
pattern && pattern
pattern || pattern
pattern ? pattern : pattern
(pattern)
! pattern
```


```
pattern1, pattern2
BEGIN
END
```

`BEGIN` and `END` are two special kinds of patterns that are not tested against the input. The action parts of all `BEGIN` rules are concatenated as if all the statements had been written in a single `BEGIN` rule. They are executed before any of the input is read. Similarly, all the `END` rules are concatenated, and executed when all the input is exhausted (or when an `exit` statement is executed). `BEGIN` and `END` patterns cannot be combined with other patterns in pattern expressions. `BEGIN` and `END` rules cannot have missing action parts.

For `/regular-expression/` patterns, the associated statement is executed for each input record that matches the regular expression. Regular expressions are summarized below.

A relational expression may use any of the operators defined below in the section on actions. These generally test whether certain fields match certain regular expressions.

The `&&', `||', and `!' operators are logical "and," logical "or," and logical "not," respectively, as in C. They do short-circuit evaluation, also as in C, and are used for combining more primitive pattern expressions. As in most languages, parentheses may be used to change the order of evaluation.

The `?:' operator is like the same operator in C. If the first pattern matches, then the second pattern is matched against the input record; otherwise, the third is matched. Only one of the second and third patterns is matched.

The `pattern1, pattern2' form of a pattern is called a range pattern. It matches all input lines starting with a line that matches pattern1, and continuing until a line that matches pattern2, inclusive. A range pattern cannot be used as an operand of any of the pattern operators.

See section Pattern Elements.

Regular Expressions

Regular expressions are based on POSIX EREs (extended regular expressions). The escape sequences allowed in string constants are also valid in regular expressions (see section Escape Sequences). Regexps are composed of characters as follows:

`c`

matches the character c (assuming c is none of the characters listed below).

`\c`

matches the literal character c.

`.`

matches any character, *including* newline. In strict POSIX mode, `.' does not match the NUL character, which is a character with all bits equal to zero.

`^`

matches the beginning of a string.

`$`

matches the end of a string.

`[abc...]`

matches any of the characters abc... (character list).

`[[:class:]]`

matches any character in the character class class. Allowable classes are `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, and `xdigit`.

`[[.symbol.]]`

matches the multi-character collating symbol symbol. `gawk` does not currently support collating symbols.

`[[=chars=]]`

matches any of the equivalent characters in chars. `gawk` does not currently support equivalence classes.

`[^abc...]`

matches any character except abc... and newline (negated character list).

`r1|r2`

matches either r1 or r2 (alternation).

`r1r2`

matches r1, and then r2 (concatenation).

`r+`

matches one or more r's.

`r*`

matches zero or more r's.

`r?`

matches zero or one r's.

`(r)`

matches r (grouping).

`r{n}`
`r{n,}`
`r{n,m}`

matches at least n, n to any number, or n to m occurrences of r (interval expressions).

`\y`

matches the empty string at either the beginning or the end of a word.

`\B`

matches the empty string within a word.

`\<`

matches the empty string at the beginning of a word.

`\>`
: matches the empty string at the end of a word.

`\w`
: matches any word-constituent character (alphanumeric characters and the underscore).

`\W`
: matches any character that is not word-constituent.

`\``
: matches the empty string at the beginning of a buffer (same as a string in `gawk`).

`\'`
: matches the empty string at the end of a buffer.

The various command line options control how `gawk` interprets characters in regexps.

No options
: In the default case, `gawk` provide all the facilities of POSIX regexps and the GNU regexp operators described above. However, interval expressions are not supported.

`--posix`
: Only POSIX regexps are supported, the GNU operators are not special (e.g., `\w' matches a literal `w'). Interval expressions are allowed.

`--traditional`
: Traditional Unix `awk` regexps are matched. The GNU operators are not special, interval expressions are not available, and neither are the POSIX character classes (`[[:alnum:]]` and so on). Characters described by octal and hexadecimal escape sequences are treated literally, even if they represent regexp metacharacters.

`--re-interval`
: Allow interval expressions in regexps, even if `--traditional' has been provided.

See section Regular Expressions.

Actions

Action statements are enclosed in braces, `{' and `}'. A missing action statement is equivalent to `{ print }'.

Action statements consist of the usual assignment, conditional, and looping statements found in most languages. The operators, control statements, and Input/Output statements available are similar to those in C.

Comments begin with the `#' character, and continue until the end of the line. Blank lines may be used to separate statements. Statements normally end with a newline; however, this is not the case for lines ending in a `,', `{', `?', `:', `&&', or `||'. Lines ending in `do` or `else` also have their statements automatically continued on the following line. In other cases, a line can be continued by ending it with a `\', in which case the newline is ignored.

Multiple statements may be put on one line by separating each one with a `;'. This applies to both the statements within the action part of a rule (the usual case), and to the rule statements.

See section Comments in awk Programs, for information on awk's commenting convention; see section awk Statements Versus Lines, for a description of the line continuation mechanism in awk.

Operators

The operators in awk, in order of decreasing precedence, are:

`(...)`
: Grouping.

`$`
: Field reference.

`++ --`
: Increment and decrement, both prefix and postfix.

`^`
: Exponentiation (`**' may also be used, and `**=' for the assignment operator, but they are not specified in the POSIX standard).

`+ - !`
: Unary plus, unary minus, and logical negation.

`* / %`
: Multiplication, division, and modulus.

`+ -`
: Addition and subtraction.

`space`
: String concatenation.

`< <= > >= != ==`
: The usual relational operators.

`~ !~`
: Regular expression match, negated match.

`in`
: Array membership.

`&&`
: Logical "and".

`||`
: Logical "or".

`?:`
: A conditional expression. This has the form `expr1 ? expr2 : expr3'. If expr1 is true, the value of

the expression is expr2; otherwise it is expr3. Only one of expr2 and expr3 is evaluated.

`= += -= *= /= %= ^=`

Assignment. Both absolute assignment (`var=value`) and operator assignment (the other forms) are supported.

See section Expressions.

Control Statements

The control statements are as follows:

```
if (condition) statement [ else statement ]
while (condition) statement
do statement while (condition)
for (expr1; expr2; expr3) statement
for (var in array) statement
break
continue
delete array[index]
delete array
exit [ expression ]
{ statements }
```

See section Control Statements in Actions.

I/O Statements

The Input/Output statements are as follows:

`getline`

Set `$0` from next input record; set `NF`, `NR`, `FNR`. See section Explicit Input with `getline`.

`getline <file`

Set `$0` from next record of file; set `NF`.

`getline var`

Set var from next input record; set `NF`, `FNR`.

`getline var <file`

Set var from next record of file.

`command | getline`

Run command, piping its output into `getline`; sets `$0`, `NF`, `NR`.

`command | getline var`

Run command, piping its output into `getline`; sets var.

`next`

Stop processing the current input record. The next input record is read and processing starts over with the first pattern in the `awk` program. If the end of the input data is reached, the `END` rule(s), if any, are executed. See section The `next` Statement.

`nextfile`

Stop processing the current input file. The next input record read comes from the next input file. `FILENAME` is updated, `FNR` is set to one, `ARGIND` is incremented, and processing starts over with the first pattern in the `awk` program. If the end of the input data is reached, the `END` rule(s), if any, are executed. Earlier versions of `gawk` used `next file'; this usage is still supported, but is considered to be deprecated. See section The `nextfile` Statement.

`print`

Prints the current record. See section Printing Output.

`print expr-list`

Prints expressions.

`print expr-list > file`

Prints expressions to file. If file does not exist, it is created. If it does exist, its contents are deleted the first time the `print` is executed.

`print expr-list >> file`

Prints expressions to file. The previous contents of file are retained, and the output of `print` is appended to the file.

`print expr-list | command`

Prints expressions, sending the output down a pipe to command. The pipeline to the command stays open until the `close` function is called.

`printf fmt, expr-list`

Format and print.

`printf fmt, expr-list > file`

Format and print to file. If file does not exist, it is created. If it does exist, its contents are deleted the first time the `printf` is executed.

`printf fmt, expr-list >> file`

Format and print to file. The previous contents of file are retained, and the output of `printf` is appended to the file.

`printf fmt, expr-list | command`

Format and print, sending the output down a pipe to command. The pipeline to the command stays open until the `close` function is called.

`getline` returns zero on end of file, and -1 on an error. In the event of an error, `getline` will set `ERRNO` to the value of a system-dependent string that describes the error.

`printf` Summary

Conversion specification have the form %[flag][width][.prec]format. Items in brackets are optional.

The `awk` `printf` statement and `sprintf` function accept the following conversion specification formats:

`%c`

An ASCII character. If the argument used for `%c' is numeric, it is treated as a character and printed. Otherwise, the argument is assumed to be a string, and the only first character of that string is printed.

`%d`

`%i`

A decimal number (the integer part).

`%e`

`%E`

A floating point number of the form `[-]d.dddddde[+-]dd'. The `%E' format uses `E' instead of `e'.

`%f`

A floating point number of the form `[-]ddd.dddddd`.

`%g`

`%G`

Use either the `%e' or `%f' formats, whichever produces a shorter string, with non-significant zeros suppressed. `%G' will use `%E' instead of `%e'.

`%o`

An unsigned octal number (again, an integer).

`%s`

A character string.

`%x`

`%X`

An unsigned hexadecimal number (an integer). The `%X' format uses `A' through `F' instead of `a' through `f' for decimal 10 through 15.

`%%`

A single `%' character; no argument is converted.

There are optional, additional parameters that may lie between the `%' and the control letter:

`-`

The expression should be left-justified within its field.

`space`

For numeric conversions, prefix positive values with a space, and negative values with a minus sign.

`+`

The plus sign, used before the width modifier (see below), says to always supply a sign for numeric conversions, even if the data to be formatted is positive. The `+' overrides the space modifier.

`#`

Use an "alternate form" for certain control letters. For `o', supply a leading zero. For `x', and `X', supply a leading `0x' or `0X' for a non-zero result. For `e', `E', and `f', the result will always contain a decimal point. For `g', and `G', trailing zeros are not removed from the result.

`0`

A leading `0' (zero) acts as a flag, that indicates output should be padded with zeros instead of spaces. This applies even to non-numeric output formats. This flag only has an effect when the field width is wider than the value to be printed.

`width`

The field should be padded to this width. The field is normally padded with spaces. If the `0' flag has been used, it is padded with zeros.

`.prec`

A number that specifies the precision to use when printing. For the `e', `E', and `f' formats, this specifies the number of digits you want printed to the right of the decimal point. For the `g', and `G' formats, it specifies the maximum number of significant digits. For the `d', `o', `i', `u', `x', and `X' formats, it specifies the minimum number of digits to print. For the `s' format, it specifies the maximum number of characters from the string that should be printed.

Either or both of the width and prec values may be specified as `*'. In that case, the particular value is taken from the argument list.

See section Using `printf` Statements for Fancier Printing.

Special File Names

When doing I/O redirection from either `print` or `printf` into a file, or via `getline` from a file, `gawk` recognizes certain special file names internally. These file names allow access to open file descriptors inherited from `gawk`'s parent process (usually the shell). The file names are:

`` `/dev/stdin' ``

The standard input.

`` `/dev/stdout' ``

The standard output.

`` `/dev/stderr' ``

The standard error output.

`` `/dev/fd/n' ``

The file denoted by the open file descriptor n.

In addition, reading the following files provides process related information about the running `gawk`

program. All returned records are terminated with a newline.

`/dev/pid'
: Returns the process ID of the current process.

`/dev/ppid'
: Returns the parent process ID of the current process.

`/dev/pgrpid'
: Returns the process group ID of the current process.

`/dev/user'
: At least four space-separated fields, containing the return values of the `getuid`, `geteuid`, `getgid`, and `getegid` system calls. If there are any additional fields, they are the group IDs returned by `getgroups` system call. (Multiple groups may not be supported on all systems.)

These file names may also be used on the command line to name data files. These file names are only recognized internally if you do not actually have files with these names on your system.

See section Special File Names in `gawk`, for a longer description that provides the motivation for this feature.

Built-in Functions

`awk` provides a number of built-in functions for performing numeric operations, string related operations, and I/O related operations.

The built-in arithmetic functions are:

`atan2(y, x)`
: the arctangent of y/x in radians.

`cos(expr)`
: the cosine in radians.

`exp(expr)`
: the exponential function (`e ^ expr`).

`int(expr)`
: truncates to integer.

`log(expr)`
: the natural logarithm of `expr`.

`rand()`
: a random number between zero and one.

`sin(expr)`
: the sine in radians.

`sqrt(expr)`
: the square root function.

`srand([expr])`

use expr as a new seed for the random number generator. If no expr is provided, the time of day is used. The return value is the previous seed for the random number generator.

`awk` has the following built-in string functions:

`gensub(regex, subst, how [, target])`

If how is a string beginning with `g' or `G', then replace each match of regex in target with subst. Otherwise, replace the how'th occurrence. If target is not supplied, use `$0`. The return value is the changed string; the original target is not modified. Within subst, `\n', where n is a digit from one to nine, can be used to indicate the text that matched the n'th parenthesized subexpression.

`gsub(regex, subst [, target])`

for each substring matching the regular expression regex in the string target, substitute the string subst, and return the number of substitutions. If target is not supplied, use `$0`.

`index(str, search)`

returns the index of the string search in the string str, or zero if search is not present.

`length([str])`

returns the length of the string str. The length of `$0` is returned if no argument is supplied.

`match(str, regex)`

returns the position in str where the regular expression regex occurs, or zero if regex is not present, and sets the values of `RSTART` and `RLENGTH`.

`split(str, arr [, regex])`

splits the string str into the array arr on the regular expression regex, and returns the number of elements. If regex is omitted, `FS` is used instead. regex can be the null string, causing each character to be placed into its own array element. The array arr is cleared first.

`sprintf(fmt, expr-list)`

prints expr-list according to fmt, and returns the resulting string.

`sub(regex, subst [, target])`

just like `gsub`, but only the first matching substring is replaced.

`substr(str, index [, len])`

returns the len-character substring of str starting at index. If len is omitted, the rest of str is used.

`tolower(str)`

returns a copy of the string str, with all the upper-case characters in str translated to their corresponding lower-case counterparts. Non-alphabetic characters are left unchanged.

`toupper(str)`

returns a copy of the string str, with all the lower-case characters in str translated to their corresponding upper-case counterparts. Non-alphabetic characters are left unchanged.

The I/O related functions are:

`close(expr)`

Close the open file or pipe denoted by expr.

`fflush([expr])`

Flush any buffered output for the output file or pipe denoted by expr. If expr is omitted, standard output is flushed. If expr is the null string (""), all output buffers are flushed.

`system(cmd-line)`

Execute the command cmd-line, and return the exit status. If your operating system does not support `system`, calling it will generate a fatal error.

`system("")' can be used to force `awk` to flush any pending output. This is more portable, but less obvious, than calling `fflush`.

Time Functions

The following two functions are available for getting the current time of day, and for formatting time stamps.

`systime()`

returns the current time of day as the number of seconds since a particular epoch (Midnight, January 1, 1970 UTC, on POSIX systems).

`strftime([format[, timestamp]])`

formats timestamp according to the specification in format. The current time of day is used if no timestamp is supplied. A default format equivalent to the output of the `date` utility is used if no format is supplied. See section Functions for Dealing with Time Stamps, for the details on the conversion specifiers that `strftime` accepts.

See section Built-in Functions, for a description of all of `awk`'s built-in functions.

String Constants

String constants in `awk` are sequences of characters enclosed in double quotes ("). Within strings, certain escape sequences are recognized, as in C. These are:

`\\`

A literal backslash.

`\a`

The "alert" character; usually the ASCII BEL character.

`\b`

Backspace.

`\f`

Formfeed.

`\n`

Newline.

`\r`

Carriage return.

`\t`

Horizontal tab.

`\v`

Vertical tab.

`\xhex digits`

The character represented by the string of hexadecimal digits following the `\x'. As in ANSI C, all following hexadecimal digits are considered part of the escape sequence. E.g., `"\x1B"` is a string containing the ASCII ESC (escape) character. (The `\x' escape sequence is not in POSIX `awk`.)

`\ddd`

The character represented by the one, two, or three digit sequence of octal digits. Thus, `"\033"` is also a string containing the ASCII ESC (escape) character.

`\c`

The literal character c, if c is not one of the above.

The escape sequences may also be used inside constant regular expressions (e.g., the regexp `/[ \t\f\n\r\v]/` matches whitespace characters).

See section Escape Sequences.

User-defined Functions

Functions in `awk` are defined as follows:

```
function name(parameter list) { statements }
```

Actual parameters supplied in the function call are used to instantiate the formal parameters declared in the function. Arrays are passed by reference, other variables are passed by value.

If there are fewer arguments passed than there are names in parameter-list, the extra names are given the null string as their value. Extra names have the effect of local variables.

The open-parenthesis in a function call of a user-defined function must immediately follow the function name, without any intervening white space. This is to avoid a syntactic ambiguity with the concatenation operator.

The word `func` may be used in place of `function` (but not in POSIX `awk`).

Use the `return` statement to return a value from a function.

See section User-defined Functions.

Historical Features

There are two features of historical `awk` implementations that `gawk` supports.

First, it is possible to call the `length` built-in function not only with no arguments, but even without parentheses!

```
a = length
```

is the same as either of

```
a = length()
a = length($0)
```

For example:

```
$ echo abcdef | awk '{ print length }'
-| 6
```

This feature is marked as "deprecated" in the POSIX standard, and `gawk` will issue a warning about its use if `--lint' is specified on the command line. (The ability to use `length` this way was actually an accident of the original Unix `awk` implementation. If any built-in function used `$0` as its default argument, it was possible to call that function without the parentheses. In particular, it was common practice to use the `length` function in this fashion, and this usage was documented in the `awk` manual page.)

The other historical feature is the use of either the `break` statement, or the `continue` statement outside the body of a `while`, `for`, or `do` loop. Traditional `awk` implementations have treated such usage as equivalent to the `next` statement. More recent versions of Unix `awk` do not allow it. `gawk` supports this usage if `--traditional' has been specified.

See section Command Line Options, for more information about the `--posix' and `--lint' options.

Go to the previous, next section.

Go to the previous, next section.

Installing gawk

This appendix provides instructions for installing `gawk` on the various platforms that are supported by the developers. The primary developers support Unix (and one day, GNU), while the other ports were contributed. The file `` `ACKNOWLEDGMENT' `` in the `gawk` distribution lists the electronic mail addresses of the people who did the respective ports, and they are also provided in section Reporting Problems and Bugs.

The gawk Distribution

This section first describes how to get the `gawk` distribution, how to extract it, and then what is in the various files and subdirectories.

Getting the gawk Distribution

There are three ways you can get GNU software.

1. You can copy it from someone else who already has it.
2. You can order `gawk` directly from the Free Software Foundation. Software distributions are available for Unix, MS-DOS, and VMS, on tape, CD-ROM, or floppies (MS-DOS only). The address is:

   Free Software Foundation
   59 Temple Place--Suite 330
   Boston, MA 02111-1307 USA
   Phone: +1-617-542-5942
   Fax (including Japan): +1-617-542-2652
   E-mail: `gnu@prep.ai.mit.edu`

   Ordering from the FSF directly contributes to the support of the foundation and to the production of more free software.
3. You can get `gawk` by using anonymous `ftp` to the Internet host `ftp.gnu.ai.mit.edu`, in the directory `` `/pub/gnu' ``.

   Here is a list of alternate `ftp` sites from which you can obtain GNU software. When a site is listed as "site`:`directory" the directory indicates the directory where GNU software is kept. You should use a site that is geographically close to you.

   Asia:

   ```
   cair-archive.kaist.ac.kr:/pub/gnu
   ftp.cs.titech.ac.jp
   ftp.nectec.or.th:/pub/mirrors/gnu
   ```

`utsun.s.u-tokyo.ac.jp:/ftpsync/prep`

- Australia:

`archie.au:/gnu`

(`archie.oz` or `archie.oz.au` for ACSnet)

- Africa:

`ftp.sun.ac.za:/pub/gnu`

- Middle East:

`ftp.technion.ac.il:/pub/unsupported/gnu`

- Europe:

`archive.eu.net`
`ftp.denet.dk`
`ftp.eunet.ch`
`ftp.funet.fi:/pub/gnu`
`ftp.ieunet.ie:pub/gnu`
`ftp.informatik.rwth-aachen.de:/pub/gnu`
`ftp.informatik.tu-muenchen.de`
`ftp.luth.se:/pub/unix/gnu`
`ftp.mcc.ac.uk`
`ftp.stacken.kth.se`
`ftp.sunet.se:/pub/gnu`
`ftp.univ-lyon1.fr:pub/gnu`
`ftp.win.tue.nl:/pub/gnu`
`irisa.irisa.fr:/pub/gnu`
`isy.liu.se`
`nic.switch.ch:/mirror/gnu`
`src.doc.ic.ac.uk:/gnu`
`unix.hensa.ac.uk:/pub/uunet/systems/gnu`

- South America:

`ftp.inf.utfsm.cl:/pub/gnu`
`ftp.unicamp.br:/pub/gnu`

- Western Canada:

`ftp.cs.ubc.ca:/mirror2/gnu`

- USA:

`col.hp.com:/mirrors/gnu`
`f.ms.uky.edu:/pub3/gnu`
`ftp.cc.gatech.edu:/pub/gnu`

`ftp.cs.columbia.edu:/archives/gnu/prep`
`ftp.digex.net:/pub/gnu`
`ftp.hawaii.edu:/mirrors/gnu`
`ftp.kpc.com:/pub/mirror/gnu`

- USA (continued):

`ftp.uu.net:/systems/gnu`
`gatekeeper.dec.com:/pub/GNU`
`jaguar.utah.edu:/gnustuff`
`labrea.stanford.edu`
`mrcnext.cso.uiuc.edu:/pub/gnu`
`vixen.cso.uiuc.edu:/gnu`
`wuarchive.wustl.edu:/systems/gnu`

Extracting the Distribution

`gawk` is distributed as a `tar` file compressed with the GNU Zip program, `gzip`.

Once you have the distribution (for example, `` `gawk-3.0.0.tar.gz' ``), first use `gzip` to expand the file, and then use `tar` to extract it. You can use the following pipeline to produce the `gawk` distribution:

```
# Under System V, add 'o' to the tar flags
gzip -d -c gawk-3.0.0.tar.gz | tar -xvpf -
```

This will create a directory named `` `gawk-3.0.0' `` in the current directory.

The distribution file name is of the form `` `gawk-V.R.n.tar.gz' ``. The V represents the major version of `gawk`, the R represents the current release of version V, and the n represents a patch level, meaning that minor bugs have been fixed in the release. The current patch level is 0, but when retrieving distributions, you should get the version with the highest version, release, and patch level. (Note that release levels greater than or equal to 90 denote "beta," or non-production software; you may not wish to retrieve such a version unless you don't mind experimenting.)

If you are not on a Unix system, you will need to make other arrangements for getting and extracting the `gawk` distribution. You should consult a local expert.

Contents of the gawk Distribution

The `gawk` distribution has a number of C source files, documentation files, subdirectories and files related to the configuration process (see section Compiling and Installing `gawk` on Unix), and several subdirectories related to different, non-Unix, operating systems.

various `.c', `.y', and `.h' files

These files are the actual `gawk` source code.

`` `README' ``

`README_d/README.*'

Descriptive files: `README' for gawk under Unix, and the rest for the various hardware and software combinations.

`INSTALL'

A file providing an overview of the configuration and installation process.

`PORTS'

A list of systems to which gawk has been ported, and which have successfully run the test suite.

`ACKNOWLEDGMENT'

A list of the people who contributed major parts of the code or documentation.

`ChangeLog'

A detailed list of source code changes as bugs are fixed or improvements made.

`NEWS'

A list of changes to gawk since the last release or patch.

`COPYING'

The GNU General Public License.

`FUTURES'

A brief list of features and/or changes being contemplated for future releases, with some indication of the time frame for the feature, based on its difficulty.

`LIMITATIONS'

A list of those factors that limit gawk's performance. Most of these depend on the hardware or operating system software, and are not limits in gawk itself.

`POSIX.STD'

A description of one area where the POSIX standard for awk is incorrect, and how gawk handles the problem.

`PROBLEMS'

A file describing known problems with the current release.

`doc/gawk.1'

The troff source for a manual page describing gawk. This is distributed for the convenience of Unix users.

`doc/gawk.texi'

The Texinfo source file for this book. It should be processed with TeX to produce a printed document, and with makeinfo to produce an Info file.

`doc/gawk.info'

The generated Info file for this book.

`doc/igawk.1'

The troff source for a manual page describing the igawk program presented in section An Easy Way to Use Library Functions.

`doc/Makefile.in'

The input file used during the configuration process to generate the actual `Makefile' for creating the documentation.

`Makefile.in'

`acconfig.h'

`aclocal.m4'

`configh.in'

`configure.in'

`configure'

`custom.h'

`missing/*'

These files and subdirectory are used when configuring gawk for various Unix systems. They are explained in detail in section Compiling and Installing gawk on Unix.

`awklib/extract.awk'

`awklib/Makefile.in'

The `awklib' directory contains a copy of `extract.awk' (see section Extracting Programs from Texinfo Source Files), which can be used to extract the sample programs from the Texinfo source file for this book, and a `Makefile.in' file, which configure uses to generate a `Makefile'. As part of the process of building gawk, the library functions from section A Library of awk Functions, and the igawk program from section An Easy Way to Use Library Functions, are extracted into ready to use files. They are installed as part of the installation process.

`amiga/*'

Files needed for building gawk on an Amiga. See section Installing gawk on an Amiga, for details.

`atari/*'

Files needed for building gawk on an Atari ST. See section Installing gawk on the Atari ST, for details.

`pc/*'

Files needed for building gawk under MS-DOS and OS/2. See section MS-DOS and OS/2 Installation and Compilation, for details.

`vms/*'

Files needed for building gawk under VMS. See section How to Compile and Install gawk on VMS, for details.

`test/*'

A test suite for gawk. You can use `make check' from the top level gawk directory to run your version of gawk against the test suite. If gawk successfully passes `make check' then you can be confident of a successful port.

Compiling and Installing gawk on Unix

Usually, you can compile and install `gawk` by typing only two commands. However, if you do use an unusual system, you may need to configure `gawk` for your system yourself.

Compiling gawk for Unix

After you have extracted the `gawk` distribution, `cd` to `` `gawk-3.0.0' ``. Like most GNU software, `gawk` is configured automatically for your Unix system by running the `configure` program. This program is a Bourne shell script that was generated automatically using GNU `autoconf`. (The `autoconf` software is described fully in Autoconf--Generating Automatic Configuration Scripts, which is available from the Free Software Foundation.)

To configure `gawk`, simply run `configure`:

```
sh ./configure
```

This produces a `` `Makefile' `` and `` `config.h' `` tailored to your system. The `` `config.h' `` file describes various facts about your system. You may wish to edit the `` `Makefile' `` to change the `CFLAGS` variable, which controls the command line options that are passed to the C compiler (such as optimization levels, or compiling for debugging).

Alternatively, you can add your own values for most `make` variables, such as `CC` and `CFLAGS`, on the command line when running `configure`:

```
CC=cc CFLAGS=-g sh ./configure
```

See the file `` `INSTALL' `` in the `gawk` distribution for all the details.

After you have run `configure`, and possibly edited the `` `Makefile' ``, type:

```
make
```

and shortly thereafter, you should have an executable version of `gawk`. That's all there is to it! (If these steps do not work, please send in a bug report; see section Reporting Problems and Bugs.)

The Configuration Process

(This section is of interest only if you know something about using the C language and the Unix operating system.)

The source code for `gawk` generally attempts to adhere to formal standards wherever possible. This means that `gawk` uses library routines that are specified by the ANSI C standard and by the POSIX operating system interface standard. When using an ANSI C compiler, function prototypes are used to help improve the compile-time checking.

Many Unix systems do not support all of either the ANSI or the POSIX standards. The `` `missing' ``

subdirectory in the gawk distribution contains replacement versions of those subroutines that are most likely to be missing.

The `config.h' file that is created by the configure program contains definitions that describe features of the particular operating system where you are attempting to compile gawk. The three things described by this file are what header files are available, so that they can be correctly included, what (supposedly) standard functions are actually available in your C libraries, and other miscellaneous facts about your variant of Unix. For example, there may not be an st_blksize element in the stat structure. In this case `HAVE_ST_BLKSIZE' would be undefined.

It is possible for your C compiler to lie to configure. It may do so by not exiting with an error when a library function is not available. To get around this, you can edit the file `custom.h'. Use an `#ifdef' that is appropriate for your system, and either #define any constants that configure should have defined but didn't, or #undef any constants that configure defined and should not have. `custom.h' is automatically included by `config.h'.

It is also possible that the configure program generated by autoconf will not work on your system in some other fashion. If you do have a problem, the file `configure.in' is the input for autoconf. You may be able to change this file, and generate a new version of configure that will work on your system. See section Reporting Problems and Bugs, for information on how to report problems in configuring gawk. The same mechanism may be used to send in updates to `configure.in' and/or `custom.h'.

How to Compile and Install gawk on VMS

This section describes how to compile and install gawk under VMS.

Compiling gawk on VMS

To compile gawk under VMS, there is a DCL command procedure that will issue all the necessary CC and LINK commands, and there is also a `Makefile' for use with the MMS utility. From the source directory, use either

```
$ @[.VMS]VMSBUILD.COM
```

or

```
$ MMS/DESCRIPTION=[.VMS]DESCRIP.MMS GAWK
```

Depending upon which C compiler you are using, follow one of the sets of instructions in this table:

VAX C V3.x

> Use either `vmsbuild.com' or `descrip.mms' as is. These use CC/OPTIMIZE=NOLINE, which is essential for Version 3.0.

VAX C V2.x

> You must have Version 2.3 or 2.4; older ones won't work. Edit either `vmsbuild.com' or

`descrip.mms' according to the comments in them. For `vmsbuild.com', this just entails removing two `!' delimiters. Also edit `config.h' (which is a copy of file `[.config]vms-conf.h') and comment out or delete the two lines `#define __STDC__ 0' and `#define VAXC_BUILTINS' near the end.

GNU C

Edit `vmsbuild.com' or `descrip.mms'; the changes are different from those for VAX C V2.x, but equally straightforward. No changes to `config.h' should be needed.

DEC C

Edit `vmsbuild.com' or `descrip.mms' according to their comments. No changes to `config.h' should be needed.

gawk has been tested under VAX/VMS 5.5-1 using VAX C V3.2, GNU C 1.40 and 2.3. It should work without modifications for VMS V4.6 and up.

Installing gawk on VMS

To install gawk, all you need is a "foreign" command, which is a DCL symbol whose value begins with a dollar sign. For example:

```
$ GAWK :== $disk1:[gnubin]GAWK
```

(Substitute the actual location of gawk.exe for `$disk1:[gnubin]'.) The symbol should be placed in the `login.com' of any user who wishes to run gawk, so that it will be defined every time the user logs on. Alternatively, the symbol may be placed in the system-wide `sylogin.com' procedure, which will allow all users to run gawk.

Optionally, the help entry can be loaded into a VMS help library:

```
$ LIBRARY/HELP SYS$HELP:HELPLIB [.VMS]GAWK.HLP
```

(You may want to substitute a site-specific help library rather than the standard VMS library `HELPLIB'.) After loading the help text,

```
$ HELP GAWK
```

will provide information about both the gawk implementation and the awk programming language.

The logical name `AWK_LIBRARY' can designate a default location for awk program files. For the `-f' option, if the specified filename has no device or directory path information in it, gawk will look in the current directory first, then in the directory specified by the translation of `AWK_LIBRARY' if the file was not found. If after searching in both directories, the file still is not found, then gawk appends the suffix `.awk' to the filename and the file search will be re-tried. If `AWK_LIBRARY' is not defined, that portion of the file search will fail benignly.

Running gawk on VMS

Command line parsing and quoting conventions are significantly different on VMS, so examples in this book or from other sources often need minor changes. They *are* minor though, and all `awk` programs should run correctly.

Here are a couple of trivial tests:

```
$ gawk -- "BEGIN {print ""Hello, World!""}"
$ gawk -"W" version
! could also be -"W version" or "-W version"
```

Note that upper-case and mixed-case text must be quoted.

The VMS port of `gawk` includes a `DCL`-style interface in addition to the original shell-style interface (see the help entry for details). One side-effect of dual command line parsing is that if there is only a single parameter (as in the quoted string program above), the command becomes ambiguous. To work around this, the normally optional `--' flag is required to force Unix style rather than `DCL` parsing. If any other dash-type options (or multiple parameters such as data files to be processed) are present, there is no ambiguity and `--' can be omitted.

The default search path when looking for `awk` program files specified by the `-f' option is `"SYS$DISK:[],AWK_LIBRARY:"`. The logical name `AWKPATH' can be used to override this default. The format of `AWKPATH' is a comma-separated list of directory specifications. When defining it, the value should be quoted so that it retains a single translation, and not a multi-translation `RMS` searchlist.

Building and Using gawk on VMS POSIX

Ignore the instructions above, although `` `vms/gawk.hlp' `` should still be made available in a help library. Make sure that the `configure` script is executable; use `chmod +x' on it if necessary. Then execute the following commands:

```
$ POSIX
psx> CC=vms/posix-cc.sh configure
psx> CC=c89 make gawk
```

The first command will construct files `` `config.h' `` and `` `Makefile' `` out of templates. The second command will compile and link `gawk`. Ignore the warning `"Could not find lib m in lib list"`; it is harmless, caused by the explicit use of `-lm' as a linker option which is not needed under VMS POSIX. Under V1.1 (but not V1.0) a problem with the `yacc` skeleton `` `/etc/yyparse.c' `` will cause a compiler warning for `` `awktab.c' ``, followed by a linker warning about compilation warnings in the resulting object module. These warnings can be ignored.

Once built, `gawk` will work like any other shell utility. Unlike the normal VMS port of `gawk`, no special command line manipulation is needed in the VMS POSIX environment.

MS-DOS and OS/2 Installation and Compilation

If you have received a binary distribution prepared by the DOS maintainers, then `gawk` and the necessary support files will appear under the `` `gnu' `` directory, with executables in `` `gnu/bin' ``, libraries in `` `gnu/lib/awk' ``, and manual pages under `` `gnu/man' ``. This is designed for easy installation to a `` `/gnu' `` directory on your drive, but the files can be installed anywhere provided `AWKPATH` is set properly. Regardless of the installation directory, the first line of `` `igawk.cmd' `` and `` `igawk.bat' `` (in `` `gnu/bin' ``) may need to be edited.

The binary distribution will contain a separate file describing the contents. In particular, it may include more than one version of the `gawk` executable. OS/2 binary distributions may have a different arrangement, but installation is similar.

The OS/2 and MS-DOS versions of `gawk` search for program files as described in section The `AWKPATH` Environment Variable. However, semicolons (rather than colons) separate elements in the `AWKPATH` variable. If `AWKPATH` is not set or is empty, then the default search path is `".;c:/lib/awk;c:/gnu/lib/awk"`.

An `sh`-like shell (as opposed to `command.com` under MS-DOS or `cmd.exe` under OS/2) may be useful for `awk` programming. Ian Stewartson has written an excellent shell for MS-DOS and OS/2, and a `ksh` clone and GNU Bash are available for OS/2. The file `` `README_d/README.pc' `` in the `gawk` distribution contains information on these shells. Users of Stewartson's shell on DOS should examine its documentation on handling of command-lines. In particular, the setting for `gawk` in the shell configuration may need to be changed, and the `ignoretype` option may also be of interest.

`gawk` can be compiled for MS-DOS and OS/2 using the GNU development tools from DJ Delorie (DJGPP, MS-DOS-only) or Eberhard Mattes (EMX, MS-DOS and OS/2). Microsoft C can be used to build 16-bit versions for MS-DOS and OS/2. The file `` `README_d/README.pc' `` in the `gawk` distribution contains additional notes, and `` `pc/Makefile' `` contains important notes on compilation options.

To build `gawk`, copy the files in the `` `pc' `` directory to the directory with the rest of the `gawk` sources. The `` `Makefile' `` contains a configuration section with comments, and may need to be edited in order to work with your `make` utility.

The `` `Makefile' `` contains a number of targets for building various MS-DOS and OS/2 versions. A list of targets will be printed if the `make` command is given without a target. As an example, to build `gawk` using the DJGPP tools, enter `make djgpp'.

Using `make` to run the standard tests and to install `gawk` requires additional Unix-like tools, including `sh`, `sed`, and `cp`. In order to run the tests, the `` `test/*.ok' `` files may need to be converted so that they have the usual DOS-style end-of-line markers. Most of the tests will work properly with Stewartson's shell along with the companion utilities or appropriate GNU utilities. However, some editing of `` `test/Makefile' `` is required. It is recommended that the file `` `pc/Makefile.tst' `` be copied to `` `test/Makefile' `` as a replacement. Details can be found in `` `README_d/README.pc' ``.

TOC

Installing gawk on the Atari ST

There are no substantial differences when installing `gawk` on various Atari models. Compiled `gawk` executables do not require a large amount of memory with most `awk` programs and should run on all Motorola processor based models (called further ST, even if that is not exactly right).

In order to use `gawk`, you need to have a shell, either text or graphics, that does not map all the characters of a command line to upper-case. Maintaining case distinction in option flags is very important (see section Command Line Options). These days this is the default, and it may only be a problem for some very old machines. If your system does not preserve the case of option flags, you will need to upgrade your tools. Support for I/O redirection is necessary to make it easy to import `awk` programs from other environments. Pipes are nice to have, but not vital.

Compiling gawk on the Atari ST

A proper compilation of `gawk` sources when `sizeof(int)` differs from `sizeof(void *)` requires an ANSI C compiler. An initial port was done with `gcc`. You may actually prefer executables where `int`s are four bytes wide, but the other variant works as well.

You may need quite a bit of memory when trying to recompile the `gawk` sources, as some source files (`` `regex.c' `` in particular) are quite big. If you run out of memory compiling such a file, try reducing the optimization level for this particular file; this may help.

With a reasonable shell (Bash will do), and in particular if you run Linux, MiNT or a similar operating system, you have a pretty good chance that the `configure` utility will succeed. Otherwise sample versions of `` `config.h' `` and `` `Makefile.st' `` are given in the `` `atari' `` subdirectory and can be edited and copied to the corresponding files in the main source directory. Even if `configure` produced something, it might be advisable to compare its results with the sample versions and possibly make adjustments.

Some `gawk` source code fragments depend on a preprocessor define `atarist'. This basically assumes the TOS environment with `gcc`. Modify these sections as appropriate if they are not right for your environment. Also see the remarks about `AWKPATH` and `envsep` in section Running `gawk` on the Atari ST.

As shipped, the sample `` `config.h' `` claims that the `system` function is missing from the libraries, which is not true, and an alternative implementation of this function is provided in `` `atari/system.c' ``. Depending upon your particular combination of shell and operating system, you may wish to change the file to indicate that `system` is available.

Running gawk on the Atari ST

An executable version of `gawk` should be placed, as usual, anywhere in your `PATH` where your shell can find it.

While executing, `gawk` creates a number of temporary files. When using `gcc` libraries for TOS, `gawk`

looks for either of the environment variables `TEMP` or `TMPDIR`, in that order. If either one is found, its value is assumed to be a directory for temporary files. This directory must exist, and if you can spare the memory, it is a good idea to put it on a RAM drive. If neither `TEMP` nor `TMPDIR` are found, then `gawk` uses the current directory for its temporary files.

The ST version of `gawk` searches for its program files as described in section The `AWKPATH` Environment Variable. The default value for the `AWKPATH` variable is taken from `DEFPATH` defined in `` `Makefile' ``. The sample `gcc`/TOS `` `Makefile' `` for the ST in the distribution sets `DEFPATH` to `".,c:\lib\awk,c:\gnu\lib\awk"`. The search path can be modified by explicitly setting `AWKPATH` to whatever you wish. Note that colons cannot be used on the ST to separate elements in the `AWKPATH` variable, since they have another, reserved, meaning. Instead, you must use a comma to separate elements in the path. When recompiling, the separating character can be modified by initializing the `envsep` variable in `` `atari/gawkmisc.atr' `` to another value.

Although `awk` allows great flexibility in doing I/O redirections from within a program, this facility should be used with care on the ST running under TOS. In some circumstances the OS routines for file handle pool processing lose track of certain events, causing the computer to crash, and requiring a reboot. Often a warm reboot is sufficient. Fortunately, this happens infrequently, and in rather esoteric situations. In particular, avoid having one part of an `awk` program using `print` statements explicitly redirected to `"/dev/stdout"`, while other `print` statements use the default standard output, and a calling shell has redirected standard output to a file.

When `gawk` is compiled with the ST version of `gcc` and its usual libraries, it will accept both `/' and `\' as path separators. While this is convenient, it should be remembered that this removes one, technically valid, character (`/') from your file names, and that it may create problems for external programs, called via the `system` function, which may not support this convention. Whenever it is possible that a file created by `gawk` will be used by some other program, use only backslashes. Also remember that in `awk`, backslashes in strings have to be doubled in order to get literal backslashes (see section Escape Sequences).

Installing gawk on an Amiga

You can install `gawk` on an Amiga system using a Unix emulation environment available via anonymous `ftp` from `wuarchive.wustl.edu` in the directory `` `pub/aminet/dev/gcc' ``. This includes a shell based on `pdksh`. The primary component of this environment is a Unix emulation library, `` `ixemul.lib' ``.

A more complete distribution for the Amiga is available on the FreshFish CD-ROM from:

> Amiga Library Services
> 610 North Alma School Road, Suite 18
> Chandler, AZ 85224 USA
> Phone: +1-602-491-0048
> FAX: +1-602-491-0048
> E-mail: `orders@amigalib.com`

Once you have the distribution, you can configure `gawk` simply by running `configure`:


```
configure -v m68k-cbm-amigados
```

Then run `make`, and you should be all set! (If these steps do not work, please send in a bug report; see section Reporting Problems and Bugs.)

Reporting Problems and Bugs

If you have problems with `gawk` or think that you have found a bug, please report it to the developers; we cannot promise to do anything but we might well want to fix it.

Before reporting a bug, make sure you have actually found a real bug. Carefully reread the documentation and see if it really says you can do what you're trying to do. If it's not clear whether you should be able to do something or not, report that too; it's a bug in the documentation!

Before reporting a bug or trying to fix it yourself, try to isolate it to the smallest possible `awk` program and input data file that reproduces the problem. Then send us the program and data file, some idea of what kind of Unix system you're using, and the exact results `gawk` gave you. Also say what you expected to occur; this will help us decide whether the problem was really in the documentation.

Once you have a precise problem, there are two e-mail addresses you can send mail to.

Internet:

`bug-gnu-utils@prep.ai.mit.edu'

UUCP:

`uunet!prep.ai.mit.edu!bug-gnu-utils'

Please include the version number of `gawk` you are using. You can get this information with the command `gawk --version'. You should send a carbon copy of your mail to Arnold Robbins, who can be reached at `arnold@gnu.ai.mit.edu'.

Important! Do *not* try to report bugs in `gawk` by posting to the Usenet/Internet newsgroup `comp.lang.awk`. While the `gawk` developers do occasionally read this newsgroup, there is no guarantee that we will see your posting. The steps described above are the official, recognized ways for reporting bugs.

Non-bug suggestions are always welcome as well. If you have questions about things that are unclear in the documentation or are just obscure features, ask Arnold Robbins; he will try to help you out, although he may not have the time to fix the problem. You can send him electronic mail at the Internet address above.

If you find bugs in one of the non-Unix ports of `gawk`, please send an electronic mail message to the person who maintains that port. They are listed below, and also in the `README' file in the `gawk` distribution. Information in the `README` file should be considered authoritative if it conflicts with this book.

The people maintaining the non-Unix ports of `gawk` are:

MS-DOS

Scott Deifik, `scottd@amgen.com', and Darrel Hankerson, `hankedr@mail.auburn.edu'.

OS/2

Kai Uwe Rommel, `rommel@ars.de'.

VMS

Pat Rankin, `rankin@eql.caltech.edu'.

Atari ST

Michal Jaegermann, `michal@gortel.phys.ualberta.ca'.

Amiga

Fred Fish, `fnf@amigalib.com'.

If your bug is also reproducible under Unix, please send copies of your report to the general GNU bug list, as well as to Arnold Robbins, at the addresses listed above.

Other Freely Available `awk` Implementations

There are two other freely available `awk` implementations. This section briefly describes where to get them.

Unix `awk`

Brian Kernighan has been able to make his implementation of `awk` freely available. You can get it via anonymous `ftp` to the host `netlib.att.com`. Change directory to `` `/netlib/research' ``. Use "binary" or "image" mode, and retrieve `` `awk.bundle.Z' ``.

This is a shell archive that has been compressed with the `compress` utility. It can be uncompressed with either `uncompress` or the GNU `gunzip` utility.

This version requires an ANSI C compiler; GCC (the GNU C compiler) works quite nicely.

`mawk`

Michael Brennan has written an independent implementation of `awk`, called `mawk`. It is available under the GPL (see section GNU GENERAL PUBLIC LICENSE), just as `gawk` is.

You can get it via anonymous `ftp` to the host `oxy.edu`. Change directory to `` `/public' ``. Use "binary" or "image" mode, and retrieve `` `mawk1.2.1.tar.gz' `` (or the latest version that is there).

`gunzip` may be used to decompress this file. Installation is similar to `gawk`'s (see section Compiling and Installing `gawk` on Unix).

Go to the previous, next section.

Go to the previous, next section.

Implementation Notes

This appendix contains information mainly of interest to implementors and maintainers of gawk. Everything in it applies specifically to gawk, and not to other implementations.

Downward Compatibility and Debugging

See section Extensions in gawk Not in POSIX awk, for a summary of the GNU extensions to the awk language and program. All of these features can be turned off by invoking gawk with the `--traditional' option, or with the `--posix' option.

If gawk is compiled for debugging with `-DDEBUG', then there is one more option available on the command line:

`-W parsedebug`

`--parsedebug`

Print out the parse stack information as the program is being parsed.

This option is intended only for serious gawk developers, and not for the casual user. It probably has not even been compiled into your version of gawk, since it slows down execution.

Making Additions to gawk

If you should find that you wish to enhance gawk in a significant fashion, you are perfectly free to do so. That is the point of having free software; the source code is available, and you are free to change it as you wish (see section GNU GENERAL PUBLIC LICENSE).

This section discusses the ways you might wish to change gawk, and any considerations you should bear in mind.

Adding New Features

You are free to add any new features you like to gawk. However, if you want your changes to be incorporated into the gawk distribution, there are several steps that you need to take in order to make it possible for me to include to your changes.

1. Get the latest version. It is much easier for me to integrate changes if they are relative to the most recent distributed version of gawk. If your version of gawk is very old, I may not be able to integrate them at all. See section Getting the gawk Distribution, for information on getting the latest version of gawk.
2. Follow the GNU Coding Standards. This document describes how GNU software should be written. If you haven't read it, please do so, preferably *before* starting to modify gawk. (The GNU

Coding Standards are available as part of the Autoconf distribution, from the FSF.)

3. Use the `gawk` coding style. The C code for `gawk` follows the instructions in the GNU Coding Standards, with minor exceptions. The code is formatted using the traditional "K&R" style, particularly as regards the placement of braces and the use of tabs. In brief, the coding rules for `gawk` are:
   - Use old style (non-prototype) function headers when defining functions.
   - Put the name of the function at the beginning of its own line.
   - Put the return type of the function, even if it is `int`, on the line above the line with the name and arguments of the function.
   - The declarations for the function arguments should not be indented.
   - Put spaces around parentheses used in control structures (`if`, `while`, `for`, `do`, `switch` and `return`).
   - Do not put spaces in front of parentheses used in function calls.
   - Put spaces around all C operators, and after commas in function calls.
   - Do not use the comma operator to produce multiple side-effects, except in `for` loop initialization and increment parts, and in macro bodies.
   - Use real tabs for indenting, not spaces.
   - Use the "K&R" brace layout style.
   - Use comparisons against `NULL` and `'\0'` in the conditions of `if`, `while` and `for` statements, and in the `case`s of `switch` statements, instead of just the plain pointer or character value.
   - Use the `TRUE`, `FALSE`, and `NULL` symbolic constants, and the character constant `'\0'` where appropriate, instead of `1` and `0`.
   - Provide one-line descriptive comments for each function.
   - Do not use `#elif'. Many older Unix C compilers cannot handle it.

   If I have to reformat your code to follow the coding style used in `gawk`, I may not bother.
4. Be prepared to sign the appropriate paperwork. In order for the FSF to distribute your changes, you must either place those changes in the public domain, and submit a signed statement to that effect, or assign the copyright in your changes to the FSF. Both of these actions are easy to do, and *many* people have done so already. If you have questions, please contact me (see section Reporting Problems and Bugs), or `gnu@prep.ai.mit.edu`.
5. Update the documentation. Along with your new code, please supply new sections and or chapters for this book. If at all possible, please use real Texinfo, instead of just supplying unformatted ASCII text (although even that is better than no documentation at all). Conventions to be followed in AWK Language Programming are provided after the `@bye' at the end of the Texinfo source file. If possible, please update the man page as well.

   You will also have to sign paperwork for your documentation changes.
6. Submit changes as context diffs or unified diffs. Use `diff -c -r -N' or `diff -u -r -N' to compare the original `gawk` source tree with your version. (I find context diffs to be more readable, but unified

diffs are more compact.) I recommend using the GNU version of `diff`. Send the output produ by either run of `diff` to me when you submit your changes. See section Reporting Problems a Bugs, for the electronic mail information.

Using this format makes it easy for me to apply your changes to the master version of the `gawk` source code (using `patch`). If I have to apply the changes manually, using a text editor, I may not do so, particularly if there are lots of changes.

Although this sounds like a lot of work, please remember that while you may write the new code, I have to maintain it and support it, and if it isn't possible for me to do that with a minimum of extra work, then I probably will not.

Porting gawk to a New Operating System

If you wish to port `gawk` to a new operating system, there are several steps to follow.

1. Follow the guidelines in section Adding New Features, concerning coding style, submission of diffs, and so on.
2. When doing a port, bear in mind that your code must co-exist peacefully with the rest of `gawk`, and the other ports. Avoid gratuitous changes to the system-independent parts of the code. If at all possible, avoid sprinkling `#ifdef's just for your port throughout the code.

   If the changes needed for a particular system affect too much of the code, I probably will not accept them. In such a case, you will, of course, be able to distribute your changes on your own, as long as you comply with the GPL (see section GNU GENERAL PUBLIC LICENSE).
3. A number of the files that come with `gawk` are maintained by other people at the Free Software Foundation. Thus, you should not change them unless it is for a very good reason. I.e. changes are not out of the question, but changes to these files will be scrutinized extra carefully. The files are `` `alloca.c' ``, `` `getopt.h' ``, `` `getopt.c' ``, `` `getopt1.c' ``, `` `regex.h' ``, `` `regex.c' ``, `` `dfa.h' ``, `` `dfa.c' ``, `` `install-sh' ``, and `` `mkinstalldirs' ``.
4. Be willing to continue to maintain the port. Non-Unix operating systems are supported by volunteers who maintain the code needed to compile and run `gawk` on their systems. If no-one volunteers to maintain a port, that port becomes unsupported, and it may be necessary to remove it from the distribution.
5. Supply an appropriate `` `gawkmisc.???' `` file. Each port has its own `` `gawkmisc.???' `` that implements certain operating system specific functions. This is cleaner than a plethora of `#ifdef's scattered throughout the code. The `` `gawkmisc.c' `` in the main source directory includes the appropriate `` `gawkmisc.???' `` file from each subdirectory. Be sure to update it as well.

   Each port's `` `gawkmisc.???' `` file has a suffix reminiscent of the machine or operating system for the port. For example, `` `pc/gawkmisc.pc' `` and `` `vms/gawkmisc.vms' ``. The use of separate suffixes, instead of plain `` `gawkmisc.c' ``, makes it possible to move files from a port's subdirectory into the main subdirectory, without accidentally destroying the real `` `gawkmisc.c' `` file. (Currently, this is only an issue for the MS-DOS and OS/2 ports.)
6. Supply a `` `Makefile' `` and any other C source and header files that are necessary for your operating system. All your code should be in a separate subdirectory, with a name that is the same

as, or reminiscent of, either your operating system or the computer system. If possible, try to structure things so that it is not necessary to move files out of the subdirectory into the main source directory. If that is not possible, then be sure to avoid using names for your files that duplicate the names of files in the main source directory.

7. Update the documentation. Please write a section (or sections) for this book describing the installation and compilation steps needed to install and/or compile `gawk` for your system.
8. Be prepared to sign the appropriate paperwork. In order for the FSF to distribute your code, you must either place your code in the public domain, and submit a signed statement to that effect, or assign the copyright in your code to the FSF.

Following these steps will make it much easier to integrate your changes into `gawk`, and have them co-exist happily with the code for other operating systems that is already there.

In the code that you supply, and that you maintain, feel free to use a coding style and brace layout that suits your taste.

Probable Future Extensions

`AWK is a language similar to PERL, only considerably more elegant.`
`Arnold Robbins`

`Hey!`
`Larry Wall`

This section briefly lists extensions and possible improvements that indicate the directions we are currently considering for `gawk`. The file `` `FUTURES' `` in the `gawk` distributions lists these extensions as well.

This is a list of probable future changes that will be usable by the `awk` language programmer.

Localization

The GNU project is starting to support multiple languages. It will at least be possible to make `gawk` print its warnings and error messages in languages other than English. It may be possible for `awk` programs to also use the multiple language facilities, separate from `gawk` itself.

Databases

It may be possible to map a GDBM/NDBM/SDBM file into an `awk` array.

A `PROCINFO` Array

The special files that provide process-related information (see section Special File Names in gawk) may be superseded by a `PROCINFO` array that would provide the same information, in an easier to access fashion.

More `lint` warnings

There are more things that could be checked for portability.

Control of subprocess environment

Changes made in `gawk` to the array `ENVIRON` may be propagated to subprocesses run by `gaw`

This is a list of probable improvements that will make `gawk` perform better.

An Improved Version of `dfa`

The `dfa` pattern matcher from GNU `grep` has some problems. Either a new version or a fixed one will deal with some important regexp matching issues.

Use of `mmap`

On systems that support the `mmap` system call, its use would provide much faster file input, and considerably simplified input buffer management.

Use of GNU `malloc`

The GNU version of `malloc` could potentially speed up `gawk`, since it relies heavily on the use of dynamic memory allocation.

Use of the `rx` regexp library

The `rx` regular expression library could potentially speed up all regexp operations that require knowing the exact location of matches. This includes record termination, field and array splitting, and the `sub`, `gsub`, `gensub` and `match` functions.

Suggestions for Improvements

Here are some projects that would-be `gawk` hackers might like to take on. They vary in size from a few days to a few weeks of programming, depending on which one you choose and how fast a programmer you are. Please send any improvements you write to the maintainers at the GNU project. See section Adding New Features, for guidelines to follow when adding new features to `gawk`. See section Reporting Problems and Bugs, for information on contacting the maintainers.

1. Compilation of `awk` programs: `gawk` uses a Bison (YACC-like) parser to convert the script given it into a syntax tree; the syntax tree is then executed by a simple recursive evaluator. This method incurs a lot of overhead, since the recursive evaluator performs many procedure calls to do even the simplest things.

   It should be possible for `gawk` to convert the script's parse tree into a C program which the user would then compile, using the normal C compiler and a special `gawk` library to provide all the needed functions (regexps, fields, associative arrays, type coercion, and so on).

   An easier possibility might be for an intermediate phase of `awk` to convert the parse tree into a linear byte code form like the one used in GNU Emacs Lisp. The recursive evaluator would then be replaced by a straight line byte code interpreter that would be intermediate in speed between running a compiled program and doing what `gawk` does now.
2. The programs in the test suite could use documenting in this book.
3. See the `` `FUTURES' `` file for more ideas. Contact us if you would seriously like to tackle any of the items listed there.

Go to the previous, next section.

Go to the previous, next section.

Glossary

Action

A series of `awk` statements attached to a rule. If the rule's pattern matches an input record, `awk` executes the rule's action. Actions are always enclosed in curly braces. See section Overview of Actions.

Amazing `awk` Assembler

Henry Spencer at the University of Toronto wrote a retargetable assembler completely as `awk` scripts. It is thousands of lines long, including machine descriptions for several eight-bit microcomputers. It is a good example of a program that would have been better written in another language.

Amazingly Workable Formatter (`awf`)

Henry Spencer at the University of Toronto wrote a formatter that accepts a large subset of the `nroff -ms' and `nroff -man' formatting commands, using `awk` and `sh`.

ANSI

The American National Standards Institute. This organization produces many standards, among them the standards for the C and C++ programming languages.

Assignment

An `awk` expression that changes the value of some `awk` variable or data object. An object that you can assign to is called an lvalue. The assigned values are called rvalues. See section Assignment Expressions.

`awk` Language

The language in which `awk` programs are written.

`awk` Program

An `awk` program consists of a series of patterns and actions, collectively known as rules. For each input record given to the program, the program's rules are all processed in turn. `awk` programs may also contain function definitions.

`awk` Script

Another name for an `awk` program.

Bash

The GNU version of the standard shell (the Bourne-Again shell). See "Bourne Shell."

BBS

See "Bulletin Board System."

Boolean Expression

Named after the English mathematician Boole. See "Logical Expression."

Bourne Shell

The standard shell (`/bin/sh') on Unix and Unix-like systems, originally written by Steven Bourne. Many shells (Bash, ksh, pdksh, zsh) are generally upwardly compatible with the Bourne shell.

Built-in Function

The awk language provides built-in functions that perform various numerical, time stamp related, and string computations. Examples are sqrt (for the square root of a number) and substr (for a substring of a string). See section Built-in Functions.

Built-in Variable

ARGC, ARGIND, ARGV, CONVFMT, ENVIRON, ERRNO, FIELDWIDTHS, FILENAME, FNR, FS, IGNORECASE, NF, NR, OFMT, OFS, ORS, RLENGTH, RSTART, RS, RT, and SUBSEP, are the variables that have special meaning to awk. Changing some of them affects awk's running environment. Several of these variables are specific to gawk. See section Built-in Variables.

Braces

See "Curly Braces."

Bulletin Board System

A computer system allowing users to log in and read and/or leave messages for other users of the system, much like leaving paper notes on a bulletin board.

C

The system programming language that most GNU software is written in. The awk programming language has C-like syntax, and this book points out similarities between awk and C when appropriate.

Character Set

The set of numeric codes used by a computer system to represent the characters (letters, numbers, punctuation, etc.) of a particular country or place. The most common character set in use today is ASCII (American Standard Code for Information Interchange). Many European countries use an extension of ASCII known as ISO-8859-1 (ISO Latin-1).

CHEM

A preprocessor for pic that reads descriptions of molecules and produces pic input for drawing them. It was written in awk by Brian Kernighan and Jon Bentley, and is available from netlib@research.att.com.

Compound Statement

A series of awk statements, enclosed in curly braces. Compound statements may be nested. See section Control Statements in Actions.

Concatenation

Concatenating two strings means sticking them together, one after another, giving a new string. For example, the string `foo' concatenated with the string `bar' gives the string `foobar'. See section String Concatenation.

Conditional Expression

An expression using the `?:' ternary operator, such as `expr1 ? expr2 : expr3'. The expression expr1 is evaluated; if the result is true, the value of the whole expression is the value of expr2, otherwise

the value is expr3. In either case, only one of expr2 and expr3 is evaluated. See section Conditional Expressions.

Comparison Expression

A relation that is either true or false, such as `(a < b)'. Comparison expressions are used in `if`, `while`, `do`, and `for` statements, and in patterns to select which input records to process. See section Variable Typing and Comparison Expressions.

Curly Braces

The characters `{' and `}'. Curly braces are used in `awk` for delimiting actions, compound statements, and function bodies.

Dark Corner

An area in the language where specifications often were (or still are) not clear, leading to unexpected or undesirable behavior. Such areas are marked in this book with "(d.c.)" in the text, and are indexed under the heading "dark corner."

Data Objects

These are numbers and strings of characters. Numbers are converted into strings and vice versa, as needed. See section Conversion of Strings and Numbers.

Double Precision

An internal representation of numbers that can have fractional parts. Double precision numbers keep track of more digits than do single precision numbers, but operations on them are more expensive. This is the way `awk` stores numeric values. It is the C type `double`.

Dynamic Regular Expression

A dynamic regular expression is a regular expression written as an ordinary expression. It could be a string constant, such as `"foo"`, but it may also be an expression whose value can vary. See section Using Dynamic Regexps.

Environment

A collection of strings, of the form name=val, that each program has available to it. Users generally place values into the environment in order to provide information to various programs. Typical examples are the environment variables `HOME` and `PATH`.

Empty String

See "Null String."

Escape Sequences

A special sequence of characters used for describing non-printing characters, such as `\n' for newline, or `\033' for the ASCII ESC (escape) character. See section Escape Sequences.

Field

When `awk` reads an input record, it splits the record into pieces separated by whitespace (or by a separator regexp which you can change by setting the built-in variable `FS`). Such pieces are called fields. If the pieces are of fixed length, you can use the built-in variable `FIELDWIDTHS` to describe their lengths. See section Specifying How Fields are Separated, and also see See section Reading Fixed-width Data.

Floating Point Number

Often referred to in mathematical terms as a "rational" number, this is just a number that can have a fractional part. See "Double Precision" and "Single Precision."

Format

Format strings are used to control the appearance of output in the `printf` statement. Also, data conversions from numbers to strings are controlled by the format string contained in the built-in variable `CONVFMT`. See section Format-Control Letters.

Function

A specialized group of statements used to encapsulate general or program-specific tasks. `awk` has a number of built-in functions, and also allows you to define your own. See section Built-in Functions, and section User-defined Functions.

FSF

See "Free Software Foundation."

Free Software Foundation

A non-profit organization dedicated to the production and distribution of freely distributable software. It was founded by Richard M. Stallman, the author of the original Emacs editor. GNU Emacs is the most widely used version of Emacs today.

`gawk`

The GNU implementation of `awk`.

General Public License

This document describes the terms under which `gawk` and its source code may be distributed. (see section GNU GENERAL PUBLIC LICENSE)

GNU

"GNU's not Unix". An on-going project of the Free Software Foundation to create a complete, freely distributable, POSIX-compliant computing environment.

GPL

See "General Public License."

Hexadecimal

Base 16 notation, where the digits are `0-9` and `A-F`, with `A' representing 10, `B' representing 11, and so on up to `F' for 15. Hexadecimal numbers are written in C using a leading `0x', to indicate their base. Thus, `0x12` is 18 (one times 16 plus 2).

I/O

Abbreviation for "Input/Output," the act of moving data into and/or out of a running program.

Input Record

A single chunk of data read in by `awk`. Usually, an `awk` input record consists of one line of text. See section How Input is Split into Records.

Integer

A whole number, i.e. a number that does not have a fractional part.

Keyword

In the `awk` language, a keyword is a word that has special meaning. Keywords are reserved and may not be used as variable names.

`gawk`'s keywords are: `BEGIN`, `END`, `if`, `else`, `while`, `do...while`, `for`, `for...in`, `break`, `continue`, `delete`, `next`, `nextfile`, `function`, `func`, and `exit`.

Logical Expression

An expression using the operators for logic, AND, OR, and NOT, written `&&', `||', and `!' in `awk`. Often called Boolean expressions, after the mathematician who pioneered this kind of mathematical logic.

Lvalue

An expression that can appear on the left side of an assignment operator. In most languages, lvalues can be variables or array elements. In `awk`, a field designator can also be used as an lvalue.

Null String

A string with no characters in it. It is represented explicitly in `awk` programs by placing two double-quote characters next to each other (`""`). It can appear in input data by having two successive occurrences of the field separator appear next to each other.

Number

A numeric valued data object. The `gawk` implementation uses double precision floating point to represent numbers. Very old `awk` implementations use single precision floating point.

Octal

Base-eight notation, where the digits are `0-7`. Octal numbers are written in C using a leading `0', to indicate their base. Thus, `013` is 11 (one times 8 plus 3).

Pattern

Patterns tell `awk` which input records are interesting to which rules.

A pattern is an arbitrary conditional expression against which input is tested. If the condition is satisfied, the pattern is said to match the input record. A typical pattern might compare the input record against a regular expression. See section Pattern Elements.

POSIX

The name for a series of standards being developed by the IEEE that specify a Portable Operating System interface. The "IX" denotes the Unix heritage of these standards. The main standard of interest for `awk` users is IEEE Standard for Information Technology, Standard 1003.2-1992, Portable Operating System Interface (POSIX) Part 2: Shell and Utilities. Informally, this standard is often referred to as simply "P1003.2."

Private

Variables and/or functions that are meant for use exclusively by library functions, and not for the main `awk` program. Special care must be taken when naming such variables and functions. See section Naming Library Function Global Variables.

Range (of input lines)

A sequence of consecutive lines from the input file. A pattern can specify ranges of input lines for

awk to process, or it can specify single lines. See section Pattern Elements.

Recursion

When a function calls itself, either directly or indirectly. If this isn't clear, refer to the entry for "recursion."

Redirection

Redirection means performing input from other than the standard input stream, or output to other than the standard output stream.

You can redirect the output of the `print` and `printf` statements to a file or a system command, using the `>', `>>', and `|' operators. You can redirect input to the `getline` statement using the `<' and `|' operators. See section Redirecting Output of `print` and `printf`, and section Explicit Input with `getline`.

Regexp

Short for regular expression. A regexp is a pattern that denotes a set of strings, possibly an infinite set. For example, the regexp `R.*xp' matches any string starting with the letter `R' and ending with the letters `xp'. In awk, regexps are used in patterns and in conditional expressions. Regexps may contain escape sequences. See section Regular Expressions.

Regular Expression

See "regexp."

Regular Expression Constant

A regular expression constant is a regular expression written within slashes, such as `/foo/`. This regular expression is chosen when you write the awk program, and cannot be changed doing its execution. See section How to Use Regular Expressions.

Rule

A segment of an awk program that specifies how to process single input records. A rule consists of a pattern and an action. awk reads an input record; then, for each rule, if the input record satisfies the rule's pattern, awk executes the rule's action. Otherwise, the rule does nothing for that input record.

Rvalue

A value that can appear on the right side of an assignment operator. In awk, essentially every expression has a value. These values are rvalues.

`sed`

See "Stream Editor."

Short-Circuit

The nature of the awk logical operators `&&' and `||'. If the value of the entire expression can be deduced from evaluating just the left-hand side of these operators, the right-hand side will not be evaluated (see section Boolean Expressions).

Side Effect

A side effect occurs when an expression has an effect aside from merely producing a value. Assignment expressions, increment and decrement expressions and function calls have side

effects. See section Assignment Expressions.

Single Precision

An internal representation of numbers that can have fractional parts. Single precision numbers keep track of fewer digits than do double precision numbers, but operations on them are less expensive in terms of CPU time. This is the type used by some very old versions of `awk` to store numeric values. It is the C type `float`.

Space

The character generated by hitting the space bar on the keyboard.

Special File

A file name interpreted internally by `gawk`, instead of being handed directly to the underlying operating system. For example, `` `/dev/stderr' ``. See section Special File Names in `gawk`.

Stream Editor

A program that reads records from an input stream and processes them one or more at a time. This is in contrast with batch programs, which may expect to read their input files in entirety before starting to do anything, and with interactive programs, which require input from the user.

String

A datum consisting of a sequence of characters, such as `I am a string'. Constant strings are written with double-quotes in the `awk` language, and may contain escape sequences. See section Escape Sequences.

Tab

The character generated by hitting the TAB key on the keyboard. It usually expands to up to eight spaces upon output.

Unix

A computer operating system originally developed in the early 1970's at AT&T Bell Laboratories. It initially became popular in universities around the world, and later moved into commercial evnironments as a software development system and network server system. There are many commercial versions of Unix, as well as several work-alike systems whose source code is freely available (such as Linux, NetBSD, and FreeBSD).

Whitespace

A sequence of space or tab characters occurring inside an input record or a string.

Go to the previous, next section.

Go to the previous, next section.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works

based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
    1. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
    2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
    3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for non-commercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

    The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

    If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.
5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other

pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,

REPAIR OR CORRECTION.

13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place -- Suite 330, Boston, MA 02111-1307, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type `show c'
```


```
for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program `Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Go to the previous, next section.

Go to the previous section.

Index

/

<

=

>

\

_

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

S

t

u

V

W

|

~

Go to the previous section.

AWK Language Programming

(1)

These commands are available on POSIX compliant systems, as well as on traditional Unix based systems. If you are using some other operating system, you still need to be familiar with the ideas of I/O redirection and pipes

(2)

Often, these systems use `gawk` for their `awk` implementation!

(3)

The `#!' mechanism works on Linux systems, Unix systems derived from Berkeley Unix, System V Release 4, and some System V Release 3 systems.

(4)

The line beginning with `#!' lists the full file name of an interpreter to be run, and an optional initial command line argument to pass to that interpreter. The operating system then runs the interpreter with the given argument and the full argument list of the executed program. The first argument in the list is the full file name of the `awk` program. The rest of the argument list will either be options to `awk`, or data files, or both.

(5)

The `sed` utility is a "stream editor." Its behavior is also defined by the POSIX standard.

(6)

The internal representation uses double-precision floating point numbers. If you don't know what that means, then don't worry about it.

(7)

Some early implementations of Unix `awk` initialized `FILENAME` to `"-"`, even if there were data files to be processed. This behavior was incorrect, and should not be relied upon in your programs.

(8)

Computer generated random numbers really are not truly random. They are technically known as "pseudo-random." This means that while the numbers in a sequence appear to be random, you can in fact generate the same sequence of random numbers over and over again.

(9)

This consequence was certainly unintended.

(10)

As of December 1995, with final approval and publication hopefully sometime in 1996.

(11)

Occasionally there are minutes in a year with one or two leap seconds, which is why the seconds can go up to 61.

(12)

This is because ANSI C leaves the behavior of the C version of `strftime` undefined, and `gawk` will use the system's version of `strftime` if it's there. Typically, the conversion specifier will either not appear in the returned string, or it will appear literally.

(13)

If you don't understand any of this, don't worry about it; these facilities are meant to make it easier to "internationalize" programs.

(14)

Not recommended.

(15)

Your version of `gawk` may use a directory that is different than `` `/usr/local/share/awk' ``; it will depend upon how `gawk` was built and installed. The actual directory will be the value of `$(datadir)' generated when `gawk` was configured. You probably don't need to worry about this though.

(16)

Some implementations of `awk` do not allow you to execute `next` from within a function body. Some other work-around will be necessary if you use such a version.

(17)

ASCII has been extended in many countries to use the values from 128 to 255 for country-specific characters. If your system uses these extensions, you can simplify `_ord_init` to simply loop from zero to 255.

(18)

This is the Epoch on POSIX systems. It may be different on other systems.

(19)

Examine the code in section Noting Data File Boundaries. Why must `wc` use a separate `lines` variable, instead of using the value of `FNR` in `endfile`?

(20)

On older, non-POSIX systems, `tr` often does not require that the lists be enclosed in square brackets and quoted. This is a feature.

(21)

This program was written before `gawk` acquired the ability to split each character in a string into separate array elements. How might this ability simplify the program?

(22)

"Real world" is defined as "a program actually used to get something done."

(23)

The path may use a directory other than `` `/usr/local/share/awk' ``, depending upon how `gawk` was built and installed.

Using and Porting GNU CC

Go to the next section.

Using and Porting GNU CC

Richard M. Stallman

Last updated 14 June 1995

For GCC Version 2.7.

Published by the Free Software Foundation
675 Massachusetts Avenue
Cambridge, MA 02139 USA

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it.

(Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.
2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate

copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License a to the absence of any warranty; and give any other recipients of the Program a copy of this Licer along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
    1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
    2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
    3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

    These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

    Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

    In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
    1. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
    2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of

Sections 1 and 2 above on a medium customarily used for software interchange; or,

3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting th integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM

(INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
`Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Go to the next section.

Go to the previous, next section.

Contributors to GNU CC

In addition to Richard Stallman, several people have written parts of GNU CC.

- The idea of using RTL and some of the optimization ideas came from the program PO written at the University of Arizona by Jack Davidson and Christopher Fraser. See "Register Allocation and Exhaustive Peephole Optimization", Software Practice and Experience 14 (9), Sept. 1984, 857-866.
- Paul Rubin wrote most of the preprocessor.
- Leonard Tower wrote parts of the parser, RTL generator, and RTL definitions, and of the Vax machine description.
- Ted Lemon wrote parts of the RTL reader and printer.
- Jim Wilson implemented loop strength reduction and some other loop optimizations.
- Nobuyuki Hikichi of Software Research Associates, Tokyo, contributed the support for the Sony NEWS machine.
- Charles LaBrec contributed the support for the Integrated Solutions 68020 system.
- Michael Tiemann of Cygnus Support wrote the front end for C++, as well as the support for inline functions and instruction scheduling. Also the descriptions of the National Semiconductor 32000 series cpu, the SPARC cpu and part of the Motorola 88000 cpu.
- Gerald Baumgartner added the signature extension to the C++ front-end.
- Jan Stein of the Chalmers Computer Society provided support for Genix, as well as part of the 32000 machine description.
- Randy Smith finished the Sun FPA support.
- Robert Brown implemented the support for Encore 32000 systems.
- David Kashtan of SRI adapted GNU CC to VMS.
- Alex Crain provided changes for the 3b1.
- Greg Satz and Chris Hanson assisted in making GNU CC work on HP-UX for the 9000 series 300.
- William Schelter did most of the work on the Intel 80386 support.
- Christopher Smith did the port for Convex machines.
- Paul Petersen wrote the machine description for the Alliant FX/8.
- Dario Dariol contributed the four varieties of sample programs that print a copy of their source.
- Alain Lichnewsky ported GNU CC to the Mips cpu.
- Devon Bowen, Dale Wiles and Kevin Zachmann ported GNU CC to the Tahoe.
- Jonathan Stone wrote the machine description for the Pyramid computer.
- Gary Miller ported GNU CC to Charles River Data Systems machines.
- Richard Kenner of the New York University Ultracomputer Research Laboratory wrote the machine descriptions for the AMD 29000, the DEC Alpha, the IBM RT PC, and the IBM RS/6000

as well as the support for instruction attributes. He also made changes to better support RISC processors including changes to common subexpression elimination, strength reduction, functi calling sequence handling, and condition code support, in addition to generalizing the code for frame pointer elimination.

- Richard Kenner and Michael Tiemann jointly developed reorg.c, the delay slot scheduler.
- Mike Meissner and Tom Wood of Data General finished the port to the Motorola 88000.
- Masanobu Yuhara of Fujitsu Laboratories implemented the machine description for the Tron architecture (specifically, the Gmicro).
- NeXT, Inc. donated the front end that supports the Objective C language.
- James van Artsdalen wrote the code that makes efficient use of the Intel 80387 register stack.
- Mike Meissner at the Open Software Foundation finished the port to the MIPS cpu, including adding ECOFF debug support, and worked on the Intel port for the Intel 80386 cpu.
- Ron Guilmette implemented the `protoize` and `unprotoize` tools, the support for Dwarf symbolic debugging information, and much of the support for System V Release 4. He has also worked heavily on the Intel 386 and 860 support.
- Torbjorn Granlund of the Swedish Institute of Computer Science implemented multiply-by-constant optimization and better long long support, and improved leaf function register allocation.
- Mike Stump implemented the support for Elxsi 64 bit CPU.
- John Wehle added the machine description for the Western Electric 32000 processor used in several 3b series machines (no relation to the National Semiconductor 32000 processor).
- Holger Teutsch provided the support for the Clipper cpu.
- Kresten Krab Thorup wrote the run time support for the Objective C language.
- Stephen Moshier contributed the floating point emulator that assists in cross-compilation and permits support for floating point numbers wider than 64 bits.
- David Edelsohn contributed the changes to RS/6000 port to make it support the PowerPC and POWER2 architectures.
- Steve Chamberlain wrote the support for the Hitachi SH processor.
- Peter Schauer wrote the code to allow debugging to work on the Alpha.
- Oliver M. Kellogg of Deutsche Aerospace contributed the port to the MIL-STD-1750A.

Go to the previous, next section.

Go to the previous, next section.

Funding Free Software

If you want to have more free software a few years from now, it makes sense for you to help encourage people to contribute funds for its development. The most effective approach known is to encourage commercial redistributors to donate.

Users of free software systems can boost the pace of development by encouraging for-a-fee distributors to donate part of their selling price to free software developers--the Free Software Foundation, and others.

The way to convince distributors to do this is to demand it and expect it from them. So when you compare distributors, judge them partly by how much they give to free software development. Show distributors they must compete to be the one who gives the most.

To make this approach work, you must insist on numbers that you can compare, such as, "We will donate ten dollars to the Frobnitz project for each disk sold." Don't be satisfied with a vague promise, such as "A portion of the profits are donated," since it doesn't give a basis for comparison.

Even a precise fraction "of the profits from this disk" is not very meaningful, since creative accounting and unrelated business decisions can greatly alter what fraction of the sales price counts as profit. If the price you pay is $50, ten percent of the profit is probably less than a dollar; it might be a few cents, or nothing at all.

Some redistributors do development work themselves. This is useful too; but to keep everyone honest, you need to inquire how much they do, and what kind. Some kinds of development make much more long-term difference than others. For example, maintaining a separate version of a program contributes very little; maintaining the standard version of a program for the whole community contributes much. Easy new ports contribute little, since someone else would surely do them; difficult ports such as adding a new CPU to the GNU C compiler contribute more; major new features or packages contribute the most.

By establishing the idea that supporting further development is "the proper thing to do" when distributing free software for a fee, we can assure a steady flow of resources into making more free software.

Go to the previous, next section.

Go to the previous, next section.

Protect Your Freedom--Fight "Look And Feel"

This section is a political message from the League for Programming Freedom to the users of GNU CC. We have included it here because the issue of interface copyright is important to the GNU project.

Apple and Lotus have tried to create a new form of legal monopoly: a copyright on a user interface.

An interface is a kind of language--a set of conventions for communication between two entities, human or machine. Until a few years ago, the law seemed clear: interfaces were outside the domain of copyright, so programmers could program freely and implement whatever interface the users demanded. Imitating de-facto standard interfaces, sometimes with improvements, was standard practice in the computer field. These improvements, if accepted by the users, caught on and became the norm; in this way, much progress took place.

Computer users, and most software developers, were happy with this state of affairs. However, large companies such as Apple and Lotus would prefer a different system--one in which they can own interfaces and thereby rid themselves of all serious competitors. They hope that interface copyright will give them, in effect, monopolies on major classes of software.

Other large companies such as IBM and Digital also favor interface monopolies, for the same reason: if languages become property, they expect to own many de-facto standard languages. But Apple and Lotus are the ones who have actually sued. Lotus has won lawsuits against two small companies, which were thus put out of business. Then they sued Borland; this case is now before the court of appeals. Apple's lawsuit against HP and Microsoft is also being decided by an appeals court. Widespread rumors that Apple had lost the case are untrue; as of July 1994, the final outcome is unknown.

If the monopolists get their way, they will hobble the software field:

- Gratuitous incompatibilities will burden users. Imagine if each car manufacturer had to design a different way to start, stop, and steer a car.
- Users will be "locked in" to whichever interface they learn; then they will be prisoners of one supplier, who will charge a monopolistic price.
- Large companies have an unfair advantage wherever lawsuits become commonplace. Since they can afford to sue, they can intimidate smaller developers with threats even when they don't really have a case.
- Interface improvements will come slower, since incremental evolution through creative partial imitation will no longer occur.

If interface monopolies are accepted, other large companies are waiting to grab theirs:

- Adobe is expected to claim a monopoly on the interfaces of various popular application programs, if Borland's appeal against Lotus fails.

- Open Computing magazine reported a Microsoft vice president as threatening to sue people w copy the interface of Windows.

Users invest a great deal of time and money in learning to use computer interfaces. Far more, in fact, than software developers invest in developing *and even implementing* the interfaces. Whoever can own an interface, has made its users into captives, and misappropriated their investment.

To protect our freedom from monopolies like these, a group of programmers and users have formed a grass-roots political organization, the League for Programming Freedom.

The purpose of the League is to oppose monopolistic practices such as interface copyright and software patents. The League calls for a return to the legal policies of the recent past, in which programmers could program freely. The League is not concerned with free software as an issue, and is not affiliated with the Free Software Foundation.

The League's activities include publicizing the issue, as is being done here, and filing friend-of-the-court briefs on behalf of defendants sued by monopolists. Recently the League filed a friend-of-the-court brief for Borland in its appeal against Lotus.

The League's membership rolls include John McCarthy, inventor of Lisp, Marvin Minsky, founder of the MIT Artificial Intelligence lab, Guy L. Steele, Jr., author of well-known books on Lisp and C, as well as Richard Stallman, the developer of GNU CC. Please join and add your name to the list. Membership dues in the League are $42 per year for programmers, managers and professionals; $10.50 for students; $21 for others.

Activist members are especially important, but members who have no time to give are also important. Surveys at major ACM conferences have indicated a vast majority of attendees agree with the League. If just ten percent of the programmers who agree with the League join the League, we will probably triumph.

To join, or for more information, phone (617) 243-4091 or write to:

```
League for Programming Freedom
1 Kendall Square #143
P.O. Box 9171
Cambridge, MA 02139
```

You can also send electronic mail to `lpf@uunet.uu.net`.

In addition to joining the League, here are some suggestions from the League for other things you can do to protect your freedom to write programs:

- Tell your friends and colleagues about this issue and how it threatens to ruin the computer industry.
- Mention that you are a League member in your `` `.signature' ``, and mention the League's email address for inquiries.
- Ask the companies you consider working for or working with to make statements against software monopolies, and give preference to those that do.

- When employers ask you to sign contracts giving them copyright or patent rights, insist on cla saying they can use these rights only defensively. Don't rely on "company policy," since that c change at any time; don't rely on an individual executive's private word, since that person may be replaced. Get a commitment just as binding as the commitment they get from you.
- Write to Congress to explain the importance of this issue.

```
House Subcommittee on Intellectual Property
2137 Rayburn Bldg
Washington, DC 20515

Senate Subcommittee on Patents, Trademarks and Copyrights
United States Senate
Washington, DC 20510
```

(These committees have received lots of mail already; let's give them even more.)

Democracy means nothing if you don't use it. Stand up and be counted!

Go to the previous, next section.

Go to the previous, next section.

Compile C, C++, or Objective C

The C, C++, and Objective C versions of the compiler are integrated; the GNU C compiler can compile programs written in C, C++, or Objective C.

"GCC" is a common shorthand term for the GNU C compiler. This is both the most general name for the compiler, and the name used when the emphasis is on compiling C programs.

When referring to C++ compilation, it is usual to call the compiler "G++". Since there is only one compiler, it is also accurate to call it "GCC" no matter what the language context; however, the term "G++" is more useful when the emphasis is on compiling C++ programs.

We use the name "GNU CC" to refer to the compilation system as a whole, and more specifically to the language-independent part of the compiler. For example, we refer to the optimization options as affecting the behavior of "GNU CC" or sometimes just "the compiler".

Front ends for other languages, such as Ada 9X, Fortran, Modula-3, and Pascal, are under development. These front-ends, like that for C++, are built in subdirectories of GNU CC and link to it. The result is an integrated compiler that can compile programs written in C, C++, Objective C, or any of the languages for which you have installed front ends.

In this manual, we only discuss the options for the C, Objective-C, and C++ compilers and those of the GNU CC core. Consult the documentation of the other front ends for the options to use when compiling programs written in other languages.

G++ is a *compiler*, not merely a preprocessor. G++ builds object code directly from your C++ program source. There is no intermediate C version of the program. (By contrast, for example, some other implementations use a program that generates a C program from your C++ source.) Avoiding an intermediate C representation of the program means that you get better object code, and better debugging information. The GNU debugger, GDB, works with this information in the object code to give you comprehensive C++ source-level editing capabilities (see section 'C and C++' in Debugging with GDB).

Go to the previous, next section.

Go to the previous, next section.

GNU CC Command Options

When you invoke GNU CC, it normally does preprocessing, compilation, assembly and linking. The "overall options" allow you to stop this process at an intermediate stage. For example, the `-c' option says not to run the linker. Then the output consists of object files output by the assembler.

Other options are passed on to one stage of processing. Some options control the preprocessor and others the compiler itself. Yet other options control the assembler and linker; most of these are not documented here, since you rarely need to use any of them.

Most of the command line options that you can use with GNU CC are useful for C programs; when an option is only useful with another language (usually C++), the explanation says so explicitly. If the description for a particular option does not mention a source language, you can use that option with all supported languages.

See section Compiling C++ Programs, for a summary of special options for compiling C++ programs.

The `gcc` program accepts options and file names as operands. Many options have multiletter names; therefore multiple single-letter options may *not* be grouped: `-dr' is very different from `-d -r'.

You can mix options and other arguments. For the most part, the order you use doesn't matter. Order does matter when you use several options of the same kind; for example, if you specify `-L' more than once, the directories are searched in the order specified.

Many options have long names starting with `-f' or with `-W'---for example, `-fforce-mem', `-fstrength-reduce', `-Wformat' and so on. Most of these have both positive and negative forms; the negative form of `-ffoo' would be `-fno-foo'. This manual documents only one of these two forms, whichever one is not the default.

Option Summary

Here is a summary of all the options, grouped by type. Explanations are in the following sections.

Overall Options

See section Options Controlling the Kind of Output.

```
-c  -S  -E  -o file  -pipe  -v  -x language
```

C Language Options

See section Options Controlling C Dialect.

```
-ansi  -fallow-single-precision -fcond-mismatch  -fno-asm
-fno-builtin  -fsigned-bitfields  -fsigned-char
-funsigned-bitfields  -funsigned-char  -fwritable-strings
```


```
-traditional  -traditional-cpp  -trigraphs
```

C++ Language Options

See section Options Controlling C++ Dialect.

```
-fall-virtual  -fdollars-in-identifiers  -felide-constructors
-fenum-int-equiv -fexternal-templates  -fhandle-signatures
-fmemoize-lookups  -fno-default-inline -fno-gnu-keywords
-fnonnull-objects  -foperator-names  -fstrict-prototype
-fthis-is-variable -nostdinc++ -traditional  +en
```

Warning Options

See section Options to Request or Suppress Warnings.

```
-fsyntax-only  -pedantic  -pedantic-errors
-w  -W  -Wall  -Waggregate-return  -Wbad-function-cast
-Wcast-align -Wcast-qual  -Wchar-subscript  -Wcomment
-Wconversion -Wenum-clash  -Werror  -Wformat
-Wid-clash-len  -Wimplicit  -Wimport  -Winline
-Wlarger-than-len  -Wmissing-declarations
-Wmissing-prototypes  -Wnested-externs
-Wno-import  -Woverloaded-virtual -Wparentheses
-Wpointer-arith  -Wredundant-decls -Wreorder -Wreturn-type -Wshadow
-Wstrict-prototypes  -Wswitch  -Wsynth  -Wtemplate-debugging
-Wtraditional  -Wtrigraphs -Wuninitialized  -Wunused
-Wwrite-strings
```

Debugging Options

See section Options for Debugging Your Program or GNU CC.

```
-a  -dletters  -fpretend-float
-g  -glevel -gcoff  -gdwarf  -gdwarf+
-ggdb  -gstabs  -gstabs+  -gxcoff  -gxcoff+
-p  -pg  -print-file-name=library  -print-libgcc-file-name
-print-prog-name=program  -print-search-dirs  -save-temps
```

Optimization Options

See section Options That Control Optimization.

```
-fcaller-saves  -fcse-follow-jumps  -fcse-skip-blocks
-fdelayed-branch  -fexpensive-optimizations
-ffast-math  -ffloat-store  -fforce-addr  -fforce-mem
-finline-functions  -fkeep-inline-functions
-fno-default-inline  -fno-defer-pop  -fno-function-cse
-fno-inline  -fno-peephole  -fomit-frame-pointer
-frerun-cse-after-loop  -fschedule-insns
-fschedule-insns2  -fstrength-reduce  -fthread-jumps
```


```
-funroll-all-loops  -funroll-loops
-O  -O0  -O1  -O2  -O3
```

Preprocessor Options

See section Options Controlling the Preprocessor.

```
-Aquestion(answer)  -C  -dD  -dM  -dN
-Dmacro[=defn]  -E  -H
-idirafter dir
-include file  -imacros file
-iprefix file  -iwithprefix dir
-iwithprefixbefore dir  -isystem dir
-M  -MD  -MM  -MMD  -MG  -nostdinc  -P  -trigraphs
-undef  -Umacro  -Wp,option
```

Assembler Option

See section Passing Options to the Assembler.

```
-Wa,option
```

Linker Options

See section Options for Linking.

```
object-file-name  -llibrary
-nostartfiles  -nodefaultlibs  -nostdlib
-s  -static  -shared  -symbolic
-Wl,option  -Xlinker option
-u symbol
```

Directory Options

See section Options for Directory Search.

```
-Bprefix  -Idir  -I-  -Ldir
```

Target Options

See section Specifying Target Machine and Compiler Version.

```
-b machine  -V version
```

Machine Dependent Options

See section Hardware Models and Configurations.

```
M680x0 Options
-m68000  -m68020  -m68020-40  -m68030  -m68040  -m68881
-mbitfield  -mc68000  -mc68020  -mfpa  -mnobitfield
-mrtd  -mshort  -msoft-float
```

VAX Options

```
-mg  -mgnu  -munix
```

SPARC Options

```
-mapp-regs  -mcypress  -mepilogue  -mflat  -mfpu  -mhard-float
-mhard-quad-float  -mno-app-regs  -mno-flat  -mno-fpu
-mno-epilogue  -mno-unaligned-doubles
-msoft-float  -msoft-quad-float
-msparclite  -msupersparc  -munaligned-doubles  -mv8
```

SPARC V9 compilers support the following options in addition to the above:

```
-mmedlow  -mmedany
-mint32  -mint64  -mlong32  -mlong64
-mno-stack-bias  -mstack-bias
```

Convex Options

```
-mc1  -mc2  -mc32  -mc34  -mc38
-margcount  -mnoargcount
-mlong32  -mlong64
-mvolatile-cache  -mvolatile-nocache
```

AMD29K Options

```
-m29000  -m29050  -mbw  -mnbw  -mdw  -mndw
-mlarge  -mnormal  -msmall
-mkernel-registers  -mno-reuse-arg-regs
-mno-stack-check  -mno-storem-bug
-mreuse-arg-regs  -msoft-float  -mstack-check
-mstorem-bug  -muser-registers
```

ARM Options

```
-mapcs -m2 -m3 -m6 -mbsd -mxopen -mno-symrename
```

M88K Options

```
-m88000  -m88100  -m88110  -mbig-pic
-mcheck-zero-division  -mhandle-large-shift
-midentify-revision  -mno-check-zero-division
-mno-ocs-debug-info  -mno-ocs-frame-position
-mno-optimize-arg-area  -mno-serialize-volatile
-mno-underscores  -mocs-debug-info
-mocs-frame-position  -moptimize-arg-area
-mserialize-volatile  -mshort-data-num  -msvr3
-msvr4  -mtrap-large-shift  -muse-div-instruction
-mversion-03.00  -mwarn-passed-structs
```

RS/6000 and PowerPC Options

```
-mcpu=cpu  type
-mpower  -mno-power  -mpower2  -mno-power2
-mpowerpc  -mno-powerpc
-mpowerpc-gpopt  -mno-powerpc-gpopt
-mpowerpc-gfxopt  -mno-powerpc-gfxopt
-mnew-mnemonics  -mno-new-mnemonics
-mfull-toc   -mminimal-toc  -mno-fop-in-toc  -mno-sum-in-toc
-msoft-float  -mhard-float -mmultiple -mno-multiple
-mstring -mno-string -mbit-align -mno-bit-align
-mstrict-align -mno-strict-align -mrelocatable -mno-relocatable
-mtoc -mno-toc -mtraceback -mno-traceback
-mlittle -mlittle-endian -mbig -mbig-endian
```

RT Options

```
-mcall-lib-mul  -mfp-arg-in-fpregs  -mfp-arg-in-gregs
-mfull-fp-blocks  -mhc-struct-return  -min-line-mul
-mminimum-fp-blocks  -mnohc-struct-return
```

MIPS Options

```
-mabicalls  -mcpu=cpu  type  -membedded-data
-membedded-pic  -mfp32  -mfp64  -mgas  -mgp32  -mgp64
-mgpopt  -mhalf-pic  -mhard-float  -mint64 -mips1
-mips2 -mips3  -mlong64  -mlong-calls  -mmemcpy
-mmips-as  -mmips-tfile  -mno-abicalls
-mno-embedded-data  -mno-embedded-pic
-mno-gpopt  -mno-long-calls
-mno-memcpy  -mno-mips-tfile  -mno-rnames  -mno-stats
-mrnames -msoft-float
-m4650 -msingle-float -mmad
-mstats  -EL  -EB  -G num  -nocpp
```

i386 Options

```
-m486  -m386 -mieee-fp  -mno-fancy-math-387
-mno-fp-ret-in-387  -msoft-float  -msvr3-shlib
-mno-wide-multiply -mrtd -malign-double
-mreg-alloc=list -mregparm=num
-malign-jumps=num -malign-loops=num
-malign-functions=num
```

HPPA Options

```
-mdisable-fpregs  -mdisable-indexing  -mfast-indirect-calls
-mgas  -mjump-in-delay -mlong-millicode-calls -mno-disable-fpregs
-mno-disable-indexing -mno-fast-indirect-calls -mno-gas
-mno-jump-in-delay -mno-millicode-long-calls
-mno-portable-runtime -mno-soft-float -msoft-float
```


```
-mpa-risc-1-0  -mpa-risc-1-1  -mportable-runtime -mschedule=list
```

Intel 960 Options

```
-mcpu type  -masm-compat  -mclean-linkage
-mcode-align  -mcomplex-addr  -mleaf-procedures
-mic-compat  -mic2.0-compat  -mic3.0-compat
-mintel-asm  -mno-clean-linkage  -mno-code-align
-mno-complex-addr  -mno-leaf-procedures
-mno-old-align  -mno-strict-align  -mno-tail-call
-mnumerics  -mold-align  -msoft-float  -mstrict-align
-mtail-call
```

DEC Alpha Options

```
-mfp-regs  -mno-fp-regs  -mno-soft-float
-msoft-float
```

Clipper Options

```
-mc300 -mc400
```

H8/300 Options

```
-mrelax  -mh
```

System V Options

```
-Qy  -Qn  -YP,paths  -Ym,dir
```

Code Generation Options

See section Options for Code Generation Conventions.

```
-fcall-saved-reg  -fcall-used-reg
-ffixed-reg  -finhibit-size-directive
-fno-common  -fno-ident  -fno-gnu-linker
-fpcc-struct-return  -fpic  -fPIC
-freg-struct-return  -fshared-data  -fshort-enums
-fshort-double  -fvolatile  -fvolatile-global
-fverbose-asm -fpack-struct +e0  +e1
```

Options Controlling the Kind of Output

Compilation can involve up to four stages: preprocessing, compilation proper, assembly and linking, always in that order. The first three stages apply to an individual source file, and end by producing an object file; linking combines all the object files (those newly compiled, and those specified as input) into an executable file.

For any given input file, the file name suffix determines what kind of compilation is done:

`file.c`

C source code which must be preprocessed.

`file.i`

C source code which should not be preprocessed.

`file.ii`

C++ source code which should not be preprocessed.

`file.m`

Objective-C source code. Note that you must link with the library `` `libobjc.a' `` to make an Objective-C program work.

`file.h`

C header file (not to be compiled or linked).

`file.cc`

`file.cxx`

`file.cpp`

`file.C`

C++ source code which must be preprocessed. Note that in `.cxx', the last two letters must both be literally `x'. Likewise, `.C' refers to a literal capital C.

`file.s`

Assembler code.

`file.S`

Assembler code which must be preprocessed.

`other`

An object file to be fed straight into linking. Any file name with no recognized suffix is treated this way.

You can specify the input language explicitly with the `-x' option:

`-x language`

Specify explicitly the language for the following input files (rather than letting the compiler choose a default based on the file name suffix). This option applies to all following input files until the next `-x' option. Possible values for language are:

```
c  objective-c  c++
c-header  cpp-output  c++-cpp-output
assembler  assembler-with-cpp
```

`-x none`

Turn off any specification of a language, so that subsequent files are handled according to their file name suffixes (as they are if `-x' has not been used at all).

If you only want some of the stages of compilation, you can use `-x' (or filename suffixes) to tell `gcc` where to start, and one of the options `-c', `-S', or `-E' to say where `gcc` is to stop. Note that some combinations (for example, `-x cpp-output -E' instruct `gcc` to do nothing at all.

`-c`

Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

By default, the object file name for a source file is made by replacing the suffix `.c', `.i', `.s', etc., with `.o'.

Unrecognized input files, not requiring compilation or assembly, are ignored.

`-S`

Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified.

By default, the assembler file name for a source file is made by replacing the suffix `.c', `.i', etc., with `.s'.

Input files that don't require compilation are ignored.

`-E`

Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output.

Input files which don't require preprocessing are ignored.

`-o file`

Place output in file file. This applies regardless to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code.

Since only one output file can be specified, it does not make sense to use `-o' when compiling more than one input file, unless you are producing an executable file as output.

If `-o' is not specified, the default is to put an executable file in `a.out', the object file for `source.suffix' in `source.o', its assembler file in `source.s', and all preprocessed C source on standard output.

`-v`

Print (on standard error output) the commands executed to run the stages of compilation. Also print the version number of the compiler driver program and of the preprocessor and the compiler proper.

`-pipe`

Use pipes rather than temporary files for communication between the various stages of compilation. This fails to work on some systems where the assembler is unable to read from a pipe; but the GNU assembler has no trouble.

Compiling C++ Programs

C++ source files conventionally use one of the suffixes `.C', `.cc', `cpp', or `.cxx'; preprocessed C++ files use the suffix `.ii'. GNU CC recognizes files with these names and compiles them as C++ programs even if you call the compiler the same way as for compiling C programs (usually with the name `gcc').

However, C++ programs often require class libraries as well as a compiler that understands the C++

language--and under some circumstances, you might want to compile programs from standard input, c otherwise without a suffix that flags them as C++ programs. `g++` is a program that calls GNU CC wit default language set to C++, and automatically specifies linking against the GNU class library libg++. (1) On many systems, the script `g++` is also installed with the name `c++`.

When you compile C++ programs, you may specify many of the same command-line options that you use for compiling programs in any language; or command-line options meaningful for C and related languages; or options that are meaningful only for C++ programs. See section Options Controlling C Dialect, for explanations of options for languages related to C. See section Options Controlling C++ Dialect, for explanations of options that are meaningful only for C++ programs.

Options Controlling C Dialect

The following options control the dialect of C (or languages derived from C, such as C++ and Objective C) that the compiler accepts:

`-ansi`

Support all ANSI standard C programs.

This turns off certain features of GNU C that are incompatible with ANSI C, such as the `asm`, `inline` and `typeof` keywords, and predefined macros such as `unix` and `vax` that identify the type of system you are using. It also enables the undesirable and rarely used ANSI trigraph feature, and disallows `$' as part of identifiers.

The alternate keywords `__asm__`, `__extension__`, `__inline__` and `__typeof__` continue to work despite `-ansi'. You would not want to use them in an ANSI C program, of course, but it is useful to put them in header files that might be included in compilations done with `-ansi'. Alternate predefined macros such as `__unix__` and `__vax__` are also available, with or without `-ansi'.

The `-ansi' option does not cause non-ANSI programs to be rejected gratuitously. For that, `-pedantic' is required in addition to `-ansi'. See section Options to Request or Suppress Warnings.

The macro `__STRICT_ANSI__` is predefined when the `-ansi' option is used. Some header files may notice this macro and refrain from declaring certain functions or defining certain macros that the ANSI standard doesn't call for; this is to avoid interfering with any programs that might use these names for other things.

The functions `alloca`, `abort`, `exit`, and `_exit` are not builtin functions when `-ansi' is used.

`-fno-asm`

Do not recognize `asm`, `inline` or `typeof` as a keyword, so that code can use these words as identifiers. You can use the keywords `__asm__`, `__inline__` and `__typeof__` instead. `-ansi' implies `-fno-asm'.

In C++, this switch only affects the `typeof` keyword, since `asm` and `inline` are standard keywords. You may want to use the `-fno-gnu-keywords' flag instead, as it also disables the other, C++-specific, extension keywords such as `headof`.

`-fno-builtin`

Don't recognize builtin functions that do not begin with two leading underscores. Currently, the functions affected include `abort`, `abs`, `alloca`, `cos`, `exit`, `fabs`, `ffs`, `labs`, `memcmp`, `memcpy`, `sin`, `sqrt`, `strcmp`, `strcpy`, and `strlen`.

GCC normally generates special code to handle certain builtin functions more efficiently; for instance, calls to `alloca` may become single instructions that adjust the stack directly, and calls to `memcpy` may become inline copy loops. The resulting code is often both smaller and faster, but since the function calls no longer appear as such, you cannot set a breakpoint on those calls, nor can you change the behavior of the functions by linking with a different library.

The `-ansi' option prevents `alloca` and `ffs` from being builtin functions, since these functions do not have an ANSI standard meaning.

`-trigraphs`

Support ANSI C trigraphs. You don't want to know about this brain-damage. The `-ansi' option implies `-trigraphs'.

`-traditional`

Attempt to support some aspects of traditional C compilers. Specifically:

All `extern` declarations take effect globally even if they are written inside of a function definition. This includes implicit declarations of functions.

The newer keywords `typeof`, `inline`, `signed`, `const` and `volatile` are not recognized. (You can still use the alternative keywords such as `__typeof__`, `__inline__`, and so on.)

Comparisons between pointers and integers are always allowed.

Integer types `unsigned short` and `unsigned char` promote to `unsigned int`.

Out-of-range floating point literals are not an error.

Certain constructs which ANSI regards as a single invalid preprocessing number, such as `0xe-0xd', are treated as expressions instead.

String "constants" are not necessarily constant; they are stored in writable space, and identical looking constants are allocated separately. (This is the same as the effect of `-fwritable-strings'.)

All automatic variables not declared `register` are preserved by `longjmp`. Ordinarily, GNU C follows ANSI C: automatic variables not declared `volatile` may be clobbered.

The character escape sequences `\x' and `\a' evaluate as the literal characters `x' and `a' respectively. Without `-traditional', `\x' is a prefix for the hexadecimal representation of a character, and `\a' produces a bell.

In C++ programs, assignment to `this` is permitted with `-traditional'. (The option `-fthis-is-variable' also has this effect.)

You may wish to use `-fno-builtin' as well as `-traditional' if your program uses names that are normally GNU C builtin functions for other purposes of its own.

You cannot use `-traditional' if you include any header files that rely on ANSI C features. Some vendors are starting to ship systems with ANSI C header files and you cannot use `-traditional' on such systems to compile files that include any system headers.

In the preprocessor, comments convert to nothing at all, rather than to a space. This allows traditional token concatenation.

In preprocessing directive, the `#' symbol must appear as the first character of a line.

In the preprocessor, macro arguments are recognized within string constants in a macro definition (and their values are stringified, though without additional quote marks, when they appear in such a context). The preprocessor always considers a string constant to end at a newline.

The predefined macro `__STDC__` is not defined when you use `-traditional', but `__GNUC__` is (since the GNU extensions which `__GNUC__` indicates are not affected by `-traditional'). If you need to write header files that work differently depending on whether `-traditional' is in use, by testing both of these predefined macros you can distinguish four situations: GNU C, traditional GNU C, other ANSI C compilers, and other old C compilers. The predefined macro `__STDC_VERSION__` is also not defined when you use `-traditional'. See section 'Standard Predefined Macros' in The C Preprocessor, for more discussion of these and other predefined macros.

The preprocessor considers a string constant to end at a newline (unless the newline is escaped with `\'). (Without `-traditional', string constants can contain the newline character as typed.)

`-traditional-cpp`

Attempt to support some aspects of traditional C preprocessors. This includes the last five items in the table immediately above, but none of the other effects of `-traditional'.

`-fcond-mismatch`

Allow conditional expressions with mismatched types in the second and third arguments. The value of such an expression is void.

`-funsigned-char`

Let the type `char` be unsigned, like `unsigned char`.

Each kind of machine has a default for what `char` should be. It is either like `unsigned char` by default or like `signed char` by default.

Ideally, a portable program should always use `signed char` or `unsigned char` when it depends on the signedness of an object. But many programs have been written to use plain `char` and expect it to be signed, or expect it to be unsigned, depending on the machines they were written for. This option, and its inverse, let you make such a program work with the opposite default.

The type `char` is always a distinct type from each of `signed char` or `unsigned char`, even though its behavior is always just like one of those two.

`-fsigned-char`

Let the type `char` be signed, like `signed char`.

Note that this is equivalent to `-fno-unsigned-char', which is the negative form of `-funsigned-char'. Likewise, the option `-fno-signed-char' is equivalent to `-funsigned-char'.

`-fsigned-bitfields`

`-funsigned-bitfields`
`-fno-signed-bitfields`
`-fno-unsigned-bitfields`

These options control whether a bitfield is signed or unsigned, when the declaration does not use either `signed` or `unsigned`. By default, such a bitfield is signed, because this is consistent: the basic integer types such as `int` are signed types.

However, when `-traditional' is used, bitfields are all unsigned no matter what.

`-fwritable-strings`

Store string constants in the writable data segment and don't uniquize them. This is for compatibility with old programs which assume they can write into string constants. The option `-traditional' also has this effect.

Writing into string constants is a very bad idea; "constants" should be constant.

`-fallow-single-precision`

Do not promote single precision math operations to double precision, even when compiling with `-traditional'.

Traditional K&R C promotes all floating point operations to double precision, regardless of the sizes of the operands. On the architecture for which you are compiling, single precision may be faster than double precision. If you must use `-traditional', but want to use single precision operations when the operands are single precision, use this option. This option has no effect when compiling with ANSI or GNU C conventions (the default).

Options Controlling C++ Dialect

This section describes the command-line options that are only meaningful for C++ programs; but you can also use most of the GNU compiler options regardless of what language your program is in. For example, you might compile a file `firstClass.C` like this:

```
g++ -g -felide-constructors -O -c firstClass.C
```

In this example, only `-felide-constructors' is an option meant only for C++ programs; you can use the other options with any language supported by GNU CC.

Here is a list of options that are *only* for compiling C++ programs:

`-fno-access-control`

Turn off all access checking. This switch is mainly useful for working around bugs in the access control code.

`-fall-virtual`

Treat all possible member functions as virtual, implicitly. All member functions (except for constructor functions and `new` or `delete` member operators) are treated as virtual functions of the class where they appear.

This does not mean that all calls to these member functions will be made through the internal table

of virtual functions. Under some circumstances, the compiler can determine that a call to a given virtual function can be made directly; in these cases the calls are direct in any case.

`-fcheck-new`

Check that the pointer returned by `operator new` is non-null before attempting to modify the storage allocated. The current Working Paper requires that `operator new` never return a null pointer, so this check is normally unnecessary.

`-fconserve-space`

Put uninitialized or runtime-initialized global variables into the common segment, as C does. This saves space in the executable at the cost of not diagnosing duplicate definitions. If you compile with this flag and your program mysteriously crashes after `main()` has completed, you may have an object that is being destroyed twice because two definitions were merged.

`-fdollars-in-identifiers`

Accept `$' in identifiers. You can also explicitly prohibit use of `$' with the option `-fno-dollars-in-identifiers'. (GNU C++ allows `$' by default on some target systems but not others.) Traditional C allowed the character `$' to form part of identifiers. However, ANSI C and C++ forbid `$' in identifiers.

`-fenum-int-equiv`

Anachronistically permit implicit conversion of `int` to enumeration types. Current C++ allows conversion of `enum` to `int`, but not the other way around.

`-fexternal-templates`

Cause template instantiations to obey `#pragma interface' and `implementation'; template instances are emitted or not according to the location of the template definition. See section Where's the Template?, for more information.

`-falt-external-templates`

Similar to -fexternal-templates, but template instances are emitted or not according to the place where they are first instantiated. See section Where's the Template?, for more information.

`-fno-gnu-keywords`

Do not recognize `classof`, `headof`, `signature`, `sigof` or `typeof` as a keyword, so that code can use these words as identifiers. You can use the keywords `__classof__`, `__headof__`, `__signature__`, `__sigof__`, and `__typeof__` instead. `-ansi' implies `-fno-gnu-keywords'.

`-fno-implicit-templates`

Never emit code for templates which are instantiated implicitly (i.e. by use); only emit code for explicit instantiations. See section Where's the Template?, for more information.

`-fhandle-signatures`

Recognize the `signature` and `sigof` keywords for specifying abstract types. The default (`-fno-handle-signatures') is not to recognize them. See section Type Abstraction using Signatures.

`-fhuge-objects`

Support virtual function calls for objects that exceed the size representable by a `short int'. Users should not use this flag by default; if you need to use it, the compiler will tell you so. If you compile any of your code with this flag, you must compile *all* of your code with this flag (including libg++,

if you use it).

This flag is not useful when compiling with -fvtable-thunks.

`-fno-implement-inlines`

To save space, do not emit out-of-line copies of inline functions controlled by `#pragma implementation'. This will cause linker errors if these functions are not inlined everywhere they are called.

`-fmemoize-lookups`

`-fsave-memoized`

Use heuristics to compile faster. These heuristics are not enabled by default, since they are only effective for certain input files. Other input files compile more slowly.

The first time the compiler must build a call to a member function (or reference to a data member), it must (1) determine whether the class implements member functions of that name; (2) resolve which member function to call (which involves figuring out what sorts of type conversions need to be made); and (3) check the visibility of the member function to the caller. All of this adds up to slower compilation. Normally, the second time a call is made to that member function (or reference to that data member), it must go through the same lengthy process again. This means that code like this:

```
cout << "This " << p << " has " << n << " legs.\n";
```

makes six passes through all three steps. By using a software cache, a "hit" significantly reduces this cost. Unfortunately, using the cache introduces another layer of mechanisms which must be implemented, and so incurs its own overhead. `-fmemoize-lookups' enables the software cache.

Because access privileges (visibility) to members and member functions may differ from one function context to the next, G++ may need to flush the cache. With the `-fmemoize-lookups' flag, the cache is flushed after every function that is compiled. The `-fsave-memoized' flag enables the same software cache, but when the compiler determines that the context of the last function compiled would yield the same access privileges of the next function to compile, it preserves the cache. This is most helpful when defining many member functions for the same class: with the exception of member functions which are friends of other classes, each member function has exactly the same access privileges as every other, and the cache need not be flushed.

The code that implements these flags has rotted; you should probably avoid using them.

`-fstrict-prototype`

Within an `extern "C"' linkage specification, treat a function declaration with no arguments, such as `int foo ();', as declaring the function to take no arguments. Normally, such a declaration means that the function `foo` can take any combination of arguments, as in C. `-pedantic' implies `-fstrict-prototype' unless overridden with `-fno-strict-prototype'.

This flag no longer affects declarations with C++ linkage.

`-fno-nonnull-objects`

Don't assume that a reference is initialized to refer to a valid object. Although the current C++ Working Paper prohibits null references, some old code may rely on them, and you can use

`-fno-nonnull-objects' to turn on checking.

At the moment, the compiler only does this checking for conversions to virtual base classes.

`-foperator-names`

Recognize the operator name keywords `and`, `bitand`, `bitor`, `compl`, `not`, `or` and `xor` as synonyms for the symbols they refer to. `-ansi' implies `-foperator-names'.

`-fthis-is-variable`

Permit assignment to `this`. The incorporation of user-defined free store management into C++ has made assignment to `this' an anachronism. Therefore, by default it is invalid to assign to `this` within a class member function; that is, GNU C++ treats `this' in a member function of class `X` as a non-lvalue of type `X *'. However, for backwards compatibility, you can make it valid with `-fthis-is-variable'.

`-fvtable-thunks`

Use `thunks' to implement the virtual function dispatch table (`vtable'). The traditional (cfront-style) approach to implementing vtables was to store a pointer to the function and two offsets for adjusting the `this' pointer at the call site. Newer implementations store a single pointer to a `thunk' function which does any necessary adjustment and then calls the target function.

This option also enables a heuristic for controlling emission of vtables; if a class has any non-inline virtual functions, the vtable will be emitted in the translation unit containing the first one of those.

`-nostdinc++`

Do not search for header files in the standard directories specific to C++, but do still search the other standard directories. (This option is used when building libg++.)

`-traditional`

For C++ programs (in addition to the effects that apply to both C and C++), this has the same effect as `-fthis-is-variable'. See section Options Controlling C Dialect.

In addition, these optimization, warning, and code generation options have meanings only for C++ programs:

`-fno-default-inline`

Do not assume `inline' for functions defined inside a class scope. See section Options That Control Optimization.

`-Wenum-clash`

`-Woverloaded-virtual`

`-Wtemplate-debugging`

Warnings that apply only to C++ programs. See section Options to Request or Suppress Warnings.

`+en`

Control how virtual function definitions are used, in a fashion compatible with `cfront` 1.x. See section Options for Code Generation Conventions.

Options to Request or Suppress Warnings

Warnings are diagnostic messages that report constructions which are not inherently erroneous but which are risky or suggest there may have been an error.

You can request many specific warnings with options beginning `-W', for example `-Wimplicit' to request warnings on implicit declarations. Each of these specific warning options also has a negative form beginning `-Wno-' to turn off warnings; for example, `-Wno-implicit'. This manual lists only one of the two forms, whichever is not the default.

These options control the amount and kinds of warnings produced by GNU CC:

`-fsyntax-only`

Check the code for syntax errors, but don't do anything beyond that.

`-pedantic`

Issue all the warnings demanded by strict ANSI standard C; reject all programs that use forbidden extensions.

Valid ANSI standard C programs should compile properly with or without this option (though a rare few will require `-ansi'). However, without this option, certain GNU extensions and traditional C features are supported as well. With this option, they are rejected.

`-pedantic' does not cause warning messages for use of the alternate keywords whose names begin and end with `__'. Pedantic warnings are also disabled in the expression that follows `__extension__`. However, only system header files should use these escape routes; application programs should avoid them. See section Alternate Keywords.

This option is not intended to be *useful*; it exists only to satisfy pedants who would otherwise claim that GNU CC fails to support the ANSI standard.

Some users try to use `-pedantic' to check programs for strict ANSI C conformance. They soon find that it does not do quite what they want: it finds some non-ANSI practices, but not all--only those for which ANSI C *requires* a diagnostic.

A feature to report any failure to conform to ANSI C might be useful in some instances, but would require considerable additional work and would be quite different from `-pedantic'. We recommend, rather, that users take advantage of the extensions of GNU C and disregard the limitations of other compilers. Aside from certain supercomputers and obsolete small machines, there is less and less reason ever to use any other C compiler other than for bootstrapping GNU CC.

`-pedantic-errors`

Like `-pedantic', except that errors are produced rather than warnings.

`-w`

Inhibit all warning messages.

`-Wno-import`

Inhibit warning messages about the use of `#import'.

`-Wchar-subscripts`

Warn if an array subscript has type `char`. This is a common cause of error, as programmers often forget that this type is signed on some machines.

`-Wcomment`

Warn whenever a comment-start sequence `/*' appears in a comment.

`-Wformat`

Check calls to `printf` and `scanf`, etc., to make sure that the arguments supplied have types appropriate to the format string specified.

`-Wimplicit`

Warn whenever a function or parameter is implicitly declared.

`-Wparentheses`

Warn if parentheses are omitted in certain contexts, such as when there is an assignment in a context where a truth value is expected, or when operators are nested whose precedence people often get confused about.

`-Wreturn-type`

Warn whenever a function is defined with a return-type that defaults to `int`. Also warn about any `return` statement with no return-value in a function whose return-type is not `void`.

`-Wswitch`

Warn whenever a `switch` statement has an index of enumeral type and lacks a `case` for one or more of the named codes of that enumeration. (The presence of a `default` label prevents this warning.) `case` labels outside the enumeration range also provoke warnings when this option is used.

`-Wtrigraphs`

Warn if any trigraphs are encountered (assuming they are enabled).

`-Wunused`

Warn whenever a variable is unused aside from its declaration, whenever a function is declared static but never defined, whenever a label is declared but not used, and whenever a statement computes a result that is explicitly not used.

To suppress this warning for an expression, simply cast it to void. For unused variables and parameters, use the `unused' attribute (see section Specifying Attributes of Variables).

`-Wuninitialized`

An automatic variable is used without first being initialized.

These warnings are possible only in optimizing compilation, because they require data flow information that is computed only when optimizing. If you don't specify `-O', you simply won't get these warnings.

These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared `volatile`, or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers.

Note that there may be no warning about a variable that is used only to compute a value that itself is

never used, because such computations may be deleted by data flow analysis before the warnings are printed.

These warnings are made optional because GNU CC is not smart enough to see all the reasons why the code might be correct despite appearing to have an error. Here is one example of how this can happen:

```
{
  int x;
  switch (y)
    {
    case 1: x = 1;
      break;
    case 2: x = 4;
      break;
    case 3: x = 5;
    }
  foo (x);
}
```

If the value of `y` is always 1, 2 or 3, then `x` is always initialized, but GNU CC doesn't know this. Here is another common case:

```
{
  int save_y;
  if (change_y) save_y = y, y = new_y;
  ...
  if (change_y) y = save_y;
}
```

This has no bug because `save_y` is used only if it is set.

Some spurious warnings can be avoided if you declare all the functions you use that never return as `noreturn`. See section Declaring Attributes of Functions.

`-Wenum-clash`

Warn about conversion between different enumeration types. (C++ only).

`-Wreorder (C++ only)`

Warn when the order of member initializers given in the code does not match the order in which they must be executed. For instance:

```
struct A {
  int i;
  int j;
  A(): j (0), i (1) { }
};
```

Here the compiler will warn that the member initializers for `i' and `j' will be rearranged to mat the declaration order of the members.

`-Wtemplate-debugging`

When using templates in a C++ program, warn if debugging is not yet fully available (C++ only).

`-Wall`

All of the above `-W' options combined. These are all the options which pertain to usage that we recommend avoiding and that we believe is easy to avoid, even in conjunction with macros.

The remaining `-W...' options are not implied by `-Wall' because they warn about constructions that we consider reasonable to use, on occasion, in clean programs.

`-W`

Print extra warning messages for these events:

A nonvolatile automatic variable might be changed by a call to `longjmp`. These warnings as well are possible only in optimizing compilation.

The compiler sees only the calls to `setjmp`. It cannot know where `longjmp` will be called; in fact, a signal handler could call it at any point in the code. As a result, you may get a warning even when there is in fact no problem because `longjmp` cannot in fact be called at the place which would cause a problem.

A function can return either with or without a value. (Falling off the end of the function body is considered returning without a value.) For example, this function would evoke such a warning:

```
foo (a)
{
  if (a > 0)
    return a;
}
```

An expression-statement contains no side effects.

An unsigned value is compared against zero with `<' or `<='.

A comparison like `x<=y<=z' appears; this is equivalent to `(x<=y ? 1 : 0) <= z', which is a different interpretation from that of ordinary mathematical notation.

Storage-class specifiers like `static` are not the first things in a declaration. According to the C Standard, this usage is obsolescent.

If `-Wall' or `-Wunused' is also specified, warn about unused arguments.

An aggregate has a partly bracketed initializer. For example, the following code would evoke such a warning, because braces are missing around the initializer for `x.h`:

```
struct s { int f, g; };
struct t { struct s h; int i; };
struct t x = { 1, 2, 3 };
```

`-Wtraditional`

Warn about certain constructs that behave differently in traditional and ANSI C.

Macro arguments occurring within string constants in the macro body. These would substitute the argument in traditional C, but are part of the constant in ANSI C.

A function declared external in one block and then used after the end of the block.

A `switch` statement has an operand of type `long`.

`-Wshadow`

Warn whenever a local variable shadows another local variable.

`-Wid-clash-len`

Warn whenever two distinct identifiers match in the first len characters. This may help you prepare a program that will compile with certain obsolete, brain-damaged compilers.

`-Wlarger-than-len`

Warn whenever an object of larger than len bytes is defined.

`-Wpointer-arith`

Warn about anything that depends on the "size of" a function type or of `void`. GNU C assigns these types a size of 1, for convenience in calculations with `void *` pointers and pointers to functions.

`-Wbad-function-cast`

Warn whenever a function call is cast to a non-matching type. For example, warn if `int malloc()` is cast to `anything *`.

`-Wcast-qual`

Warn whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warn if a `const char *` is cast to an ordinary `char *`.

`-Wcast-align`

Warn whenever a pointer is cast such that the required alignment of the target is increased. For example, warn if a `char *` is cast to an `int *` on machines where integers can only be accessed at two- or four-byte boundaries.

`-Wwrite-strings`

Give string constants the type `const char[length]` so that copying the address of one into a non-`const` `char *` pointer will get a warning. These warnings will help you find at compile time code that can try to write into a string constant, but only if you have been very careful about using `const` in declarations and prototypes. Otherwise, it will just be a nuisance; this is why we did not make `-Wall' request these warnings.

`-Wconversion`

Warn if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, and conversions changing the width or signedness of a fixed point argument except when the same as the default promotion.

Also, warn if a negative integer constant expression is implicitly converted to an unsigned type. For example, warn about the assignment `x = -1` if `x` is unsigned. But do not warn about explicit casts like `(unsigned) -1`.

`-Waggregate-return`

Warn if any functions that return structures or unions are defined or called. (In languages where you can return an array, this also elicits a warning.)

`-Wstrict-prototypes`

Warn if a function is declared or defined without specifying the argument types. (An old-style function definition is permitted without a warning if preceded by a declaration which specifies the argument types.)

`-Wmissing-prototypes`

Warn if a global function is defined without a previous prototype declaration. This warning is issued even if the definition itself provides a prototype. The aim is to detect global functions that fail to be declared in header files.

`-Wmissing-declarations`

Warn if a global function is defined without a previous declaration. Do so even if the definition itself provides a prototype. Use this option to detect global functions that are not declared in header files.

`-Wredundant-decls`

Warn if anything is declared more than once in the same scope, even in cases where multiple declaration is valid and changes nothing.

`-Wnested-externs`

Warn if an `extern` declaration is encountered within an function.

`-Winline`

Warn if a function can not be inlined, and either it was declared as inline, or else the `-finline-functions' option was given.

`-Woverloaded-virtual`

Warn when a derived class function declaration may be an error in defining a virtual function (C++ only). In a derived class, the definitions of virtual functions must match the type signature of a virtual function declared in the base class. With this option, the compiler warns when you define a function with the same name as a virtual function, but with a type signature that does not match any declarations from the base class.

`-Wsynth (C++ only)`

Warn when g++'s synthesis behavior does not match that of cfront. For instance:

```
struct A {
  operator int ();
  A& operator = (int);
};

main ()
{
  A a,b;
  a = b;
}
```

In this example, g++ will synthesize a default `A& operator = (const A&);', while cfront will us user-defined `operator ='.

`-Werror`

Make all warnings into errors.

Options for Debugging Your Program or GNU CC

GNU CC has various special options that are used for debugging either your program or GCC:

`-g`

Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF). GDB can work with this debugging information.

On most systems that use stabs format, `-g' enables use of extra debugging information that only GDB can use; this extra information makes debugging work better in GDB but will probably make other debuggers crash or refuse to read the program. If you want to control for certain whether to generate the extra information, use `-gstabs+', `-gstabs', `-gxcoff+', `-gxcoff', `-gdwarf+', or `-gdwarf' (see below).

Unlike most other C compilers, GNU CC allows you to use `-g' with `-O'. The shortcuts taken by optimized code may occasionally produce surprising results: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values were already at hand; some statements may execute in different places because they were moved out of loops.

Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

The following options are useful when GNU CC is generated with the capability for more than one debugging format.

`-ggdb`

Produce debugging information in the native format (if that is supported), including GDB extensions if at all possible.

`-gstabs`

Produce debugging information in stabs format (if that is supported), without GDB extensions. This is the format used by DBX on most BSD systems. On MIPS, Alpha and System V Release 4 systems this option produces stabs debugging output which is not understood by DBX or SDB. On System V Release 4 systems this option requires the GNU assembler.

`-gstabs+`

Produce debugging information in stabs format (if that is supported), using GNU extensions understood only by the GNU debugger (GDB). The use of these extensions is likely to make other debuggers crash or refuse to read the program.

`-gcoff`

Produce debugging information in COFF format (if that is supported). This is the format used by

SDB on most System V systems prior to System V Release 4.

`-gxcoff`

Produce debugging information in XCOFF format (if that is supported). This is the format used by the DBX debugger on IBM RS/6000 systems.

`-gxcoff+`

Produce debugging information in XCOFF format (if that is supported), using GNU extensions understood only by the GNU debugger (GDB). The use of these extensions is likely to make other debuggers crash or refuse to read the program, and may cause assemblers other than the GNU assembler (GAS) to fail with an error.

`-gdwarf`

Produce debugging information in DWARF format (if that is supported). This is the format used by SDB on most System V Release 4 systems.

`-gdwarf+`

Produce debugging information in DWARF format (if that is supported), using GNU extensions understood only by the GNU debugger (GDB). The use of these extensions is likely to make other debuggers crash or refuse to read the program.

`-glevel`

`-ggdblevel`

`-gstabslevel`

`-gcofflevel`

`-gxcofflevel`

`-gdwarflevel`

Request debugging information and also use level to specify how much information. The default level is 2.

Level 1 produces minimal information, enough for making backtraces in parts of the program that you don't plan to debug. This includes descriptions of functions and external variables, but no information about local variables and no line numbers.

Level 3 includes extra information, such as all the macro definitions present in the program. Some debuggers support macro expansion when you use `-g3'.

`-p`

Generate extra code to write profile information suitable for the analysis program `prof`. You must use this option when compiling the source files you want data about, and you must also use it when linking.

`-pg`

Generate extra code to write profile information suitable for the analysis program `gprof`. You must use this option when compiling the source files you want data about, and you must also use it when linking.

`-a`

Generate extra code to write profile information for basic blocks, which will record the number of times each basic block is executed, the basic block start address, and the function name containing

the basic block. If `-g' is used, the line number and filename of the start of the basic block will also be recorded. If not overridden by the machine description, the default action is to append to the file `bb.out'.

This data could be analyzed by a program like `tcov`. Note, however, that the format of the data is not what `tcov` expects. Eventually GNU `gprof` should be extended to process this data.

`-dletters`

Says to make debugging dumps during compilation at times specified by letters. This is used for debugging the compiler. The file names for most of the dumps are made by appending a word to the source file name (e.g. `foo.c.rtl' or `foo.c.jump'). Here are the possible letters for use in letters, and their meanings:

`M'

Dump all macro definitions, at the end of preprocessing, and write no output.

`N'

Dump all macro names, at the end of preprocessing.

`D'

Dump all macro definitions, at the end of preprocessing, in addition to normal output.

`y'

Dump debugging information during parsing, to standard error.

`r'

Dump after RTL generation, to `file.rtl'.

`x'

Just generate RTL for a function instead of compiling it. Usually used with `r'.

`j'

Dump after first jump optimization, to `file.jump'.

`s'

Dump after CSE (including the jump optimization that sometimes follows CSE), to `file.cse'.

`L'

Dump after loop optimization, to `file.loop'.

`t'

Dump after the second CSE pass (including the jump optimization that sometimes follows CSE), to `file.cse2'.

`f'

Dump after flow analysis, to `file.flow'.

`c'

Dump after instruction combination, to the file `file.combine'.

`S'

Dump after the first instruction scheduling pass, to `file.sched'.

`l'

Dump after local register allocation, to `file.lreg'.

`g'

Dump after global register allocation, to `file.greg'.

`R'

Dump after the second instruction scheduling pass, to `file.sched2'.

`J'

Dump after last jump optimization, to `file.jump2'.

`d'

Dump after delayed branch scheduling, to `file.dbr'.

`k'

Dump after conversion from registers to stack, to `file.stack'.

`a'

Produce all the dumps listed above.

`m'

Print statistics on memory usage, at the end of the run, to standard error.

`p'

Annotate the assembler output with a comment indicating which pattern and alternative was used.

- -fpretend-float When running a cross-compiler, pretend that the target machine uses the same floating point format as the host machine. This causes incorrect output of the actual floating constants, but the actual instruction sequence will probably be the same as GNU CC would make when running on the target machine.
- -save-temps Store the usual "temporary" intermediate files permanently; place them in the current directory and name them based on the source file. Thus, compiling `foo.c' with `-c -save-temps' would produce files `foo.i' and `foo.s', as well as `foo.o'.
- -print-file-name=library Print the full absolute name of the library file library that would be used when linking--and don't do anything else. With this option, GNU CC does not compile or link anything; it just prints the file name.
- -print-prog-name=program Like `-print-file-name', but searches for a program such as `cpp'.
- -print-libgcc-file-name Same as `-print-file-name=libgcc.a'.

This is useful when you use `-nostdlib' or `-nodefaultlibs' but you do want to link with `libgcc.a'. You can do

```
gcc -nostdlib files... `gcc -print-libgcc-file-name`
```

- -print-search-dirs Print the name of the configured installation directory and a list of program and library directories gcc will search--and don't do anything else.

This is useful when gcc prints the error message `installation problem, cannot exec cpp: No such file or directory'. To resolve this you either need to put `cpp' and the other compiler components where gcc

expects to find them, or you can set the environment variable `GCC_EXEC_PREFIX` to the directory you installed them. Don't forget the trailing '/'. See section Environment Variables Affecting GNU CC

Options That Control Optimization

These options control various sorts of optimizations:

`-O`

`-O1`

Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

Without `-O', the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

Without `-O', the compiler only allocates variables declared `register` in registers. The resulting compiled code is a little worse than produced by PCC without `-O'.

With `-O', the compiler tries to reduce code size and execution time.

When you specify `-O', the compiler turns on `-fthread-jumps' and `-fdefer-pop' on all machines. The compiler turns on `-fdelayed-branch' on machines that have delay slots, and `-fomit-frame-pointer' on machines that can support debugging even without a frame pointer. On some machines the compiler also turns on other flags.

`-O2`

Optimize even more. GNU CC performs nearly all supported optimizations that do not involve a space-speed tradeoff. The compiler does not perform loop unrolling or function inlining when you specify `-O2'. As compared to `-O', this option increases both compilation time and the performance of the generated code.

`-O2' turns on all optional optimizations except for loop unrolling and function inlining. It also turns on frame pointer elimination on machines where doing so does not interfere with debugging.

`-O3`

Optimize yet more. `-O3' turns on all optimizations specified by `-O2' and also turns on the `inline-functions' option.

`-O0`

Do not optimize.

If you use multiple `-O' options, with or without level numbers, the last such option is the one that is effective.

Options of the form `-fflag' specify machine-independent flags. Most flags have both positive and negative forms; the negative form of `-ffoo' would be `-fno-foo'. In the table below, only one of the forms is listed--the one which is not the default. You can figure out the other form by either removing `no-' or

adding it.

`-ffloat-store`

Do not store floating point variables in registers, and inhibit other options that might change whether a floating point value is taken from a register or memory.

This option prevents undesirable excess precision on machines such as the 68000 where the floating registers (of the 68881) keep more precision than a `double` is supposed to have. For most programs, the excess precision does only good, but a few programs rely on the precise definition of IEEE floating point. Use `-ffloat-store' for such programs.

`-fno-default-inline`

Do not make member functions inline by default merely because they are defined inside the class scope (C++ only). Otherwise, when you specify `-O', member functions defined inside class scope are compiled inline by default; i.e., you don't need to add `inline' in front of the member function name.

`-fno-defer-pop`

Always pop the arguments to each function call as soon as that function returns. For machines which must pop arguments after a function call, the compiler normally lets arguments accumulate on the stack for several function calls and pops them all at once.

`-fforce-mem`

Force memory operands to be copied into registers before doing arithmetic on them. This may produce better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register-load. I am interested in hearing about the difference this makes.

`-fforce-addr`

Force memory address constants to be copied into registers before doing arithmetic on them. This may produce better code just as `-fforce-mem' may. I am interested in hearing about the difference this makes.

`-fomit-frame-pointer`

Don't keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an extra register available in many functions. **It also makes debugging impossible on some machines.**

On some machines, such as the Vax, this flag has no effect, because the standard calling sequence automatically handles the frame pointer and nothing is saved by pretending it doesn't exist. The machine-description macro `FRAME_POINTER_REQUIRED` controls whether a target machine supports this flag. See section Register Usage.

`-fno-inline`

Don't pay attention to the `inline` keyword. Normally this option is used to keep the compiler from expanding any functions inline. Note that if you are not optimizing, no functions can be expanded inline.

`-finline-functions`

Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way.

If all calls to a given function are integrated, and the function is declared `static`, then the function is normally not output as assembler code in its own right.

`-fkeep-inline-functions`

Even if all calls to a given function are integrated, and the function is declared `static`, nevertheless output a separate run-time callable version of the function.

`-fno-function-cse`

Do not put function addresses in registers; make each instruction that calls a constant function contain the function's address explicitly.

This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option is not used.

`-ffast-math`

This option allows GCC to violate some ANSI or IEEE rules and/or specifications in the interest of optimizing code for speed. For example, it allows the compiler to assume arguments to the `sqrt` function are non-negative numbers and that no floating-point values are NaNs.

This option should never be turned on by any `-O' option since it can result in incorrect output for programs which depend on an exact implementation of IEEE or ANSI rules/specifications for math functions.

The following options control specific optimizations. The `-O2' option turns on all of these optimizations except `-funroll-loops' and `-funroll-all-loops'. On most machines, the `-O' option turns on the `-fthread-jumps' and `-fdelayed-branch' options, but specific machines may handle it differently.

You can use the following flags in the rare cases when "fine-tuning" of optimizations to be performed is desired.

`-fstrength-reduce`

Perform the optimizations of loop strength reduction and elimination of iteration variables.

`-fthread-jumps`

Perform optimizations where we check to see if a jump branches to a location where another comparison subsumed by the first is found. If so, the first branch is redirected to either the destination of the second branch or a point immediately following it, depending on whether the condition is known to be true or false.

`-fcse-follow-jumps`

In common subexpression elimination, scan through jump instructions when the target of the jump is not reached by any other path. For example, when CSE encounters an `if` statement with an `else` clause, CSE will follow the jump when the condition tested is false.

`-fcse-skip-blocks`

This is similar to `-fcse-follow-jumps', but causes CSE to follow jumps which conditionally skip over blocks. When CSE encounters a simple `if` statement with no else clause, `-fcse-skip-blocks' causes CSE to follow the jump around the body of the `if`.

`-frerun-cse-after-loop`

Re-run common subexpression elimination after loop optimizations has been performed.

`-fexpensive-optimizations`

Perform a number of minor optimizations that are relatively expensive.

`-fdelayed-branch`

If supported for the target machine, attempt to reorder instructions to exploit instruction slots available after delayed branch instructions.

`-fschedule-insns`

If supported for the target machine, attempt to reorder instructions to eliminate execution stalls due to required data being unavailable. This helps machines that have slow floating point or memory load instructions by allowing other instructions to be issued until the result of the load or floating point instruction is required.

`-fschedule-insns2`

Similar to `-fschedule-insns', but requests an additional pass of instruction scheduling after register allocation has been done. This is especially useful on machines with a relatively small number of registers and where memory load instructions take more than one cycle.

`-fcaller-saves`

Enable values to be allocated in registers that will be clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code than would otherwise be produced.

This option is enabled by default on certain machines, usually those which have no call-preserved registers to use instead.

`-funroll-loops`

Perform the optimization of loop unrolling. This is only done for loops whose number of iterations can be determined at compile time or run time. `-funroll-loop' implies both `-fstrength-reduce' and `-frerun-cse-after-loop'.

`-funroll-all-loops`

Perform the optimization of loop unrolling. This is done for all loops and usually makes programs run more slowly. `-funroll-all-loops' implies `-fstrength-reduce' as well as `-frerun-cse-after-loop'.

`-fno-peephole`

Disable any machine-specific peephole optimizations.

Options Controlling the Preprocessor

These options control the C preprocessor, which is run on each C source file before actual compilation.

If you use the `-E' option, nothing is done except preprocessing. Some of these options make sense only together with `-E' because they cause the preprocessor output to be unsuitable for actual compilation.

`-include file`

Process file as input before processing the regular input file. In effect, the contents of file are compiled first. Any `-D' and `-U' options on the command line are always processed before `-include file', regardless of the order in which they are written. All the `-include' and `-imacros' options are processed in the order in which they are written.

`-imacros file`

Process file as input, discarding the resulting output, before processing the regular input file. Because the output generated from file is discarded, the only effect of `-imacros file' is to make the macros defined in file available for use in the main input.

Any `-D' and `-U' options on the command line are always processed before `-imacros file', regardless of the order in which they are written. All the `-include' and `-imacros' options are processed in the order in which they are written.

`-idirafter dir`

Add the directory dir to the second include path. The directories on the second include path are searched when a header file is not found in any of the directories in the main include path (the one that `-I' adds to).

`-iprefix prefix`

Specify prefix as the prefix for subsequent `-iwithprefix' options.

`-iwithprefix dir`

Add a directory to the second include path. The directory's name is made by concatenating prefix and dir, where prefix was specified previously with `-iprefix'. If you have not specified a prefix yet, the directory containing the installed passes of the compiler is used as the default.

`-iwithprefixbefore dir`

Add a directory to the main include path. The directory's name is made by concatenating prefix and dir, as in the case of `-iwithprefix'.

`-isystem dir`

Add a directory to the beginning of the second include path, marking it as a system directory, so that it gets the same special treatment as is applied to the standard system directories.

`-nostdinc`

Do not search the standard system directories for header files. Only the directories you have specified with `-I' options (and the current directory, if appropriate) are searched. See section Options for Directory Search, for information on `-I'.

By using both `-nostdinc' and `-I-', you can limit the include-file search path to only those directories you specify explicitly.

`-undef`

Do not predefine any nonstandard macros. (Including architecture flags).

`-E`

Run only the C preprocessor. Preprocess all the C source files specified and output the results to standard output or to the specified output file.

`-C`

Tell the preprocessor not to discard comments. Used with the `-E' option.

`-P`

Tell the preprocessor not to generate `#line' directives. Used with the `-E' option.

`-M`

Tell the preprocessor to output a rule suitable for `make` describing the dependencies of each ob file. For each source file, the preprocessor outputs one `make`-rule whose target is the object file name for that source file and whose dependencies are all the `#include` header files it uses. This rule may be a single line or may be continued with `\'-newline if it is long. The list of rules is printed on standard output instead of the preprocessed C program.

`-M' implies `-E'.

Another way to specify output of a `make` rule is by setting the environment variable `DEPENDENCIES_OUTPUT` (see section Environment Variables Affecting GNU CC).

`-MM`

Like `-M' but the output mentions only the user header files included with `#include "file"'. System header files included with `#include <file>' are omitted.

`-MD`

Like `-M' but the dependency information is written to a file made by replacing ".c" with ".d" at the end of the input file names. This is in addition to compiling the file as specified---`-MD' does not inhibit ordinary compilation the way `-M' does.

In Mach, you can use the utility `md` to merge multiple dependency files into a single dependency file suitable for using with the `make' command.

`-MMD`

Like `-MD' except mention only user header files, not system header files.

`-MG`

Treat missing header files as generated files and assume they live in the same directory as the source file. If you specify `-MG', you must also specify either `-M' or `-MM'. `-MG' is not supported with `-MD' or `-MMD'.

`-H`

Print the name of each header file used, in addition to other normal activities.

`-Aquestion(answer)`

Assert the answer answer for question, in case it is tested with a preprocessing conditional such as `#if #question(answer)'. `-A-' disables the standard assertions that normally describe the target machine.

`-Dmacro`

Define macro macro with the string `1' as its definition.

`-Dmacro=defn`

Define macro macro as defn. All instances of `-D' on the command line are processed before any `-U' options.

`-Umacro`

Undefine macro macro. `-U' options are evaluated after all `-D' options, but before any `-include' and `-imacros' options.

`-dM`

Tell the preprocessor to output only a list of the macro definitions that are in effect at the end of

preprocessing. Used with the `-E' option.

`-dD`

Tell the preprocessing to pass all macro definitions into the output, in their proper sequence in the rest of the output.

`-dN`

Like `-dD' except that the macro arguments and contents are omitted. Only `#define name' is included in the output.

`-trigraphs`

Support ANSI C trigraphs. The `-ansi' option also has this effect.

`-Wp,option`

Pass option as an option to the preprocessor. If option contains commas, it is split into multiple options at the commas.

Passing Options to the Assembler

You can pass options to the assembler.

`-Wa,option`

Pass option as an option to the assembler. If option contains commas, it is split into multiple options at the commas.

Options for Linking

These options come into play when the compiler links object files into an executable output file. They are meaningless if the compiler is not doing a link step.

`object-file-name`

A file name that does not end in a special recognized suffix is considered to name an object file or library. (Object files are distinguished from libraries by the linker according to the file contents.) If linking is done, these object files are used as input to the linker.

`-c`

`-S`

`-E`

If any of these options is used, then the linker is not run, and object file names should not be used as arguments. See section Options Controlling the Kind of Output.

`-llibrary`

Search the library named library when linking.

It makes a difference where in the command you write this option; the linker searches processes libraries and object files in the order they are specified. Thus, `foo.o -lz bar.o' searches library `z' after file `foo.o' but before `bar.o'. If `bar.o' refers to functions in `z', those functions may not be loaded.

The linker searches a standard list of directories for the library, which is actually a file named `liblibrary.a'. The linker then uses this file as if it had been specified precisely by name.

The directories searched include several standard system directories plus any that you specify with `-L'.

Normally the files found this way are library files--archive files whose members are object files. The linker handles an archive file by scanning through it for members which define symbols that have so far been referenced but not defined. But if the file that is found is an ordinary object file, it is linked in the usual fashion. The only difference between using an `-l' option and specifying a file name is that `-l' surrounds library with `lib' and `.a' and searches several directories.

`-lobjc`

You need this special case of the `-l' option in order to link an Objective C program.

`-nostartfiles`

Do not use the standard system startup files when linking. The standard system libraries are used normally, unless `-nostdlib` or `-nodefaultlibs` is used.

`-nodefaultlibs`

Do not use the standard system libraries when linking. Only the libraries you specify will be passed to the linker. The standard startup files are used normally, unless `-nostartfiles` is used.

`-nostdlib`

Do not use the standard system startup files or libraries when linking. No startup files and only the libraries you specify will be passed to the linker.

One of the standard libraries bypassed by `-nostdlib' and `-nodefaultlibs' is `libgcc.a', a library of internal subroutines that GNU CC uses to overcome shortcomings of particular machines, or special needs for some languages. (See section Interfacing to GNU CC Output, for more discussion of `libgcc.a'.) In most cases, you need `libgcc.a' even when you want to avoid other standard libraries. In other words, when you specify `-nostdlib' or `-nodefaultlibs' you should usually specify `-lgcc' as well. This ensures that you have no unresolved references to internal GNU CC library subroutines. (For example, `__main', used to ensure C++ constructors will be called; see section `collect2`.)

`-s`

Remove all symbol table and relocation information from the executable.

`-static`

On systems that support dynamic linking, this prevents linking with the shared libraries. On other systems, this option has no effect.

`-shared`

Produce a shared object which can then be linked with other objects to form an executable. Only a few systems support this option.

`-symbolic`

Bind references to global symbols when building a shared object. Warn about any unresolved references (unless overridden by the link editor option `-Xlinker -z -Xlinker defs'). Only a few systems support this option.

`-Xlinker option`

Pass option as an option to the linker. You can use this to supply system-specific linker options which GNU CC does not know how to recognize.

If you want to pass an option that takes an argument, you must use `-Xlinker' twice, once for the option and once for the argument. For example, to pass `-assert definitions', you must write `-Xlinker -assert -Xlinker definitions'. It does not work to write `-Xlinker "-assert definitions"', because this passes the entire string as a single argument, which is not what the linker expects.

`-Wl,option`

Pass option as an option to the linker. If option contains commas, it is split into multiple options at the commas.

`-u symbol`

Pretend the symbol symbol is undefined, to force linking of library modules to define it. You can use `-u' multiple times with different symbols to force loading of additional library modules.

Options for Directory Search

These options specify directories to search for header files, for libraries and for parts of the compiler:

`-Idir`

Add the directory directory to the head of the list of directories to be searched for header files. This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one `-I' option, the directories are scanned in left-to-right order; the standard system directories come after.

`-I-`

Any directories you specify with `-I' options before the `-I-' option are searched only for the case of `#include "file"'; they are not searched for `#include <file>'.

If additional directories are specified with `-I' options after the `-I-', these directories are searched for all `#include' directives. (Ordinarily *all* `-I' directories are used this way.)

In addition, the `-I-' option inhibits the use of the current directory (where the current input file came from) as the first search directory for `#include "file"'. There is no way to override this effect of `-I-'. With `-I.' you can specify searching the directory which was current when the compiler was invoked. That is not exactly the same as what the preprocessor does by default, but it is often satisfactory.

`-I-' does not inhibit the use of the standard system directories for header files. Thus, `-I-' and `-nostdinc' are independent.

`-Ldir`

Add directory dir to the list of directories to be searched for `-l'.

`-Bprefix`

This option specifies where to find the executables, libraries, include files, and data files of the compiler itself.

The compiler driver program runs one or more of the subprograms `cpp', `cc1', `as' and `ld'. It tries prefix as a prefix for each program it tries to run, both with and without `machine/version/' (see section Specifying Target Machine and Compiler Version).

For each subprogram to be run, the compiler driver first tries the `-B' prefix, if any. If that name is not found, or if `-B' was not specified, the driver tries two standard prefixes, which are `/usr/lib/gcc/' and `/usr/local/lib/gcc-lib/'. If neither of those results in a file name that is found, the unmodified program name is searched for using the directories specified in your `PATH' environment variable.

`-B' prefixes that effectively specify directory names also apply to libraries in the linker, because the compiler translates these options into `-L' options for the linker. They also apply to includes files in the preprocessor, because the compiler translates these options into `-isystem' options for the preprocessor. In this case, the compiler appends `include' to the prefix.

The run-time support file `libgcc.a' can also be searched for using the `-B' prefix, if needed. If it is not found there, the two standard prefixes above are tried, and that is all. The file is left out of the link if it is not found by those means.

Another way to specify a prefix much like the `-B' prefix is to use the environment variable `GCC_EXEC_PREFIX`. See section Environment Variables Affecting GNU CC.

Specifying Target Machine and Compiler Version

By default, GNU CC compiles code for the same type of machine that you are using. However, it can also be installed as a cross-compiler, to compile for some other type of machine. In fact, several different configurations of GNU CC, for different target machines, can be installed side by side. Then you specify which one to use with the `-b' option.

In addition, older and newer versions of GNU CC can be installed side by side. One of them (probably the newest) will be the default, but you may sometimes wish to use another.

`-b machine`

The argument machine specifies the target machine for compilation. This is useful when you have installed GNU CC as a cross-compiler.

The value to use for machine is the same as was specified as the machine type when configuring GNU CC as a cross-compiler. For example, if a cross-compiler was configured with `configure i386v', meaning to compile for an 80386 running System V, then you would specify `-b i386v' to run that cross compiler.

When you do not specify `-b', it normally means to compile for the same type of machine that you are using.

`-V version`

The argument version specifies which version of GNU CC to run. This is useful when multiple versions are installed. For example, version might be `2.0', meaning to run GNU CC version 2.0.

The default version, when you do not specify `-V', is the last version of GNU CC that you installed.

The `-b' and `-V' options actually work by controlling part of the file name used for the executable files and libraries used for compilation. A given version of GNU CC, for a given target machine, is normally kept in the directory `/usr/local/lib/gcc-lib/machine/version'.

Thus, sites can customize the effect of `-b' or `-V' either by changing the names of these directories or adding alternate names (or symbolic links). If in directory `/usr/local/lib/gcc-lib/' the file `80386' is a link to the file `i386v', then `-b 80386' becomes an alias for `-b i386v'.

In one respect, the `-b' or `-V' do not completely change to a different compiler: the top-level driver program `gcc' that you originally invoked continues to run and invoke the other executables (preprocessor, compiler per se, assembler and linker) that do the real work. However, since no real work is done in the driver program, it usually does not matter that the driver program in use is not the one for the specified target and version.

The only way that the driver program depends on the target machine is in the parsing and handling of special machine-specific options. However, this is controlled by a file which is found, along with the other executables, in the directory for the specified version and target machine. As a result, a single installed driver program adapts to any specified target machine and compiler version.

The driver program executable does control one significant thing, however: the default version and target machine. Therefore, you can install different instances of the driver program, compiled for different targets or versions, under different names.

For example, if the driver for version 2.0 is installed as `ogcc' and that for version 2.1 is installed as `gcc', then the command `gcc' will use version 2.1 by default, while `ogcc' will use 2.0 by default. However, you can choose either version with either command with the `-V' option.

Hardware Models and Configurations

Earlier we discussed the standard option `-b' which chooses among different installed compilers for completely different target machines, such as Vax vs. 68000 vs. 80386.

In addition, each of these target machine types can have its own special options, starting with `-m', to choose among various hardware models or configurations--for example, 68010 vs 68020, floating coprocessor or none. A single installed version of the compiler can compile for any model or configuration, according to the options specified.

Some configurations of the compiler also support additional special options, usually for compatibility with other compilers on the same platform.

These options are defined by the macro `TARGET_SWITCHES' in the machine description. The default for the options is also defined by that macro, which enables you to change the defaults.

M680x0 Options

These are the `-m' options defined for the 68000 series. The default values for these options depends on which style of 68000 was selected when the compiler was configured; the defaults for the most common choices are given below.

`-m68000`

`-mc68000`

Generate output for a 68000. This is the default when the compiler is configured for 68000-based systems.

`-m68020`

`-mc68020`

Generate output for a 68020. This is the default when the compiler is configured for 68020-based systems.

`-m68881`

Generate output containing 68881 instructions for floating point. This is the default for most 68020 systems unless `-nfp' was specified when the compiler was configured.

`-m68030`

Generate output for a 68030. This is the default when the compiler is configured for 68030-based systems.

`-m68040`

Generate output for a 68040. This is the default when the compiler is configured for 68040-based systems.

This option inhibits the use of 68881/68882 instructions that have to be emulated by software on the 68040. If your 68040 does not have code to emulate those instructions, use `-m68040'.

`-m68020-40`

Generate output for a 68040, without using any of the new instructions. This results in code which can run relatively efficiently on either a 68020/68881 or a 68030 or a 68040. The generated code does use the 68881 instructions that are emulated on the 68040.

`-mfpa`

Generate output containing Sun FPA instructions for floating point.

`-msoft-float`

Generate output containing library calls for floating point. **Warning:** the requisite libraries are not available for all m68k targets. Normally the facilities of the machine's usual C compiler are used, but this can't be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation. The embedded targets `m68k-*-aout' and `m68k-*-coff' do provide software floating point support.

`-mshort`

Consider type `int` to be 16 bits wide, like `short int`.

`-mnobitfield`

Do not use the bit-field instructions. The `-m68000' option implies `-mnobitfield'.

`-mbitfield`

Do use the bit-field instructions. The `-m68020' option implies `-mbitfield'. This is the default if you use a configuration designed for a 68020.

`-mrtd`

Use a different function-calling convention, in which functions that take a fixed number of arguments return with the `rtd` instruction, which pops their arguments while returning. This sa one instruction in the caller since there is no need to pop the arguments there.

This calling convention is incompatible with the one normally used on Unix, so you cannot use it if you need to call libraries compiled with the Unix compiler.

Also, you must provide function prototypes for all functions that take variable numbers of arguments (including `printf`); otherwise incorrect code will be generated for calls to those functions.

In addition, seriously incorrect code will result if you call a function with too many arguments. (Normally, extra arguments are harmlessly ignored.)

The `rtd` instruction is supported by the 68010 and 68020 processors, but not by the 68000.

VAX Options

These `-m' options are defined for the Vax:

`-munix`

Do not output certain jump instructions (`aobleq` and so on) that the Unix assembler for the Vax cannot handle across long ranges.

`-mgnu`

Do output those jump instructions, on the assumption that you will assemble with the GNU assembler.

`-mg`

Output code for g-format floating point numbers instead of d-format.

SPARC Options

These `-m' switches are supported on the SPARC:

`-mno-app-regs`

`-mapp-regs`

Specify `-mapp-regs' to generate output using the global registers 2 through 4, which the SPARC SVR4 ABI reserves for applications. This is the default.

To be fully SVR4 ABI compliant at the cost of some performance loss, specify `-mno-app-regs'. You should compile libraries and system software with this option.

`-mfpu`

`-mhard-float`

Generate output containing floating point instructions. This is the default.

`-mno-fpu`

`-msoft-float`

Generate output containing library calls for floating point. **Warning:** the requisite libraries are not

available for all SPARC targets. Normally the facilities of the machine's usual C compiler are u
but this cannot be done directly in cross-compilation. You must make your own arrangements t
provide suitable library functions for cross-compilation. The embedded targets `sparc-*-aout' and
`sparclite-*-*' do provide software floating point support.

`-msoft-float' changes the calling convention in the output file; therefore, it is only useful if you compile *all* of a program with this option. In particular, you need to compile `libgcc.a', the library that comes with GNU CC, with `-msoft-float' in order for this to work.

`-mhard-quad-float`

Generate output containing quad-word (long double) floating point instructions.

`-msoft-quad-float`

Generate output containing library calls for quad-word (long double) floating point instructions. The functions called are those specified in the SPARC ABI. This is the default.

As of this writing, there are no sparc implementations that have hardware support for the quad-word floating point instructions. They all invoke a trap handler for one of these instructions, and then the trap handler emulates the effect of the instruction. Because of the trap handler overhead, this is much slower than calling the ABI library routines. Thus the `-msoft-quad-float' option is the default.

`-mno-epilogue`

`-mepilogue`

With `-mepilogue' (the default), the compiler always emits code for function exit at the end of each function. Any function exit in the middle of the function (such as a return statement in C) will generate a jump to the exit code at the end of the function.

With `-mno-epilogue', the compiler tries to emit exit code inline at every function exit.

`-mno-flat`

`-mflat`

With `-mflat', the compiler does not generate save/restore instructions and will use a "flat" or single register window calling convention. This model uses %i7 as the frame pointer and is compatible with the normal register window model. Code from either may be intermixed although debugger support is still incomplete. The local registers and the input registers (0-5) are still treated as "call saved" registers and will be saved on the stack as necessary.

With `-mno-flat' (the default), the compiler emits save/restore instructions (except for leaf functions) and is the normal mode of operation.

`-mno-unaligned-doubles`

`-munaligned-doubles`

Assume that doubles have 8 byte alignment. This is the default.

With `-munaligned-doubles', GNU CC assumes that doubles have 8 byte alignment only if they are contained in another type, or if they have an absolute address. Otherwise, it assumes they have 4 byte alignment. Specifying this option avoids some rare compatibility problems with code generated by other compilers. It is not the default because it results in a performance loss, especially for floating point code.

`-mv8`

`-msparclite`

These two options select variations on the SPARC architecture.

By default (unless specifically configured for the Fujitsu SPARClite), GCC generates code for the v7 variant of the SPARC architecture.

`-mv8' will give you SPARC v8 code. The only difference from v7 code is that the compiler emits the integer multiply and integer divide instructions which exist in SPARC v8 but not in SPARC v7.

`-msparclite' will give you SPARClite code. This adds the integer multiply, integer divide step and scan (`ffs`) instructions which exist in SPARClite but not in SPARC v7.

`-mcypress`

`-msupersparc`

These two options select the processor for which the code is optimised.

With `-mcypress' (the default), the compiler optimizes code for the Cypress CY7C602 chip, as used in the SparcStation/SparcServer 3xx series. This is also appropriate for the older SparcStation 1, 2, IPX etc.

With `-msupersparc' the compiler optimizes code for the SuperSparc cpu, as used in the SparcStation 10, 1000 and 2000 series. This flag also enables use of the full SPARC v8 instruction set.

In a future version of GCC, these options will very likely be renamed to `-mcpu=cypress' and `-mcpu=supersparc'.

These `-m' switches are supported in addition to the above on SPARC V9 processors:

`-mmedlow`

Generate code for the Medium/Low code model: assume a 32 bit address space. Programs are statically linked, PIC is not supported. Pointers are still 64 bits.

It is very likely that a future version of GCC will rename this option.

`-mmedany`

Generate code for the Medium/Anywhere code model: assume a 32 bit text segment starting at offset 0, and a 32 bit data segment starting anywhere (determined at link time). Programs are statically linked, PIC is not supported. Pointers are still 64 bits.

It is very likely that a future version of GCC will rename this option.

`-mint64`

Types long and int are 64 bits.

`-mlong32`

Types long and int are 32 bits.

`-mlong64`

`-mint32`

Type long is 64 bits, and type int is 32 bits.

`-mstack-bias`

`-mno-stack-bias`

With `-mstack-bias', GNU CC assumes that the stack pointer, and frame pointer if present, are offset by -2047 which must be added back when making stack frame references. Otherwise, assume no such offset is present.

Convex Options

These `-m' options are defined for Convex:

`-mc1`

Generate output for C1. The code will run on any Convex machine. The preprocessor symbol `__convex__c1__` is defined.

`-mc2`

Generate output for C2. Uses instructions not available on C1. Scheduling and other optimizations are chosen for max performance on C2. The preprocessor symbol `__convex_c2__` is defined.

`-mc32`

Generate output for C32xx. Uses instructions not available on C1. Scheduling and other optimizations are chosen for max performance on C32. The preprocessor symbol `__convex_c32__` is defined.

`-mc34`

Generate output for C34xx. Uses instructions not available on C1. Scheduling and other optimizations are chosen for max performance on C34. The preprocessor symbol `__convex_c34__` is defined.

`-mc38`

Generate output for C38xx. Uses instructions not available on C1. Scheduling and other optimizations are chosen for max performance on C38. The preprocessor symbol `__convex_c38__` is defined.

`-margcount`

Generate code which puts an argument count in the word preceding each argument list. This is compatible with regular CC, and a few programs may need the argument count word. GDB and other source-level debuggers do not need it; this info is in the symbol table.

`-mnoargcount`

Omit the argument count word. This is the default.

`-mvolatile-cache`

Allow volatile references to be cached. This is the default.

`-mvolatile-nocache`

Volatile references bypass the data cache, going all the way to memory. This is only needed for multi-processor code that does not use standard synchronization instructions. Making non-volatile references to volatile locations will not necessarily work.

`-mlong32`

Type long is 32 bits, the same as type int. This is the default.

`-mlong64`

Type long is 64 bits, the same as type long long. This option is useless, because no library support exists for it.

AMD29K Options

These `-m' options are defined for the AMD Am29000:

`-mdw`

Generate code that assumes the `DW` bit is set, i.e., that byte and halfword operations are directly supported by the hardware. This is the default.

`-mndw`

Generate code that assumes the `DW` bit is not set.

`-mbw`

Generate code that assumes the system supports byte and halfword write operations. This is the default.

`-mnbw`

Generate code that assumes the systems does not support byte and halfword write operations. `-mnbw' implies `-mndw'.

`-msmall`

Use a small memory model that assumes that all function addresses are either within a single 256 KB segment or at an absolute address of less than 256k. This allows the `call` instruction to be used instead of a `const`, `consth`, `calli` sequence.

`-mnormal`

Use the normal memory model: Generate `call` instructions only when calling functions in the same file and `calli` instructions otherwise. This works if each file occupies less than 256 KB but allows the entire executable to be larger than 256 KB. This is the default.

`-mlarge`

Always use `calli` instructions. Specify this option if you expect a single file to compile into more than 256 KB of code.

`-m29050`

Generate code for the Am29050.

`-m29000`

Generate code for the Am29000. This is the default.

`-mkernel-registers`

Generate references to registers `gr64-gr95` instead of to registers `gr96-gr127`. This option can be used when compiling kernel code that wants a set of global registers disjoint from that used by user-mode code.

Note that when this option is used, register names in `-f' flags must use the normal, user-mode, names.

`-muser-registers`

Use the normal set of global registers, `gr96-gr127`. This is the default.

`-mstack-check`

`-mno-stack-check`

Insert (or do not insert) a call to `__msp_check` after each stack adjustment. This is often used for kernel code.

`-mstorem-bug`

`-mno-storem-bug`

`-mstorem-bug' handles 29k processors which cannot handle the separation of a mtsrim insn and a storem instruction (most 29000 chips to date, but not the 29050).

`-mno-reuse-arg-regs`

`-mreuse-arg-regs`

`-mno-reuse-arg-regs' tells the compiler to only use incoming argument registers for copying out arguments. This helps detect calling a function with fewer arguments than it was declared with.

`-msoft-float`

Generate output containing library calls for floating point. **Warning:** the requisite libraries are not part of GNU CC. Normally the facilities of the machine's usual C compiler are used, but this can't be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation.

ARM Options

These `-m' options are defined for Advanced RISC Machines (ARM) architectures:

`-m2`

`-m3`

These options are identical. Generate code for the ARM2 and ARM3 processors. This option is the default. You should also use this option to generate code for ARM6 processors that are running with a 26-bit program counter.

`-m6`

Generate code for the ARM6 processor when running with a 32-bit program counter.

`-mapcs`

Generate a stack frame that is compliant with the ARM Procedure Call Standard for all functions, even if this is not strictly necessary for correct execution of the code.

`-mbsd`

This option only applies to RISC iX. Emulate the native BSD-mode compiler. This is the default if `-ansi' is not specified.

`-mxopen`

This option only applies to RISC iX. Emulate the native X/Open-mode compiler.

`-mno-symrename`

This option only applies to RISC iX. Do not run the assembler post-processor, `symrename', after code has been assembled. Normally it is necessary to modify some of the standard symbols in preparation for linking with the RISC iX C library; this option suppresses this pass. The post-processor is never run when the compiler is built for cross-compilation.

M88K Options

These `-m' options are defined for Motorola 88k architectures:

`-m88000`

Generate code that works well on both the m88100 and the m88110.

`-m88100`

Generate code that works best for the m88100, but that also runs on the m88110.

`-m88110`

Generate code that works best for the m88110, and may not run on the m88100.

`-mbig-pic`

Obsolete option to be removed from the next revision. Use `-fPIC'.

`-midentify-revision`

Include an `ident` directive in the assembler output recording the source file name, compiler name and version, timestamp, and compilation flags used.

`-mno-underscores`

In assembler output, emit symbol names without adding an underscore character at the beginning of each name. The default is to use an underscore as prefix on each name.

`-mocs-debug-info`

`-mno-ocs-debug-info`

Include (or omit) additional debugging information (about registers used in each stack frame) as specified in the 88open Object Compatibility Standard, "OCS". This extra information allows debugging of code that has had the frame pointer eliminated. The default for DG/UX, SVr4, and Delta 88 SVr3.2 is to include this information; other 88k configurations omit this information by default.

`-mocs-frame-position`

When emitting COFF debugging information for automatic variables and parameters stored on the stack, use the offset from the canonical frame address, which is the stack pointer (register 31) on entry to the function. The DG/UX, SVr4, Delta88 SVr3.2, and BCS configurations use `-mocs-frame-position'; other 88k configurations have the default `-mno-ocs-frame-position'.

`-mno-ocs-frame-position`

When emitting COFF debugging information for automatic variables and parameters stored on the stack, use the offset from the frame pointer register (register 30). When this option is in effect, the frame pointer is not eliminated when debugging information is selected by the -g switch.

`-moptimize-arg-area`

`-mno-optimize-arg-area`

Control how function arguments are stored in stack frames. `-moptimize-arg-area' saves space by optimizing them, but this conflicts with the 88open specifications. The opposite alternative, `-mno-optimize-arg-area', agrees with 88open standards. By default GNU CC does not optimize the argument area.

`-mshort-data-num`

Generate smaller data references by making them relative to `r0`, which allows loading a value using a single instruction (rather than the usual two). You control which data references are affected by specifying num with this option. For example, if you specify `-mshort-data-512', then the data references affected are those involving displacements of less than 512 bytes. `-mshort-data-num' is not effective for num greater than 64k.

`-mserialize-volatile`

`-mno-serialize-volatile`

Do, or don't, generate code to guarantee sequential consistency of volatile memory references. By default, consistency is guaranteed.

The order of memory references made by the MC88110 processor does not always match the order of the instructions requesting those references. In particular, a load instruction may execute before a preceding store instruction. Such reordering violates sequential consistency of volatile memory references, when there are multiple processors. When consistency must be guaranteed, GNU C generates special instructions, as needed, to force execution in the proper order.

The MC88100 processor does not reorder memory references and so always provides sequential consistency. However, by default, GNU C generates the special instructions to guarantee consistency even when you use `-m88100', so that the code may be run on an MC88110 processor. If you intend to run your code only on the MC88100 processor, you may use `-mno-serialize-volatile'.

The extra code generated to guarantee consistency may affect the performance of your application. If you know that you can safely forgo this guarantee, you may use `-mno-serialize-volatile'.

`-msvr4`

`-msvr3`

Turn on (`-msvr4') or off (`-msvr3') compiler extensions related to System V release 4 (SVr4). This controls the following:

Which variant of the assembler syntax to emit.

`-msvr4' makes the C preprocessor recognize `#pragma weak' that is used on System V release 4.

`-msvr4' makes GNU CC issue additional declaration directives used in SVr4.

`-msvr4' is the default for the m88k-motorola-sysv4 and m88k-dg-dgux m88k configurations. `-msvr3' is the default for all other m88k configurations.

`-mversion-03.00`

This option is obsolete, and is ignored.

`-mno-check-zero-division`

`-mcheck-zero-division`

Do, or don't, generate code to guarantee that integer division by zero will be detected. By default detection is guaranteed.

Some models of the MC88100 processor fail to trap upon integer division by zero under certain conditions. By default, when compiling code that might be run on such a processor, GNU C generates code that explicitly checks for zero-valued divisors and traps with exception number 503 when one is detected. Use of mno-check-zero-division suppresses such checking for code generated to run on an MC88100 processor.

GNU C assumes that the MC88110 processor correctly detects all instances of integer division by zero. When `-m88110' is specified, both `-mcheck-zero-division' and `-mno-check-zero-division' are ignored, and no explicit checks for zero-valued divisors are generated.

`-muse-div-instruction`

Use the div instruction for signed integer division on the MC88100 processor. By default, the div instruction is not used.

On the MC88100 processor the signed integer division instruction div) traps to the operating system on a negative operand. The operating system transparently completes the operation, but at a large cost in execution time. By default, when compiling code that might be run on an MC88100 processor, GNU C emulates signed integer division using the unsigned integer division instruction divu), thereby avoiding the large penalty of a trap to the operating system. Such emulation has its own, smaller, execution cost in both time and space. To the extent that your code's important signed integer division operations are performed on two nonnegative operands, it may be desirable to use the div instruction directly.

On the MC88110 processor the div instruction (also known as the divs instruction) processes negative operands without trapping to the operating system. When `-m88110' is specified, `-muse-div-instruction' is ignored, and the div instruction is used for signed integer division.

Note that the result of dividing INT_MIN by -1 is undefined. In particular, the behavior of such a division with and without `-muse-div-instruction' may differ.

`-mtrap-large-shift`

`-mhandle-large-shift`

Include code to detect bit-shifts of more than 31 bits; respectively, trap such shifts or emit code to handle them properly. By default GNU CC makes no special provision for large bit shifts.

`-mwarn-passed-structs`

Warn when a function passes a struct as an argument or result. Structure-passing conventions have changed during the evolution of the C language, and are often the source of portability problems. By default, GNU CC issues no such warning.

IBM RS/6000 and PowerPC Options

These `-m' options are defined for the IBM RS/6000 and PowerPC:

`-mpower`

`-mno-power`
`-mpower2`
`-mno-power2`
`-mpowerpc`
`-mno-powerpc`
`-mpowerpc-gpopt`
`-mno-powerpc-gpopt`
`-mpowerpc-gfxopt`
`-mno-powerpc-gfxopt`

GNU CC supports two related instruction set architectures for the RS/6000 and PowerPC. The POWER instruction set are those instructions supported by the `rios' chip set used in the original RS/6000 systems and the PowerPC instruction set is the architecture of the Motorola MPC6xx microprocessors. The PowerPC architecture defines 64-bit instructions, but they are not supported by any current processors.

Neither architecture is a subset of the other. However there is a large common subset of instructions supported by both. An MQ register is included in processors supporting the POWER architecture.

You use these options to specify which instructions are available on the processor you are using. The default value of these options is determined when configuring GNU CC. Specifying the `-mcpu=cpu_type' overrides the specification of these options. We recommend you use that option rather than these.

The `-mpower' option allows GNU CC to generate instructions that are found only in the POWER architecture and to use the MQ register. Specifying `-mpower2' implies `-power' and also allows GNU CC to generate instructions that are present in the POWER2 architecture but not the original POWER architecture.

The `-mpowerpc' option allows GNU CC to generate instructions that are found only in the 32-bit subset of the PowerPC architecture. Specifying `-mpowerpc-gpopt' implies `-mpowerpc' and also allows GNU CC to use the optional PowerPC architecture instructions in the General Purpose group, including floating-point square root. Specifying `-mpowerpc-gfxopt' implies `-mpowerpc' and also allows GNU CC to use the optional PowerPC architecture instructions in the Graphics group, including floating-point select.

If you specify both `-mno-power' and `-mno-powerpc', GNU CC will use only the instructions in the common subset of both architectures plus some special AIX common-mode calls, and will not use the MQ register. Specifying both `-mpower' and `-mpowerpc' permits GNU CC to use any instruction from either architecture and to allow use of the MQ register; specify this for the Motorola MPC601.

`-mnew-mnemonics`
`-mold-mnemonics`

Select which mnemonics to use in the generated assembler code. `-mnew-mnemonics' requests output that uses the assembler mnemonics defined for the PowerPC architecture, while `-mold-mnemonics' requests the assembler mnemonics defined for the POWER architecture.

Instructions defined in only one architecture have only one mnemonic; GNU CC uses that mnemonic irrespective of which of thse options is specified.

PowerPC assemblers support both the old and new mnemonics, as will later POWER assemblers. Current POWER assemblers only support the old mnemonics. Specify `-mnew-mnemonics' if you have an assembler that supports them, otherwise specify `-mold-mnemonics'.

The default value of these options depends on how GNU CC was configured. Specifying `-mcpu=cpu_type' sometimes overrides the value of these option. Unless you are building a cross-compiler, you should normally not specify either `-mnew-mnemonics' or `-mold-mnemonics', but should instead accept the default.

`-mcpu=cpu_type`

Set architecture type, register usage, choice of mnemonics, and instruction scheduling parameters for machine type cpu_type. By default, cpu_type is the target system defined when GNU CC was configured. Supported values for cpu_type are `rios1', `rios2', `rsc', `601', `603', `604', `power', `powerpc', `403', and `common'. `-mcpu=power' and `-mcpu=powerpc' specify generic POWER and pure PowerPC (i.e., not MPC601) architecture machine types, with an appropriate, generic processor model assumed for scheduling purposes.

Specifying `-mcpu=rios1', `-mcpu=rios2', `-mcpu=rsc', or `-mcpu=power' enables the `-mpower' option and disables the `-mpowerpc' option; `-mcpu=601' enables both the `-mpower' and `-mpowerpc' options; `-mcpu=603', `-mcpu=604', `-mcpu=403', and `-mcpu=powerpc' enable the `-mpowerpc' option and disable the `-mpower' option; `-mcpu=common' disables both the `-mpower' and `-mpowerpc' options.

To generate code that will operate on all members of the RS/6000 and PowerPC families, specify `-mcpu=common'. In that case, GNU CC will use only the instructions in the common subset of both architectures plus some special AIX common-mode calls, and will not use the MQ register. GNU CC assumes a generic processor model for scheduling purposes.

Specifying `-mcpu=rios1', `-mcpu=rios2', `-mcpu=rsc', or `-mcpu=power' also disables the `new-mnemonics' option. Specifying `-mcpu=601', `-mcpu=603', `-mcpu=604', `403', or `-mcpu=powerpc' also enables the `new-mnemonics' option.

`-mfull-toc`

`-mno-fp-in-toc`

`-mno-sum-in-toc`

`-mminimal-toc`

Modify generation of the TOC (Table Of Contents), which is created for every executable file. The `-mfull-toc' option is selected by default. In that case, GNU CC will allocate at least one TOC entry for each unique non-automatic variable reference in your program. GNU CC will also place floating-point constants in the TOC. However, only 16,384 entries are available in the TOC.

If you receive a linker error message that saying you have overflowed the available TOC space, you can reduce the amount of TOC space used with the `-mno-fp-in-toc' and `-mno-sum-in-toc' options. `-mno-fp-in-toc' prevents GNU CC from putting floating-point constants in the TOC and `-mno-sum-in-toc' forces GNU CC to generate code to calculate the sum of an address and a constant at run-time instead of putting that sum into the TOC. You may specify one or both of these

options. Each causes GNU CC to produce very slightly slower and larger code at the expense of conserving TOC space.

If you still run out of space in the TOC even when you specify both of these options, specify `-mminimal-toc' instead. This option causes GNU CC to make only one TOC entry for every file. When you specify this option, GNU CC will produce code that is slower and larger but which uses extremely little TOC space. You may wish to use this option only on files that contain less frequently executed code.

`-msoft-float`

`-mhard-float`

Generate code that does not use (uses) the floating-point register set. Software floating point emulation is provided if you use the `-msoft-float' option, and pass the option to GNU CC when linking.

`-mmultiple`

`-mno-multiple`

Generate code that uses (does not use) the load multiple word instructions and the store multiple word instructions. These instructions are generated by default on POWER systems, and not generated on PowerPC systems. Do not use `-mmultiple' on little endian PowerPC systems, since those instructions do not work when the processor is in little endian mode.

`-mstring`

`-mno-string`

Generate code that uses (does not use) the load string instructions and the store string word instructions to save multiple registers and do small block moves. These instructions are generated by default on POWER systems, anod not generated on PowerPC systems. Do not use `-mstring' on little endian PowerPC systems, since those instructions do not work when the processor is in little endian mode.

`-mno-bit-align`

`-mbit-align`

On System V.4 and embedded PowerPC systems do not (do) force structures and unions that contain bit fields to be aligned to the base type of the bit field.

For example, by default a structure containing nothing but 8 `unsigned` bitfields of length 1 would be aligned to a 4 byte boundary and have a size of 4 bytes. By using `-mno-bit-align', the structure would be aligned to a 1 byte boundary and be one byte in size.

`-mno-strict-align`

`-mstrict-align`

On System V.4 and embedded PowerPC systems do not (do) assume that unaligned memory references will be handled by the system.

`-mrelocatable`

`-mno-relocatable`

On embedded PowerPC systems generate code that allows (does not allow) the program to be relocated to a different address at runtime.

`-mno-toc`
`-mtoc`

On System V.4 and embedded PowerPC systems do not (do) assume that register 2 contains a pointer to a global area pointing to the addresses used in the program.

`-mno-traceback`
`-mtraceback`

On embedded PowerPC systems do not (do) generate a traceback tag before the start of the function. This tag can be used by the debugger to identify where the start of a function is.

`-mlittle`
`-mlittle-endian`

On System V.4 and embedded PowerPC systems compile code for the processor in little endian mode. The `-mlittle-endian' option is the same as `-mlittle'.

`-mbig`
`-mbig-endian`

On System V.4 and embedded PowerPC systems compile code for the processor in big endian mode. The `-mbig-endian' option is the same as `-mbig'.

IBM RT Options

These `-m' options are defined for the IBM RT PC:

`-min-line-mul`

Use an in-line code sequence for integer multiplies. This is the default.

`-mcall-lib-mul`

Call `lmul$$` for integer multiples.

`-mfull-fp-blocks`

Generate full-size floating point data blocks, including the minimum amount of scratch space recommended by IBM. This is the default.

`-mminimum-fp-blocks`

Do not include extra scratch space in floating point data blocks. This results in smaller code, but slower execution, since scratch space must be allocated dynamically.

`-mfp-arg-in-fpregs`

Use a calling sequence incompatible with the IBM calling convention in which floating point arguments are passed in floating point registers. Note that `varargs.h` and `stdargs.h` will not work with floating point operands if this option is specified.

`-mfp-arg-in-gregs`

Use the normal calling convention for floating point arguments. This is the default.

`-mhc-struct-return`

Return structures of more than one word in memory, rather than in a register. This provides compatibility with the MetaWare HighC (hc) compiler. Use the option `-fpcc-struct-return' for compatibility with the Portable C Compiler (pcc).

`-mnohc-struct-return`

Return some structures of more than one word in registers, when convenient. This is the default. For compatibility with the IBM-supplied compilers, use the option `-fpcc-struct-return' or the option `-mhc-struct-return'.

MIPS Options

These `-m' options are defined for the MIPS family of computers:

`-mcpu=cpu type`

Assume the defaults for the machine type cpu type when scheduling instructions. The choices for cpu type are `r2000', `r3000', `r4000', `r4400', `r4600', and `r6000'. While picking a specific cpu type will schedule things appropriately for that particular chip, the compiler will not generate any code that does not meet level 1 of the MIPS ISA (instruction set architecture) without the `-mips2' or `-mips3' switches being used.

`-mips1`

Issue instructions from level 1 of the MIPS ISA. This is the default. `r3000' is the default cpu type at this ISA level.

`-mips2`

Issue instructions from level 2 of the MIPS ISA (branch likely, square root instructions). `r6000' is the default cpu type at this ISA level.

`-mips3`

Issue instructions from level 3 of the MIPS ISA (64 bit instructions). `r4000' is the default cpu type at this ISA level. This option does not change the sizes of any of the C data types.

`-mfp32`

Assume that 32 32-bit floating point registers are available. This is the default.

`-mfp64`

Assume that 32 64-bit floating point registers are available. This is the default when the `-mips3' option is used.

`-mgp32`

Assume that 32 32-bit general purpose registers are available. This is the default.

`-mgp64`

Assume that 32 64-bit general purpose registers are available. This is the default when the `-mips3' option is used.

`-mint64`

Types long, int, and pointer are 64 bits. This works only if `-mips3' is also specified.

`-mlong64`

Types long and pointer are 64 bits, and type int is 32 bits. This works only if `-mips3' is also specified.

`-mmips-as`

Generate code for the MIPS assembler, and invoke `mips-tfile' to add normal debug

information. This is the default for all platforms except for the OSF/1 reference platform, using OSF/rose object format. If the either of the `-gstabs' or `-gstabs+' switches are used, the `mips-tfile' program will encapsulate the stabs within MIPS ECOFF.

`-mgas`

Generate code for the GNU assembler. This is the default on the OSF/1 reference platform, using the OSF/rose object format.

`-mrnames`

`-mno-rnames`

The `-mrnames' switch says to output code using the MIPS software names for the registers, instead of the hardware names (ie, a0 instead of $4). The only known assembler that supports this option is the Algorithmics assembler.

`-mgpopt`

`-mno-gpopt`

The `-mgpopt' switch says to write all of the data declarations before the instructions in the text section, this allows the MIPS assembler to generate one word memory references instead of using two words for short global or static data items. This is on by default if optimization is selected.

`-mstats`

`-mno-stats`

For each non-inline function processed, the `-mstats' switch causes the compiler to emit one line to the standard error file to print statistics about the program (number of registers saved, stack size, etc.).

`-mmemcpy`

`-mno-memcpy`

The `-mmemcpy' switch makes all block moves call the appropriate string function (`memcpy' or `bcopy') instead of possibly generating inline code.

`-mmips-tfile`

`-mno-mips-tfile`

The `-mno-mips-tfile' switch causes the compiler not postprocess the object file with the `mips-tfile' program, after the MIPS assembler has generated it to add debug support. If `mips-tfile' is not run, then no local variables will be available to the debugger. In addition, `stage2' and `stage3' objects will have the temporary file names passed to the assembler embedded in the object file, which means the objects will not compare the same. The `-mno-mips-tfile' switch should only be used when there are bugs in the `mips-tfile' program that prevents compilation.

`-msoft-float`

Generate output containing library calls for floating point. **Warning:** the requisite libraries are not part of GNU CC. Normally the facilities of the machine's usual C compiler are used, but this can't be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation.

`-mhard-float`

Generate output containing floating point instructions. This is the default if you use the unmodified

sources.

`-mabicalls`

`-mno-abicalls`

Emit (or do not emit) the pseudo operations `.abicalls', `.cpload', and `.cprestore' that some System V.4 ports use for position independent code.

`-mlong-calls`

`-mno-long-calls`

Do all calls with the `JALR' instruction, which requires loading up a function's address into a register before the call. You need to use this switch, if you call outside of the current 512 megabyte segment to functions that are not through pointers.

`-mhalf-pic`

`-mno-half-pic`

Put pointers to extern references into the data section and load them up, rather than put the references in the text section.

`-membedded-pic`

`-mno-embedded-pic`

Generate PIC code suitable for some embedded systems. All calls are made using PC relative address, and all data is addressed using the $gp register. This requires GNU as and GNU ld which do most of the work.

`-membedded-data`

`-mno-embedded-data`

Allocate variables to the read-only data section first if possible, then next in the small data section if possible, otherwise in data. This gives slightly slower code than the default, but reduces the amount of RAM required when executing, and thus may be preferred for some embedded systems.

`-msingle-float`

`-mdouble-float`

The `-msingle-float' switch tells gcc to assume that the floating point coprocessor only supports single precision operations, as on the `r4650' chip. The `-mdouble-float' switch permits gcc to use double precision operations. This is the default.

`-mmad`

`-mno-mad`

Permit use of the `mad', `madu' and `mul' instructions, as on the `r4650' chip.

`-m4650`

Turns on `-msingle-float', `-mmad', and, at least for now, `-mcpu=r4650'.

`-EL`

Compile code for the processor in little endian mode. The requisite libraries are assumed to exist.

`-EB`

Compile code for the processor in big endian mode. The requisite libraries are assumed to exist.

`-G num`

Put global and static items less than or equal to num bytes into the small data or bss sections instead of the normal data or bss section. This allows the assembler to emit one word memory reference instructions based on the global pointer (gp or $28), instead of the normal two words used. By default, num is 8 when the MIPS assembler is used, and 0 when the GNU assembler is used. The `-G num' switch is also passed to the assembler and linker. All modules should be compiled with the same `-G num' value.

`-nocpp`

Tell the MIPS assembler to not run it's preprocessor over user assembler files (with a `.s' suffix) when assembling them.

These options are defined by the macro `TARGET_SWITCHES` in the machine description. The default for the options is also defined by that macro, which enables you to change the defaults.

Intel 386 Options

These `-m' options are defined for the i386 family of computers:

`-m486`

`-m386`

Control whether or not code is optimized for a 486 instead of an 386. Code generated for an 486 will run on a 386 and vice versa.

`-mieee-fp`

`-mno-ieee-fp`

Control whether or not the compiler uses IEEE floating point comparisons. These handle correctly the case where the result of a comparison is unordered.

`-msoft-float`

Generate output containing library calls for floating point. **Warning:** the requisite libraries are not part of GNU CC. Normally the facilities of the machine's usual C compiler are used, but this can't be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation.

On machines where a function returns floating point results in the 80387 register stack, some floating point opcodes may be emitted even if `-msoft-float' is used.

`-mno-fp-ret-in-387`

Do not use the FPU registers for return values of functions.

The usual calling convention has functions return values of types `float` and `double` in an FPU register, even if there is no FPU. The idea is that the operating system should emulate an FPU.

The option `-mno-fp-ret-in-387' causes such values to be returned in ordinary CPU registers instead.

`-mno-fancy-math-387`

Some 387 emulators do not support the `sin`, `cos` and `sqrt` instructions for the 387. Specify this option to avoid generating those instructions. This option is the default on FreeBSD. As of revision 2.6.1, these instructions are not generated unless you also use the `-ffast-math' switch.

`-malign-double`

`-mno-align-double`

Control whether GNU CC aligns `double`, `long double`, and `long long` variables on a two word boundary or a one word boundary. Aligning `double` variables on a two word boundary will produce code that runs somewhat faster on a `Pentium' at the expense of more memory.

Warning: if you use the `-malign-double' switch, structures containing the above types will be aligned differently than the published application binary interface specifications for the 386.

`-msvr3-shlib`

`-mno-svr3-shlib`

Control whether GNU CC places uninitialized locals into `bss` or `data`. `-msvr3-shlib' places these locals into `bss`. These options are meaningful only on System V Release 3.

`-mno-wide-multiply`

`-mwide-multiply`

Control whether GNU CC uses the `mul` and `imul` that produce 64 bit results in `eax:edx` from 32 bit operands to do `long long` multiplies and 32-bit division by constants.

`-mrtd`

Use a different function-calling convention, in which functions that take a fixed number of arguments return with the `ret` num instruction, which pops their arguments while returning. This saves one instruction in the caller since there is no need to pop the arguments there.

You can specify that an individual function is called with this calling sequence with the function attribute `stdcall'. You can also override the `-mrtd' option by using the function attribute `cdecl'. See section Declaring Attributes of Functions

Warning: this calling convention is incompatible with the one normally used on Unix, so you cannot use it if you need to call libraries compiled with the Unix compiler.

Also, you must provide function prototypes for all functions that take variable numbers of arguments (including `printf`); otherwise incorrect code will be generated for calls to those functions.

In addition, seriously incorrect code will result if you call a function with too many arguments. (Normally, extra arguments are harmlessly ignored.)

`-mreg-alloc=regs`

Control the default allocation order of integer registers. The string regs is a series of letters specifying a register. The supported letters are: `a` allocate EAX; `b` allocate EBX; `c` allocate ECX; `d` allocate EDX; `S` allocate ESI; `D` allocate EDI; `B` allocate EBP.

`-mregparm=num`

Control how many registers are used to pass integer arguments. By default, no registers are used to pass arguments, and at most 3 registers can be used. You can control this behavior for a specific function by using the function attribute `regparm'. See section Declaring Attributes of Functions

Warning: if you use this switch, and num is nonzero, then you must build all modules with the same value, including any libraries. This includes the system libraries and startup modules.

`-malign-loops=num`

Align loops to a 2 raised to a num byte boundary. If `-malign-loops' is not specified, the default is 2.

`-malign-jumps=num`

Align instructions that are only jumped to to a 2 raised to a num byte boundary. If `-malign-jumps' is not specified, the default is 2 if optimizing for a 386, and 4 if optimizing for a 486.

`-malign-functions=num`

Align the start of functions to a 2 raised to num byte boundary. If `-malign-jumps' is not specified, the default is 2 if optimizing for a 386, and 4 if optimizing for a 486.

HPPA Options

These `-m' options are defined for the HPPA family of computers:

`-mpa-risc-1-0`

Generate code for a PA 1.0 processor.

`-mpa-risc-1-1`

Generate code for a PA 1.1 processor.

`-mjump-in-delay`

Fill delay slots of function calls with unconditional jump instructions by modifying the return pointer for the function call to be the target of the conditional jump.

`-mmillicode-long-calls`

Generate code which assumes millicode routines can not be reached by the standard millicode call sequence, linker-generated long-calls, or linker-modified millicode calls. In practice this should only be needed for dynamicly linked executables with extremely large SHLIB_INFO sections.

`-mdisable-fpregs`

Prevent floating point registers from being used in any manner. This is necessary for compiling kernels which perform lazy context switching of floating point registers. If you use this option and attempt to perform floating point operations, the compiler will abort.

`-mdisable-indexing`

Prevent the compiler from using indexing address modes. This avoids some rather obscure problems when compiling MIG generated code under MACH.

`-mfast-indirect-calls`

Generate code which performs faster indirect calls. Such code is suitable for kernels and for static linking. The fast indirect call code will fail miserably if it's part of a dynamically linked executable and in the presense of nested functions.

`-mportable-runtime`

Use the portable calling conventions proposed by HP for ELF systems.

`-mgas`

Enable the use of assembler directives only GAS understands.

`-mschedule=cpu type`

Schedule code according to the constraints for the machine type cpu type. The choices for cpu type

are `700' for 7n0 machines, `7100' for 7n5 machines, and `7100' for 7n2 machines. `700' is the default for cpu type.

Note the `7100LC' scheduling information is incomplete and using `7100LC' often leads to bad schedules. For now it's probably best to use `7100' instead of `7100LC' for the 7n2 machines.

`-msoft-float`

Generate output containing library calls for floating point. **Warning:** the requisite libraries are not available for all HPPA targets. Normally the facilities of the machine's usual C compiler are used, but this cannot be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation. The embedded target `hppa1.1-*-pro' does provide software floating point support.

`-msoft-float' changes the calling convention in the output file; therefore, it is only useful if you compile *all* of a program with this option. In particular, you need to compile `libgcc.a', the library that comes with GNU CC, with `-msoft-float' in order for this to work.

Intel 960 Options

These `-m' options are defined for the Intel 960 implementations:

`-mcpu type`

Assume the defaults for the machine type cpu type for some of the other options, including instruction scheduling, floating point support, and addressing modes. The choices for cpu type are `ka', `kb', `mc', `ca', `cf', `sa', and `sb'. The default is `kb'.

`-mnumerics`

`-msoft-float`

The `-mnumerics' option indicates that the processor does support floating-point instructions. The `-msoft-float' option indicates that floating-point support should not be assumed.

`-mleaf-procedures`

`-mno-leaf-procedures`

Do (or do not) attempt to alter leaf procedures to be callable with the `bal` instruction as well as `call`. This will result in more efficient code for explicit calls when the `bal` instruction can be substituted by the assembler or linker, but less efficient code in other cases, such as calls via function pointers, or using a linker that doesn't support this optimization.

`-mtail-call`

`-mno-tail-call`

Do (or do not) make additional attempts (beyond those of the machine-independent portions of the compiler) to optimize tail-recursive calls into branches. You may not want to do this because the detection of cases where this is not valid is not totally complete. The default is `-mno-tail-call'.

`-mcomplex-addr`

`-mno-complex-addr`

Assume (or do not assume) that the use of a complex addressing mode is a win on this implementation of the i960. Complex addressing modes may not be worthwhile on the K-series, but they definitely are on the C-series. The default is currently `-mcomplex-addr' for all processors

except the CB and CC.

`-mcode-align`

`-mno-code-align`

Align code to 8-byte boundaries for faster fetching (or don't bother). Currently turned on by default for C-series implementations only.

`-mic-compat`

`-mic2.0-compat`

`-mic3.0-compat`

Enable compatibility with iC960 v2.0 or v3.0.

`-masm-compat`

`-mintel-asm`

Enable compatibility with the iC960 assembler.

`-mstrict-align`

`-mno-strict-align`

Do not permit (do permit) unaligned accesses.

`-mold-align`

Enable structure-alignment compatibility with Intel's gcc release version 1.3 (based on gcc 1.37). Currently this is buggy in that `#pragma align 1' is always assumed as well, and cannot be turned off.

DEC Alpha Options

These `-m' options are defined for the DEC Alpha implementations:

`-mno-soft-float`

`-msoft-float`

Use (do not use) the hardware floating-point instructions for floating-point operations. When `-msoft-float` is specified, functions in `` `libgcc1.c' `` will be used to perform floating-point operations. Unless they are replaced by routines that emulate the floating-point operations, or compiled in such a way as to call such emulations routines, these routines will issue floating-point operations. If you are compiling for an Alpha without floating-point operations, you must ensure that the library is built so as not to call them.

Note that Alpha implementations without floating-point operations are required to have floating-point registers.

`-mfp-reg`

`-mno-fp-regs`

Generate code that uses (does not use) the floating-point register set. `-mno-fp-regs` implies `-msoft-float`. If the floating-point register set is not used, floating point operands are passed in integer registers as if they were integers and floating-point results are passed in $0 instead of $f0. This is a non-standard calling sequence, so any function with a floating-point argument or return value called by code compiled with `-mno-fp-regs` must also be compiled with that option.

A typical use of this option is building a kernel that does not use, and hence need not save and restore, any floating-point registers.

Clipper Options

These `-m' options are defined for the Clipper implementations:

`-mc300`

Produce code for a C300 Clipper processor. This is the default.

`-mc400`

Produce code for a C400 Clipper processor i.e. use floating point registers f8..f15.

H8/300 Options

These `-m' options are defined for the H8/300 implementations:

`-mrelax`

Shorten some address references at link time, when possible; uses the linker option `-relax'. See section '`ld` and the H8/300' in Using ld, for a fuller description.

`-mh`

Generate code for the H8/300H.

Options for System V

These additional options are available on System V Release 4 for compatibility with other compilers on those systems:

`-Qy`

Identify the versions of each tool used by the compiler, in a `.ident` assembler directive in the output.

`-Qn`

Refrain from adding `.ident` directives to the output file (this is the default).

`-YP,dirs`

Search the directories dirs, and no others, for libraries specified with `-l'.

`-Ym,dir`

Look in the directory dir to find the M4 preprocessor. The assembler uses this option.

Options for Code Generation Conventions

These machine-independent options control the interface conventions used in code generation.

Most of them have both positive and negative forms; the negative form of `-ffoo' would be `-fno-foo'. In the table below, only one of the forms is listed--the one which is not the default. You can figure out the other form by either removing `no-' or adding it.

`-fpcc-struct-return`

Return "short" `struct` and `union` values in memory like longer ones, rather than in registers. This convention is less efficient, but it has the advantage of allowing intercallability between GNU CC-compiled files and files compiled with other compilers.

The precise convention for returning structures in memory depends on the target configuration macros.

Short structures and unions are those whose size and alignment match that of some integer type.

`-freg-struct-return`

Use the convention that `struct` and `union` values are returned in registers when possible. This is more efficient for small structures than `-fpcc-struct-return'.

If you specify neither `-fpcc-struct-return' nor its contrary `-freg-struct-return', GNU CC defaults to whichever convention is standard for the target. If there is no standard convention, GNU CC defaults to `-fpcc-struct-return', except on targets where GNU CC is the principal compiler. In those cases, we can choose the standard, and we chose the more efficient register return alternative.

`-fshort-enums`

Allocate to an `enum` type only as many bytes as it needs for the declared range of possible values. Specifically, the `enum` type will be equivalent to the smallest integer type which has enough room.

`-fshort-double`

Use the same size for `double` as for `float`.

`-fshared-data`

Requests that the data and non-`const` variables of this compilation be shared data rather than private data. The distinction makes sense only on certain operating systems, where shared data is shared between processes running the same program, while private data exists in one copy per process.

`-fno-common`

Allocate even uninitialized global variables in the bss section of the object file, rather than generating them as common blocks. This has the effect that if the same variable is declared (without `extern`) in two different compilations, you will get an error when you link them. The only reason this might be useful is if you wish to verify that the program will work on other systems which always work this way.

`-fno-ident`

Ignore the `#ident' directive.

`-fno-gnu-linker`

Do not output global initializations (such as C++ constructors and destructors) in the form used by the GNU linker (on systems where the GNU linker is the standard method of handling them). Use this option when you want to use a non-GNU linker, which also requires using the `collect2` program to make sure the system linker includes constructors and destructors. (`collect2` is included in the GNU CC distribution.) For systems which *must* use `collect2`, the compiler driver `gcc` is configured to do this automatically.

`-finhibit-size-directive`

Don't output a `.size` assembler directive, or anything else that would cause trouble if the func is split in the middle, and the two halves are placed at locations far apart in memory. This optio used when compiling `` `crtstuff.c' ``; you should not need to use it for anything else.

`-fverbose-asm`

Put extra commentary information in the generated assembly code to make it more readable. This option is generally only of use to those who actually need to read the generated assembly code (perhaps while debugging the compiler itself).

`-fvolatile`

Consider all memory references through pointers to be volatile.

`-fvolatile-global`

Consider all memory references to extern and global data items to be volatile.

`-fpic`

Generate position-independent code (PIC) suitable for use in a shared library, if supported for the target machine. Such code accesses all constant addresses through a global offset table (GOT). If the GOT size for the linked executable exceeds a machine-specific maximum size, you get an error message from the linker indicating that `-fpic' does not work; in that case, recompile with `-fPIC' instead. (These maximums are 16k on the m88k, 8k on the Sparc, and 32k on the m68k and RS/6000. The 386 has no such limit.)

Position-independent code requires special support, and therefore works only on certain machines. For the 386, GNU CC supports PIC for System V but not for the Sun 386i. Code generated for the IBM RS/6000 is always position-independent.

The GNU assembler does not fully support PIC. Currently, you must use some other assembler in order for PIC to work. We would welcome volunteers to upgrade GAS to handle this; the first part of the job is to figure out what the assembler must do differently.

`-fPIC`

If supported for the target machine, emit position-independent code, suitable for dynamic linking and avoiding any limit on the size of the global offset table. This option makes a difference on the m68k, m88k and the Sparc.

Position-independent code requires special support, and therefore works only on certain machines.

`-ffixed-reg`

Treat the register named reg as a fixed register; generated code should never refer to it (except perhaps as a stack pointer, frame pointer or in some other fixed role).

reg must be the name of a register. The register names accepted are machine-specific and are defined in the `REGISTER_NAMES` macro in the machine description macro file.

This flag does not have a negative form, because it specifies a three-way choice.

`-fcall-used-reg`

Treat the register named reg as an allocatable register that is clobbered by function calls. It may be allocated for temporaries or variables that do not live across a call. Functions compiled this way will not save and restore the register reg.

Use of this flag for a register that has a fixed pervasive role in the machine's execution model, s as the stack pointer or frame pointer, will produce disastrous results.

This flag does not have a negative form, because it specifies a three-way choice.

`-fcall-saved-reg`

Treat the register named reg as an allocatable register saved by functions. It may be allocated even for temporaries or variables that live across a call. Functions compiled this way will save and restore the register reg if they use it.

Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

A different sort of disaster will result from the use of this flag for a register in which function values may be returned.

This flag does not have a negative form, because it specifies a three-way choice.

`-fpack-struct`

Pack all structure members together without holes. Usually you would not want to use this option, since it makes the code suboptimal, and the offsets of structure members won't agree with system libraries.

`+e0`

`+e1`

Control whether virtual function definitions in classes are used to generate code, or only to define interfaces for their callers. (C++ only).

These options are provided for compatibility with `cfront` 1.x usage; the recommended alternative GNU C++ usage is in flux. See section Declarations and Definitions in One Header.

With `+e0', virtual function definitions in classes are declared `extern`; the declaration is used only as an interface specification, not to generate code for the virtual functions (in this compilation).

With `+e1', G++ actually generates the code implementing virtual functions defined in the code, and makes them publicly visible.

Environment Variables Affecting GNU CC

This section describes several environment variables that affect how GNU CC operates. They work by specifying directories or prefixes to use when searching for various kinds of files.

Note that you can also specify places to search using options such as `-B', `-I' and `-L' (see section Options for Directory Search). These take precedence over places specified using environment variables, which in turn take precedence over those specified by the configuration of GNU CC. See section Controlling the Compilation Driver, `gcc'.

`TMPDIR`

If `TMPDIR` is set, it specifies the directory to use for temporary files. GNU CC uses temporary files

to hold the output of one stage of compilation which is to be used as input to the next stage: for example, the output of the preprocessor, which is the input to the compiler proper.

`GCC_EXEC_PREFIX`

If `GCC_EXEC_PREFIX` is set, it specifies a prefix to use in the names of the subprograms executed by the compiler. No slash is added when this prefix is combined with the name of a subprogram, but you can specify a prefix that ends with a slash if you wish.

If GNU CC cannot find the subprogram using the specified prefix, it tries looking in the usual places for the subprogram.

The default value of `GCC_EXEC_PREFIX` is `` `prefix/lib/gcc-lib/' `` where prefix is the value of `prefix` when you ran the `` `configure' `` script.

Other prefixes specified with `-B' take precedence over this prefix.

This prefix is also used for finding files such as `` `crt0.o' `` that are used for linking.

In addition, the prefix is used in an unusual way in finding the directories to search for header files. For each of the standard directories whose name normally begins with `/usr/local/lib/gcc-lib' (more precisely, with the value of `GCC_INCLUDE_DIR`), GNU CC tries replacing that beginning with the specified prefix to produce an alternate directory name. Thus, with `-Bfoo/', GNU CC will search `` `foo/bar' `` where it would normally search `` `/usr/local/lib/bar' ``. These alternate directories are searched first; the standard directories come next.

`COMPILER_PATH`

The value of `COMPILER_PATH` is a colon-separated list of directories, much like `PATH`. GNU CC tries the directories thus specified when searching for subprograms, if it can't find the subprograms using `GCC_EXEC_PREFIX`.

`LIBRARY_PATH`

The value of `LIBRARY_PATH` is a colon-separated list of directories, much like `PATH`. When configured as a native compiler, GNU CC tries the directories thus specified when searching for special linker files, if it can't find them using `GCC_EXEC_PREFIX`. Linking using GNU CC also uses these directories when searching for ordinary libraries for the `-l' option (but directories specified with `-L' come first).

`C_INCLUDE_PATH`

`CPLUS_INCLUDE_PATH`

`OBJC_INCLUDE_PATH`

These environment variables pertain to particular languages. Each variable's value is a colon-separated list of directories, much like `PATH`. When GNU CC searches for header files, it tries the directories listed in the variable for the language you are using, after the directories specified with `-I' but before the standard header file directories.

`DEPENDENCIES_OUTPUT`

If this variable is set, its value specifies how to output dependencies for Make based on the header files processed by the compiler. This output looks much like the output from the `-M' option (see section Options Controlling the Preprocessor), but it goes to a separate file, and is in addition to the usual results of compilation.

The value of `DEPENDENCIES_OUTPUT` can be just a file name, in which case the Make rules are written to that file, guessing the target name from the source file name. Or the value can have the form `file target', in which case the rules are written to file file using target as the target name.

Running Protoize

The program `protoize` is an optional part of GNU C. You can use it to add prototypes to a program, thus converting the program to ANSI C in one respect. The companion program `unprotoize` does the reverse: it removes argument types from any prototypes that are found.

When you run these programs, you must specify a set of source files as command line arguments. The conversion programs start out by compiling these files to see what functions they define. The information gathered about a file foo is saved in a file named `` `foo.X' ``.

After scanning comes actual conversion. The specified files are all eligible to be converted; any files they include (whether sources or just headers) are eligible as well.

But not all the eligible files are converted. By default, `protoize` and `unprotoize` convert only source and header files in the current directory. You can specify additional directories whose files should be converted with the `-d directory' option. You can also specify particular files to exclude with the `-x file' option. A file is converted if it is eligible, its directory name matches one of the specified directory names, and its name within the directory has not been excluded.

Basic conversion with `protoize` consists of rewriting most function definitions and function declarations to specify the types of the arguments. The only ones not rewritten are those for varargs functions.

`protoize` optionally inserts prototype declarations at the beginning of the source file, to make them available for any calls that precede the function's definition. Or it can insert prototype declarations with block scope in the blocks where undeclared functions are called.

Basic conversion with `unprotoize` consists of rewriting most function declarations to remove any argument types, and rewriting function definitions to the old-style pre-ANSI form.

Both conversion programs print a warning for any function declaration or definition that they can't convert. You can suppress these warnings with `-q'.

The output from `protoize` or `unprotoize` replaces the original source file. The original file is renamed to a name ending with `.save'. If the `.save' file already exists, then the source file is simply discarded.

`protoize` and `unprotoize` both depend on GNU CC itself to scan the program and collect information about the functions it uses. So neither of these programs will work until GNU CC is installed.

Here is a table of the options you can use with `protoize` and `unprotoize`. Each option works with both programs unless otherwise stated.

`-B directory`

Look for the file `` `SYSCALLS.c.X' `` in directory, instead of the usual directory (normally

`/usr/local/lib'). This file contains prototype information about standard system functi
This option applies only to `protoize`.

`-c compilation-options`

Use compilation-options as the options when running `gcc` to produce the `.X' files. The special option `-aux-info' is always passed in addition, to tell `gcc` to write a `.X' file.

Note that the compilation options must be given as a single argument to `protoize` or `unprotoize`. If you want to specify several `gcc` options, you must quote the entire set of compilation options to make them a single word in the shell.

There are certain `gcc` arguments that you cannot use, because they would produce the wrong kind of output. These include `-g', `-O', `-c', `-S', and `-o' If you include these in the compilation-options, they are ignored.

`-C`

Rename files to end in `.C' instead of `.c'. This is convenient if you are converting a C program to C++. This option applies only to `protoize`.

`-g`

Add explicit global declarations. This means inserting explicit declarations at the beginning of each source file for each function that is called in the file and was not declared. These declarations precede the first function definition that contains a call to an undeclared function. This option applies only to `protoize`.

`-i string`

Indent old-style parameter declarations with the string string. This option applies only to `protoize`.

`unprotoize` converts prototyped function definitions to old-style function definitions, where the arguments are declared between the argument list and the initial `{'. By default, `unprotoize` uses five spaces as the indentation. If you want to indent with just one space instead, use `-i " "'.

`-k`

Keep the `.X' files. Normally, they are deleted after conversion is finished.

`-l`

Add explicit local declarations. `protoize` with `-l' inserts a prototype declaration for each function in each block which calls the function without any declaration. This option applies only to `protoize`.

`-n`

Make no real changes. This mode just prints information about the conversions that would have been done without `-n'.

`-N`

Make no `.save' files. The original files are simply deleted. Use this option with caution.

`-p program`

Use the program program as the compiler. Normally, the name `gcc' is used.

`-q`

Work quietly. Most warnings are suppressed.

`-v`

Print the version number, just like `-v' for `gcc`.

If you need special compiler options to compile one of your program's source files, then you should generate that file's `.X' file specially, by running `gcc` on that source file with the appropriate options and the option `-aux-info'. Then run `protoize` on the entire set of files. `protoize` will use the existing `.X' file because it is newer than the source file. For example:

```
gcc -Dfoo=bar file1.c -aux-info
protoize *.c
```

You need to include the special files along with the rest in the `protoize` command, even though their `.X' files already exist, because otherwise they won't get converted.

See section Caveats of using `protoize`, for more information on how to use `protoize` successfully.

Go to the previous, next section.

Go to the previous, next section.

Installing GNU CC

Here is the procedure for installing GNU CC on a Unix system. See section Installing GNU CC on VMS, for VMS systems. In this section we assume you compile in the same directory that contains the source files; see section Compilation in a Separate Directory, to find out how to compile in a separate directory on Unix systems.

You cannot install GNU C by itself on MSDOS; it will not compile under any MSDOS compiler except itself. You need to get the complete compilation package DJGPP, which includes binaries as well as sources, and includes all the necessary compilation tools and libraries.

1. If you have built GNU CC previously in the same directory for a different target machine, do `make distclean' to delete all files that might be invalid. One of the files this deletes is `Makefile'; if `make distclean' complains that `Makefile' does not exist, it probably means that the directory is already suitably clean.
2. On a System V release 4 system, make sure `/usr/bin' precedes `/usr/ucb' in PATH. The cc command in `/usr/ucb' uses libraries which have bugs.
3. Specify the host, build and target machine configurations. You do this by running the file `configure'.

   The build machine is the system which you are using, the host machine is the system where you want to run the resulting compiler (normally the build machine), and the target machine is the system for which you want the compiler to generate code.

   If you are building a compiler to produce code for the machine it runs on (a native compiler), you normally do not need to specify any operands to `configure'; it will try to guess the type of machine you are on and use that as the build, host and target machines. So you don't need to specify a configuration when building a native compiler unless `configure' cannot figure out what your configuration is or guesses wrong.

   In those cases, specify the build machine's configuration name with the `--build' option; the host and target will default to be the same as the build machine. (If you are building a cross-compiler, see section Building and Installing a Cross-Compiler.)

   Here is an example:

   from an MSDOS console window or from the program manager dialog box. Configure.bat assumes that you have already installed and in your path a Unix-like sed program which is used to modify Makefile.in to create a working Makefile.@smallexample ./configure --build=sparc-sun-sunos4.1

   A configuration name may be canonical or it may be more or less abbreviated.

   A canonical configuration name has three parts, separated by dashes. It looks like this: `cpu-company-system'. (The three parts may themselves contain dashes; `configure' can figure out which dashes serve which purpose.) For example, `m68k-sun-sunos4.1' specifies a Sun 3.

   You can also replace parts of the configuration by nicknames or aliases. For example, `sun3' stands for `m68k-sun', so `sun3-sunos4.1' is another way to specify a Sun 3. You can also use simply `sun3-sunos', since the version of SunOS is assumed by default to be version 4. `sun3-bsd' also works, since

`configure' knows that the only BSD variant on a Sun 3 is SunOS.

You can specify a version number after any of the system types, and some of the CPU types. In most cases, the version is irrelevant, and will be ignored. So you might as well specify the version if you know it.

See section Configurations Supported by GNU CC, for a list of supported configuration names and notes on many of the configurations. You should check the notes in that section before proceeding any further with the installation of GNU CC.

There are four additional options you can specify independently to describe variant hardware and software configurations. These are `--with-gnu-as', `--with-gnu-ld', `--with-stabs' and `--nfp'.

`--with-gnu-as'

If you will use GNU CC with the GNU assembler (GAS), you should declare this by using the `--with-gnu-as' option when you run `configure'.

Using this option does not install GAS. It only modifies the output of GNU CC to work with GAS. Building and installing GAS is up to you.

Conversely, if you *do not* wish to use GAS and do not specify `--with-gnu-as' when building GNU CC, it is up to you to make sure that GAS is not installed. GNU CC searches for a program named `as` in various directories; if the program it finds is GAS, then it runs GAS. If you are not sure where GNU CC finds the assembler it is using, try specifying `-v' when you run it.

The systems where it makes a difference whether you use GAS are `hppa1.0-any-any', `hppa1.1-any-any', `i386-any-sysv', `i386-any-isc', `i860-any-bsd', `m68k-bull-sysv', `m68k-hp-hpux', `m68k-sony-bsd', `m68k-altos-sysv', `m68000-hp-hpux', `m68000-att-sysv', `any-lynx-lynxos', and `mips-any'). On any other system, `--with-gnu-as' has no effect.

On the systems listed above (except for the HP-PA, for ISC on the 386, and for `mips-sgi-irix5.*'), if you use GAS, you should also use the GNU linker (and specify `--with-gnu-ld').

`--with-gnu-ld'

Specify the option `--with-gnu-ld' if you plan to use the GNU linker with GNU CC.

This option does not cause the GNU linker to be installed; it just modifies the behavior of GNU CC to work with the GNU linker. Specifically, it inhibits the installation of `collect2`, a program which otherwise serves as a front-end for the system's linker on most configurations.

`--with-stabs'

On MIPS based systems and on Alphas, you must specify whether you want GNU CC to create the normal ECOFF debugging format, or to use BSD-style stabs passed through the ECOFF symbol table. The normal ECOFF debug format cannot fully handle languages other than C. BSD stabs format can handle other languages, but it only works with the GNU debugger GDB.

Normally, GNU CC uses the ECOFF debugging format by default; if you prefer BSD stabs, specify `--with-stabs' when you configure GNU CC.

No matter which default you choose when you configure GNU CC, the user can use the `-gcoff' and `-gstabs+' options to specify explicitly the debug format for a particular compilation.

`--with-stabs' is meaningful on the ISC system on the 386, also, if `--with-gas' is used. It selects use

of stabs debugging information embedded in COFF output. This kind of debugging information supports C++ well; ordinary COFF debugging information does not.

`--with-stabs' is also meaningful on 386 systems running SVR4. It selects use of stabs debugging information embedded in ELF output. The C++ compiler currently (2.6.0) does not support the DWARF debugging information normally used on 386 SVR4 platforms; stabs provide a workable alternative. This requires gas and gdb, as the normal SVR4 tools can not generate or interpret stabs.

`--nfp'

On certain systems, you must specify whether the machine has a floating point unit. These systems include `m68k-sun-sunosn' and `m68k-isi-bsd'. On any other system, `--nfp' currently has no effect, though perhaps there are other systems where it could usefully make a difference.

The `configure' script searches subdirectories of the source directory for other compilers that are to be integrated into GNU CC. The GNU compiler for C++, called G++ is in a subdirectory named `cp'. `configure' inserts rules into `Makefile' to build all of those compilers.

Here we spell out what files will be set up by `configure`. Normally you need not be concerned with these files.

- A symbolic link named `config.h' is made to the top-level config file for the machine you will run the compiler on (see section The Configuration File). This file is responsible for defining information about the host machine. It includes `tm.h'.

  The top-level config file is located in the subdirectory `config'. Its name is always `xm-something.h'; usually `xm-machine.h', but there are some exceptions.

  If your system does not support symbolic links, you might want to set up `config.h' to contain a `#include' command which refers to the appropriate file.
- A symbolic link named `tconfig.h' is made to the top-level config file for your target machine. This is used for compiling certain programs to run on that machine.
- A symbolic link named `tm.h' is made to the machine-description macro file for your target machine. It should be in the subdirectory `config' and its name is often `machine.h'.
- A symbolic link named `md' will be made to the machine description pattern file. It should be in the `config' subdirectory and its name should be `machine.md'; but machine is often not the same as the name used in the `tm.h' file because the `md' files are more general.
- A symbolic link named `aux-output.c' will be made to the output subroutine file for your machine. It should be in the `config' subdirectory and its name should be `machine.c'.
- The command file `configure' also constructs the file `Makefile' by adding some text to the template file `Makefile.in'. The additional text comes from files in the `config' directory, named `t-target' and `x-host'. If these files do not exist, it means nothing needs to be added for a given target or host.

4. The standard directory for installing GNU CC is `/usr/local/lib'. If you want to install its files somewhere else, specify `--prefix=dir' when you run `configure'. Here dir is a directory name to use instead of `/usr/local' for all purposes with one exception: the directory `/usr/local/include' is searched for header files no matter where you install the compiler. To override this name, use the `--local-prefix` option below.
5. Specify `--local-prefix=dir' if you want the compiler to search directory `dir/include' for locally installed header files *instead* of `/usr/local/include'.

You should specify `--local-prefix' **only** if your site has a different convention (not `/usr/local' where to put site-specific files.

Do not specify `/usr' as the `--local-prefix'! The directory you use for `--local-prefix' **must not** contain any of the system's standard header files. If it did contain them, certain programs would be miscompiled (including GNU Emacs, on certain targets), because this would override and nullify the header file corrections made by the `fixincludes` script.

6. Make sure the Bison parser generator is installed. (This is unnecessary if the Bison output files `c-parse.c' and `cexp.c' are more recent than `c-parse.y' and `cexp.y' and you do not plan to change the `.y' files.)

   Bison versions older than Sept 8, 1988 will produce incorrect output for `c-parse.c'.

7. If you have chosen a configuration for GNU CC which requires other GNU tools (such as GAS or the GNU linker) instead of the standard system tools, install the required tools in the build directory under the names `as', `ld' or whatever is appropriate. This will enable the compiler to find the proper tools for compilation of the program `enquire'.

   Alternatively, you can do subsequent compilation using a value of the `PATH` environment variable such that the necessary GNU tools come before the standard system tools.

8. Build the compiler. Just type `make LANGUAGES=c' in the compiler directory.

   `LANGUAGES=c' specifies that only the C compiler should be compiled. The makefile normally builds compilers for all the supported languages; currently, C, C++ and Objective C. However, C is the only language that is sure to work when you build with other non-GNU C compilers. In addition, building anything but C at this stage is a waste of time.

   In general, you can specify the languages to build by typing the argument `LANGUAGES="list"', where list is one or more words from the list `c', `c++', and `objective-c'. If you have any additional GNU compilers as subdirectories of the GNU CC source directory, you may also specify their names in this list.

   Ignore any warnings you may see about "statement not reached" in `insn-emit.c'; they are normal. Also, warnings about "unknown escape sequence" are normal in `genopinit.c' and perhaps some other files. Likewise, you should ignore warnings about "constant is so large that it is unsigned" in `insn-emit.c' and `insn-recog.c'. Any other compilation errors may represent bugs in the port to your machine or operating system, and should be investigated and reported (see section Reporting Bugs).

   Some commercial compilers fail to compile GNU CC because they have bugs or limitations. For example, the Microsoft compiler is said to run out of macro space. Some Ultrix compilers run out of expression space; then you need to break up the statement where the problem happens.

9. If you are building a cross-compiler, stop here. See section Building and Installing a Cross-Compiler.

10. Move the first-stage object files and executables into a subdirectory with this command:

    ```
    make stage1
    ```

    The files are moved into a subdirectory named `stage1'. Once installation is complete, you may wish to delete these files with `rm -r stage1`.

11. If you have chosen a configuration for GNU CC which requires other GNU tools (such as GAS or the GNU linker) instead of the standard system tools, install the required tools in the `stage1' subdirectory

under the names `as', `ld' or whatever is appropriate. This will enable the stage 1 compiler to find the proper tools in the following stage.

Alternatively, you can do subsequent compilation using a value of the PATH environment variable such that the necessary GNU tools come before the standard system tools.

12. Recompile the compiler with itself, with this command:

```
make CC="stage1/xgcc -Bstage1/" CFLAGS="-g -O"
```

This is called making the stage 2 compiler.

The command shown above builds compilers for all the supported languages. If you don't want them all, you can specify the languages to build by typing the argument `LANGUAGES="list"'. list should contain one or more words from the list `c', `c++', `objective-c', and `proto'. Separate the words with spaces. `proto' stands for the programs protoize and unprotoize; they are not a separate language, but you use LANGUAGES to enable or disable their installation.

If you are going to build the stage 3 compiler, then you might want to build only the C language in stage 2.

Once you have built the stage 2 compiler, if you are short of disk space, you can delete the subdirectory `stage1'.

On a 68000 or 68020 system lacking floating point hardware, unless you have selected a `tm.h' file that expects by default that there is no such hardware, do this instead:

```
make CC="stage1/xgcc -Bstage1/" CFLAGS="-g -O -msoft-float"
```

13. If you wish to test the compiler by compiling it with itself one more time, install any other necessary GNU tools (such as GAS or the GNU linker) in the `stage2' subdirectory as you did in the `stage1' subdirectory, then do this:

```
make stage2
make CC="stage2/xgcc -Bstage2/" CFLAGS="-g -O2"
```

This is called making the stage 3 compiler. Aside from the `-B' option, the compiler options should be the same as when you made the stage 2 compiler. But the LANGUAGES option need not be the same. The command shown above builds compilers for all the supported languages; if you don't want them all, you can specify the languages to build by typing the argument `LANGUAGES="list"', as described above.

If you do not have to install any additional GNU tools, you may use the command

```
make bootstrap LANGUAGES=language-list BOOT_CFLAGS=option-list
```

instead of making `stage1', `stage2', and performing the two compiler builds.

14. Then compare the latest object files with the stage 2 object files--they ought to be identical, aside from time stamps (if any).

On some systems, meaningful comparison of object files is impossible; they always appear "different." This is currently true on Solaris and probably on all systems that use ELF object file format. On some versions of Irix on SGI machines and OSF/1 on Alpha systems, you will not be able to compare the files without specifying `-save-temps'; see the description of individual systems above to see if you get

comparison failures. You may have similar problems on other systems.

Use this command to compare the files:

```
make compare
```

This will mention any object files that differ between stage 2 and stage 3. Any difference, no matter how innocuous, indicates that the stage 2 compiler has compiled GNU CC incorrectly, and is therefore a potentially serious bug which you should investigate and report (see section Reporting Bugs).

If your system does not put time stamps in the object files, then this is a faster way to compare them (using the Bourne shell):

```
for file in *.o; do
cmp $file stage2/$file
done
```

If you have built the compiler with the `-mno-mips-tfile' option on MIPS machines, you will not be able to compare the files.

15. Build the Objective C library (if you have built the Objective C compiler). Here is the command to do this:

    ```
    make objc-runtime CC="stage2/xgcc -Bstage2/" CFLAGS="-g -O"
    ```

16. Install the compiler driver, the compiler's passes and run-time support with `make install'. Use the same value for `CC`, `CFLAGS` and `LANGUAGES` that you used when compiling the files that are being installed. One reason this is necessary is that some versions of Make have bugs and recompile files gratuitously when you do this step. If you use the same variable values, those files will be recompiled properly.

    For example, if you have built the stage 2 compiler, you can use the following command:

    ```
    make install CC="stage2/xgcc -Bstage2/" CFLAGS="-g -O" LANGUAGES="list"
    ```

    This copies the files `cc1', `cpp' and `libgcc.a' to files `cc1', `cpp' and `libgcc.a' in the directory `/usr/local/lib/gcc-lib/target/version', which is where the compiler driver program looks for them. Here target is the target machine type specified when you ran `configure', and version is the version number of GNU CC. This naming scheme permits various versions and/or cross-compilers to coexist.

    This also copies the driver program `xgcc' into `/usr/local/bin/gcc', so that it appears in typical execution search paths.

    On some systems, this command causes recompilation of some files. This is usually due to bugs in `make`. You should either ignore this problem, or use GNU Make.

    Warning: there is a bug in `alloca` in the Sun library. To avoid this bug, be sure to install the executables of GNU CC that were compiled by GNU CC. (That is, the executables from stage 2 or 3, not stage 1.) They use `alloca` as a built-in function and never the one in the library.

    (It is usually better to install GNU CC executables from stage 2 or 3, since they usually run faster than the ones compiled with some other compiler.)

17. Install the Objective C library (if you are installing the Objective C compiler). Here is the command to do this:

    ```
    make install-libobjc CC="stage2/xgcc -Bstage2/" CFLAGS="-g -O"
    ```

18. If you're going to use C++, it's likely that you need to also install the libg++ distribution. It should be available from the same place where you got the GNU C distribution. Just as GNU C does not distribute a C runtime library, it also does not include a C++ run-time library. All I/O functionality, special class libraries, etc., are available in the libg++ distribution.

Configurations Supported by GNU CC

Here are the possible CPU types:

> 1750a, a29k, alpha, arm, cn, clipper, dsp16xx, elxsi, h8300, hppa1.0, hppa1.1, i370, i386, i486, i586, i860, i960, m68000, m68k, m88k, mips, mipsel, mips64, mips64el, ns32k, powerpc, powerpcle, pyramid, romp, rs6000, sh, sparc, sparclite, sparc64, vax, we32k.

Here are the recognized company names. As you can see, customary abbreviations are used rather than the longer official names.

> acorn, alliant, altos, apollo, att, bull, cbm, convergent, convex, crds, dec, dg, dolphin, elxsi, encore, harris, hitachi, hp, ibm, intergraph, isi, mips, motorola, ncr, next, ns, omron, plexus, sequent, sgi, sony, sun, tti, unicom, wrs.

The company name is meaningful only to disambiguate when the rest of the information supplied is insufficient. You can omit it, writing just `cpu-system', if it is not needed. For example, `vax-ultrix4.2' is equivalent to `vax-dec-ultrix4.2'.

Here is a list of system types:

> 386bsd, aix, acis, amigados, aos, aout, bosx, bsd, clix, coff, ctix, cxux, dgux, dynix, ebmon, ecoff, elf, esix, freebsd, hms, genix, gnu, gnu/linux, hiux, hpux, iris, irix, isc, luna, lynxos, mach, minix, msdos, mvs, netbsd, newsos, nindy, ns, osf, osfrose, ptx, riscix, riscos, rtu, sco, sim, solaris, sunos, sym, sysv, udi, ultrix, unicos, uniplus, unos, vms, vsta, vxworks, winnt, xenix.

You can omit the system type; then `configure' guesses the operating system from the CPU and company.

You can add a version number to the system type; this may or may not make a difference. For example, you can write `bsd4.3' or `bsd4.4' to distinguish versions of BSD. In practice, the version number is most needed for `sysv3' and `sysv4', which are often treated differently.

If you specify an impossible combination such as `i860-dg-vms', then you may get an error message from `configure', or it may ignore part of the information and do the best it can with the rest. `configure' always prints the canonical name for the alternative that it used. GNU CC does not support all possible alternatives.

Often a particular model of machine has a name. Many machine names are recognized as aliases for CPU/company combinations. Thus, the machine name `sun3', mentioned above, is an alias for `m68k-sun'. Sometimes we accept a company name as a machine name, when the name is popularly used for a particular machine. Here is a table of the known machine names:

> 3300, 3b1, 3bn, 7300, altos3068, altos, apollo68, att-7300, balance, convex-cn, crds, decstation-3100, decstation, delta, encore, fx2800, gmicro, hp7nn, hp8nn, hp9k2nn, hp9k3nn,

hp9k7nn, hp9k8nn, iris4d, iris, isi68, m3230, magnum, merlin, miniframe, mmax, news-3600, news800, news, next, pbd, pc532, pmax, powerpc, powerpcle, ps2, risc-news, rtpc, sun2, sun386i, sun386, sun3, sun4, symmetry, tower-32, tower.

Remember that a machine name specifies both the cpu type and the company name. If you want to install your own homemade configuration files, you can use `local' as the company name to access them. If you use configuration `cpu-local', the configuration name without the cpu prefix is used to form the configuration file names.

Thus, if you specify `m68k-local', configuration uses files `m68k.md', `local.h', `m68k.c', `xm-local.h', `t-local', and `x-local', all in the directory `config/m68k'.

Here is a list of configurations that have special treatment or special things you must know:

`1750a-*-*'

MIL-STD-1750A processors.

Starting with GCC 2.6.1, the MIL-STD-1750A cross configuration no longer supports the Tektronix Assembler, but instead produces output for `as1750`, an assembler/linker available under the GNU Public License for the 1750A. Contact *okellogg@salyko.cube.net* for more details on obtaining `as1750'. A similarly licensed simulator for the 1750A is available from same address.

You should ignore a fatal error during the building of libgcc (libgcc is not yet implemented for the 1750A.)

The `as1750` assembler requires the file `ms1750.inc', which is found in the directory `config/1750a'.

GNU CC produced the same sections as the Fairchild F9450 C Compiler, namely:

`NREL`
: The program code section.

`SREL`
: The read/write (RAM) data section.

`KREL`
: The read-only (ROM) constants section.

`IREL`
: Initialization section (code to copy KREL to SREL).

The smallest addressable unit is 16 bits (BITS_PER_UNIT is 16). This means that type `char' is represented with a 16-bit word per character. The 1750A's "Load/Store Upper/Lower Byte" instructions are not used by GNU CC.

There is a problem with long argument lists to functions. The compiler aborts if the sum of space needed by all arguments exceeds 14 words. This is because the arguments are passed in registers (R0..R13) not on the stack, and there is a problem with passing further arguments (i.e. beyond those in R0..R13) via the stack.

If efficiency is less important than using long argument lists, you can change the definition of the `FUNCTION_ARG` macro in `config/1750/1750a.h' to always return zero. If you do that, GNU CC will pass all parameters on the stack.

- alpha-*-osf1 Systems using processors that implement the DEC Alpha architecture and are running the OSF/1

operating system, for example the DEC Alpha AXP systems. (VMS on the Alpha is not currently supported GNU CC.)

GNU CC writes a `.verstamp' directive to the assembler output file unless it is built as a cross-compiler. It gets the version to use from the system header file `/usr/include/stamp.h'. If you install a new version of OSF/1, you should rebuild GCC to pick up the new version stamp.

Note that since the Alpha is a 64-bit architecture, cross-compilers from 32-bit machines will not generate code as efficient as that generated when the compiler is running on a 64-bit machine because many optimizations that depend on being able to represent a word on the target in an integral value on the host cannot be performed. Building cross-compilers on the Alpha for 32-bit machines has only been tested in a few cases and may not work properly.

`make compare` may fail on old versions of OSF/1 unless you add `-save-temps' to `CFLAGS`. On these systems, the name of the assembler input file is stored in the object file, and that makes comparison fail if it differs between the `stage1` and `stage2` compilations. The option `-save-temps' forces a fixed name to be used for the assembler input file, instead of a randomly chosen name in `/tmp'. Do not add `-save-temps' unless the comparisons fail without that option. If you add `-save-temps', you will have to manually delete the `.i' and `.s' files after each series of compilations.

GNU CC now supports both the native (ECOFF) debugging format used by DBX and GDB and an encapsulated STABS format for use only with GDB. See the discussion of the `--with-stabs' option of `configure' above for more information on these formats and how to select them.

There is a bug in DEC's assembler that produces incorrect line numbers for ECOFF format when the `.align' directive is used. To work around this problem, GNU CC will not emit such alignment directives while writing ECOFF format debugging information even if optimization is being performed. Unfortunately, this has the very undesirable side-effect that code addresses when `-O' is specified are different depending on whether or not `-g' is also specified.

To avoid this behavior, specify `-gstabs+' and use GDB instead of DBX. DEC is now aware of this problem with the assembler and hopes to provide a fix shortly.

- arm Advanced RISC Machines ARM-family processors. These are often used in embedded applications. There are no standard Unix configurations. This configuration corresponds to the basic instruction sequences and will produce a.out format object modules.

You may need to make a variant of the file `arm.h' for your particular configuration.

- arm-*-riscix The ARM2 or ARM3 processor running RISC iX, Acorn's port of BSD Unix. If you are running a version of RISC iX prior to 1.2 then you must specify the version number during configuration. Note that the assembler shipped with RISC iX does not support stabs debugging information; a new version of the assembler, with stabs support included, is now available from Acorn.
- a29k AMD Am29k-family processors. These are normally used in embedded applications. There are no standard Unix configurations. This configuration corresponds to AMD's standard calling sequence and binary interface and is compatible with other 29k tools.

You may need to make a variant of the file `a29k.h' for your particular configuration.

- a29k-*-bsd AMD Am29050 used in a system running a variant of BSD Unix.
- decstation-* DECstations can support three different personalities: Ultrix, DEC OSF/1, and OSF/rose. To configure GCC for these platforms use the following configurations:

`decstation-ultrix'

Ultrix configuration.

`decstation-osf1'

Dec's version of OSF/1.

`decstation-osfrose'

Open Software Foundation reference port of OSF/1 which uses the OSF/rose object file format instead of ECOFF. Normally, you would not select this configuration.

The MIPS C compiler needs to be told to increase its table size for switch statements with the `-Wf,-XNg1500' option in order to compile `cp/parse.c'. If you use the `-O2' optimization option, you also need to use `-Olimit 3000'. Both of these options are automatically generated in the `Makefile' that the shell script `configure' builds. If you override the CC make variable and use the MIPS compilers, you may need to add `-Wf,-XNg1500 -Olimit 3000'.

- elxsi-elxsi-bsd The Elxsi's C compiler has known limitations that prevent it from compiling GNU C. Please contact mrs@cygnus.com for more details.
- dsp16xx A port to the AT&T DSP1610 family of processors.
- h8300-*-* The calling convention and structure layout has changed in release 2.6. All code must be recompiled. The calling convention now passes the first three arguments in function calls in registers. Structures are no longer a multiple of 2 bytes.
- hppa*-*-* There are two variants of this CPU, called 1.0 and 1.1, which have different machine descriptions. You must use the right one for your machine. All 7nn machines and 8n7 machines use 1.1, while all other 8nn machines use 1.0.

The easiest way to handle this problem is to use `configure hpnnn' or `configure hpnnn-hpux', where nnn is the model number of the machine. Then `configure' will figure out if the machine is a 1.0 or 1.1. Use `uname -a' to find out the model number of your machine.

`-g' does not work on HP-UX, since that system uses a peculiar debugging format which GNU CC does not know about. However, `-g' will work if you also use GAS and GDB in conjunction with GCC. We highly recommend using GAS for all HP-PA configurations.

You should be using GAS-2.3 (or later) along with GDB-4.12 (or later). These can be retrieved from all the traditional GNU ftp archive sites.

Build GAS and install the resulting binary as:

```
/usr/local/lib/gcc-lib/configuration/gccversion/as
```

where configuration is the configuration name (perhaps `hpnnn-hpux') and gccversion is the GNU CC version number. Do this *before* starting the build process, otherwise you will get errors from the HPUX assembler while building `libgcc2.a'. The command

```
make install-dir
```

will create the necessary directory hierarchy so you can install GAS before building GCC.

To enable debugging, configure GNU CC with the `--with-gnu-as' option before building.

It has been reported that GNU CC produces invalid assembly code for 1.1 machines running HP-UX 8.02 when using the HP assembler. Typically the errors look like this:


```
as: bug.s @line#15 [err#1060]
  Argument 0 or 2 in FARG upper
          - lookahead = ARGW1=FR,RTNVAL=GR
as: foo.s @line#28 [err#1060]
  Argument 0 or 2 in FARG upper
          - lookahead = ARGW1=FR
```

You can check the version of HP-UX you are running by executing the command `uname -r'. If you are indeed running HP-UX 8.02 on a PA and using the HP assembler then configure GCC with "hpnnn-hpux8.02".

- i370-*-* This port is very preliminary and has many known bugs. We hope to have a higher-quality port for this machine soon.
- i386-*-linuxoldld Use this configuration to generate a.out binaries on Linux if you do not have gas/binutils version 2.5.2 or later installed. This is an obsolete configuration.
- i386-*-linuxaout Use this configuration to generate a.out binaries on Linux. This is an obsolete configuration. You must use gas/binutils version 2.5.2 or later.
- i386-*-linux Use this configuration to generate ELF binaries on Linux. You must use gas/binutils version 2.5.2 or later.
- i386-*-sco Compilation with RCC is recommended. Also, it may be a good idea to link with GNU malloc instead of the malloc that comes with the system.
- i386-*-sco3.2v4 Use this configuration for SCO release 3.2 version 4.
- i386-*-isc It may be a good idea to link with GNU malloc instead of the malloc that comes with the system.

In ISC version 4.1, `sed' core dumps when building `deduced.h'. Use the version of `sed' from version 4.0.

- i386-*-esix It may be good idea to link with GNU malloc instead of the malloc that comes with the system.
- i386-ibm-aix You need to use GAS version 2.1 or later, and and LD from GNU binutils version 2.2 or later.
- i386-sequent-bsd Go to the Berkeley universe before compiling. In addition, you probably need to create a file named `string.h' containing just one line: `#include <strings.h>'.
- i386-sequent-ptx1* Sequent DYNIX/ptx 1.x.
- i386-sequent-ptx2* Sequent DYNIX/ptx 2.x.
- i386-sun-sunos4 You may find that you need another version of GNU CC to begin bootstrapping with, since the current version when built with the system's own compiler seems to get an infinite loop compiling part of `libgcc2.c'. GNU CC version 2 compiled with GNU CC (any version) seems not to have this problem.

See section Installing GNU CC on the Sun, for information on installing GNU CC on Sun systems.

- i[345]86-*-winnt3.5 This version requires a GAS that has not let been released. Until it is, you can get a prebuilt binary version via anonymous ftp from `cs.washington.edu:pub/gnat' or `cs.nyu.edu:pub/gnat'. You must also use the Microsoft header files from the Windows NT 3.5 SDK. Find these on the CDROM in the `/mstools/h' directory dated 9/4/94. You must use a fixed version of Microsoft linker made especially for NT 3.5, which is also is available on the NT 3.5 SDK CDROM. If you do not have this linker, can you also use the linker from Visual C/C++ 1.0 or 2.0.

Installing GNU CC for NT builds a wrapper linker, called `ld.exe', which mimics the behaviour of Unix `ld' in the specification of libraries (`-L' and `-l'). `ld.exe' looks for both Unix and Microsoft named libraries. For example, if you specify `-lfoo', `ld.exe' will look first for `libfoo.a' and then for

`foo.lib'.

You may install GNU CC for Windows NT in one of two ways, depending on whether or not you have a Unix-like shell and various Unix-like utilities.

1. If you do not have a Unix-like shell and few Unix-like utilities, you will use a DOS style batch script called `configure.bat'. Invoke it as `configure winnt` from an MSDOS console window or from the program manager dialog box. `configure.bat' assumes you have already installed and have in your path a Unix-like `sed' program which is used to create a working `Makefile' from `Makefile.in'.

   `Makefile' uses the Microsoft Nmake program maintenance utility and the Visual C/C++ V8.00 compiler to build GNU CC. You need only have the utilities `sed' and `touch' to use this installation method, which only automatically builds the compiler itself. You must then examine what `fixinc.winnt' does, edit the header files by hand and build `libgcc.a' manually.
2. The second type of installation assumes you are running a Unix-like shell, have a complete suite of Unix-like utilities in your path, and have a previous version of GNU CC already installed, either through building it via the above installation method or acquiring a pre-built binary. In this case, use the `configure' script in the normal fashion.

- i860-intel-osf1 This is the Paragon. If you have version 1.0 of the operating system, see section Installation Problems, for special things you need to do to compensate for peculiarities in the system.
- *-lynx-lynxos LynxOS 2.2 and earlier comes with GNU CC 1.x already installed as `/bin/gcc'. You should compile with this instead of `/bin/cc'. You can tell GNU CC to use the GNU assembler and linker, by specifying `--with-gnu-as --with-gnu-ld' when configuring. These will produce COFF format object files and executables; otherwise GNU CC will use the installed tools, which produce a.out format executables.
- m68000-hp-bsd HP 9000 series 200 running BSD. Note that the C compiler that comes with this system cannot compile GNU CC; contact `law@cs.utah.edu` to get binaries of GNU CC for bootstrapping.
- m68k-altos Altos 3068. You must use the GNU assembler, linker and debugger. Also, you must fix a kernel bug. Details in the file `README.ALTOS'.
- m68k-att-sysv AT&T 3b1, a.k.a. 7300 PC. Special procedures are needed to compile GNU CC with this machine's standard C compiler, due to bugs in that compiler. You can bootstrap it more easily with previous versions of GNU CC if you have them.

Installing GNU CC on the 3b1 is difficult if you do not already have GNU CC running, due to bugs in the installed C compiler. However, the following procedure might work. We are unable to test it.

1. Comment out the `#include "config.h"' line on line 37 of `cccp.c' and do `make cpp'. This makes a preliminary version of GNU cpp.
2. Save the old `/lib/cpp' and copy the preliminary GNU cpp to that file name.
3. Undo your change in `cccp.c', or reinstall the original version, and do `make cpp' again.
4. Copy this final version of GNU cpp into `/lib/cpp'.
5. Replace every occurrence of `obstack_free` in the file `tree.c' with `_obstack_free`.
6. Run `make` to get the first-stage GNU CC.
7. Reinstall the original version of `/lib/cpp'.
8. Now you can compile GNU CC with itself and install it in the normal fashion.

- m68k-bull-sysv Bull DPX/2 series 200 and 300 with BOS-2.00.45 up to BOS-2.01. GNU CC works either with native assembler or GNU assembler. You can use GNU assembler with native coff generation by providing

`--with-gnu-as' to the configure script or use GNU assembler with dbx-in-coff encapsulation by providing `--with-gnu-as --stabs'. For any problem with native assembler or for availability of the DPX/2 port of GAS contact `F.Pierresteguy@frcl.bull.fr'.

- m68k-crds-unox Use `configure unos' for building on Unos.

The Unos assembler is named `casm` instead of `as`. For some strange reason linking `` `/bin/as' `` to `` `/bin/casm' `` changes the behavior, and does not work. So, when installing GNU CC, you should install the following script as `` `as' `` in the subdirectory where the passes of GCC are installed:

```
#!/bin/sh
casm $*
```

The default Unos library is named `` `libunos.a' `` instead of `` `libc.a' ``. To allow GNU CC to function, either change all references to `-lc' in `` `gcc.c' `` to `-lunos' or link `` `/lib/libc.a' `` to `` `/lib/libunos.a' ``.

When compiling GNU CC with the standard compiler, to overcome bugs in the support of `alloca`, do not use `-O' when making stage 2. Then use the stage 2 compiler with `-O' to make the stage 3 compiler. This compiler will have the same characteristics as the usual stage 2 compiler on other systems. Use it to make a stage 4 compiler and compare that with stage 3 to verify proper compilation.

(Perhaps simply defining `ALLOCA` in `` `x-crds' `` as described in the comments there will make the above paragraph superfluous. Please inform us of whether this works.)

Unos uses memory segmentation instead of demand paging, so you will need a lot of memory. 5 Mb is barely enough if no other tasks are running. If linking `` `cc1' `` fails, try putting the object files into a library and linking from that library.

- m68k-hp-hpux HP 9000 series 300 or 400 running HP-UX. HP-UX version 8.0 has a bug in the assembler that prevents compilation of GNU CC. To fix it, get patch PHCO_4484 from HP.

In addition, if you wish to use gas `--with-gnu-as' you must use gas version 2.1 or later, and you must use the GNU linker version 2.1 or later. Earlier versions of gas relied upon a program which converted the gas output into the native HP/UX format, but that program has not been kept up to date. gdb does not understand that native HP/UX format, so you must use gas if you wish to use gdb.

- m68k-sun Sun 3. We do not provide a configuration file to use the Sun FPA by default, because programs that establish signal handlers for floating point traps inherently cannot work with the FPA.

See section Installing GNU CC on the Sun, for information on installing GNU CC on Sun systems.

- m88k-*-svr3 Motorola m88k running the AT&T/Unisoft/Motorola V.3 reference port. These systems tend to use the Green Hills C, revision 1.8.5, as the standard C compiler. There are apparently bugs in this compiler that result in object files differences between stage 2 and stage 3. If this happens, make the stage 4 compiler and compare it to the stage 3 compiler. If the stage 3 and stage 4 object files are identical, this suggests you encountered a problem with the standard C compiler; the stage 3 and 4 compilers may be usable.

It is best, however, to use an older version of GNU CC for bootstrapping if you have one.

- m88k-*-dgux Motorola m88k running DG/UX. To build 88open BCS native or cross compilers on DG/UX, specify the configuration name as `m88k-*-dguxbcs' and build in the 88open BCS software development environment. To build ELF native or cross compilers on DG/UX, specify `m88k-*-dgux' and build in the DG/UX ELF development environment. You set the software development environment by issuing `sde-target' command and specifying either `m88kbcs' or `m88kdguxelf' as the operand.

If you do not specify a configuration name, `configure' guesses the configuration based on the current software development environment.

- m88k-tektronix-sysv3 Tektronix XD88 running UTekV 3.2e. Do not turn on optimization while building stage1 if you bootstrap with the buggy Green Hills compiler. Also, The bundled LAI System V NFS is buggy so if you build in an NFS mounted directory, start from a fresh reboot, or avoid NFS all together. Otherwise you may have trouble getting clean comparisons between stages.
- mips-mips-bsd MIPS machines running the MIPS operating system in BSD mode. It's possible that some old versions of the system lack the functions `memcpy`, `memcmp`, and `memset`. If your system lacks these, you must remove or undo the definition of `TARGET_MEM_FUNCTIONS` in `mips-bsd.h'.

The MIPS C compiler needs to be told to increase its table size for switch statements with the `-Wf,-XNg1500' option in order to compile `cp/parse.c'. If you use the `-O2' optimization option, you also need to use `-Olimit 3000'. Both of these options are automatically generated in the `Makefile' that the shell script `configure' builds. If you override the `CC` make variable and use the MIPS compilers, you may need to add `-Wf,-XNg1500 -Olimit 3000'.

- mips-mips-riscos* The MIPS C compiler needs to be told to increase its table size for switch statements with the `-Wf,-XNg1500' option in order to compile `cp/parse.c'. If you use the `-O2' optimization option, you also need to use `-Olimit 3000'. Both of these options are automatically generated in the `Makefile' that the shell script `configure' builds. If you override the `CC` make variable and use the MIPS compilers, you may need to add `-Wf,-XNg1500 -Olimit 3000'.

MIPS computers running RISC-OS can support four different personalities: default, BSD 4.3, System V.3, and System V.4 (older versions of RISC-OS don't support V.4). To configure GCC for these platforms use the following configurations:

`mips-mips-riscos`rev`'
: Default configuration for RISC-OS, revision `rev`.

`mips-mips-riscos`rev`bsd'
: BSD 4.3 configuration for RISC-OS, revision `rev`.

`mips-mips-riscos`rev`sysv4'
: System V.4 configuration for RISC-OS, revision `rev`.

`mips-mips-riscos`rev`sysv'
: System V.3 configuration for RISC-OS, revision `rev`.

The revision `rev` mentioned above is the revision of RISC-OS to use. You must reconfigure GCC when going from a RISC-OS revision 4 to RISC-OS revision 5. This has the effect of avoiding a linker bug (see section Installation Problems, for more details).

- mips-sgi-* In order to compile GCC on an SGI running IRIX 4, the "c.hdr.lib" option must be installed from the CD-ROM supplied from Silicon Graphics. This is found on the 2nd CD in release 4.0.1.

In order to compile GCC on an SGI running IRIX 5, the "compiler_dev.hdr" subsystem must be installed from the IDO CD-ROM supplied by Silicon Graphics.

`make compare` may fail on version 5 of IRIX unless you add `-save-temps' to `CFLAGS`. On these systems, the name of the assembler input file is stored in the object file, and that makes comparison fail if it differs between the `stage1` and `stage2` compilations. The option `-save-temps' forces a fixed name to be used for the assembler input file, instead of a randomly chosen name in `/tmp'. Do not add `-save-temps' unless the comparisons fail without that option. If you do you `-save-temps', you will have to manually delete the `.i' and

`.s' files after each series of compilations.

The MIPS C compiler needs to be told to increase its table size for switch statements with the `-Wf,-XNg1500' option in order to compile `cp/parse.c'. If you use the `-O2' optimization option, you also need to use `-Olimit 3000'. Both of these options are automatically generated in the `Makefile' that the shell script `configure' builds. If you override the CC make variable and use the MIPS compilers, you may need to add `-Wf,-XNg1500 -Olimit 3000'.

On Irix version 4.0.5F, and perhaps on some other versions as well, there is an assembler bug that reorders instructions incorrectly. To work around it, specify the target configuration `mips-sgi-irix4loser'. This configuration inhibits assembler optimization.

In a compiler configured with target `mips-sgi-irix4', you can turn off assembler optimization by using the `-noasmopt' option. This compiler option passes the option `-O0' to the assembler, to inhibit reordering.

The `-noasmopt' option can be useful for testing whether a problem is due to erroneous assembler reordering. Even if a problem does not go away with `-noasmopt', it may still be due to assembler reordering--perhaps GNU CC itself was miscompiled as a result.

To enable debugging under Irix 5, you must use GNU as 2.5 or later, and use the `--with-gnu-as' configure option when configuring gcc. GNU as is distributed as part of the binutils package.

- mips-sony-sysv Sony MIPS NEWS. This works in NEWSOS 5.0.1, but not in 5.0.2 (which uses ELF instead of COFF). Support for 5.0.2 will probably be provided soon by volunteers. In particular, the linker does not like the code generated by GCC when shared libraries are linked in.
- ns32k-encore Encore ns32000 system. Encore systems are supported only under BSD.
- ns32k-*-genix National Semiconductor ns32000 system. Genix has bugs in alloca and malloc; you must get the compiled versions of these from GNU Emacs.
- ns32k-sequent Go to the Berkeley universe before compiling. In addition, you probably need to create a file named `string.h' containing just one line: `#include <strings.h>'.
- ns32k-utek UTEK ns32000 system ("merlin"). The C compiler that comes with this system cannot compile GNU CC; contact `tektronix!reed!mason' to get binaries of GNU CC for bootstrapping.
- romp-*-aos
- romp-*-mach The only operating systems supported for the IBM RT PC are AOS and MACH. GNU CC does not support AIX running on the RT. We recommend you compile GNU CC with an earlier version of itself; if you compile GNU CC with hc, the Metaware compiler, it will work, but you will get mismatches between the stage 2 and stage 3 compilers in various files. These errors are minor differences in some floating-point constants and can be safely ignored; the stage 3 compiler is correct.
- rs6000-*-aix
- powerpc-*-aix Various early versions of each release of the IBM XLC compiler will not bootstrap GNU CC. Symptoms include differences between the stage2 and stage3 object files, and errors when compiling `libgcc.a' or `enquire'. Known problematic releases include: xlc-1.2.1.8, xlc-1.3.0.0 (distributed with AIX 3.2.5), and xlc-1.3.0.19. Both xlc-1.2.1.28 and xlc-1.3.0.24 (PTF 432238) are known to produce working versions of GNU CC, but most other recent releases correctly bootstrap GNU CC. Also, releases of AIX prior to AIX 3.2.4 include a version of the IBM assembler which does not accept debugging directives: assembler updates are available as PTFs. See the file `README.RS6000' for more details on both of these problems.

Only AIX is supported on the PowerPC. GNU CC does not yet support the 64-bit PowerPC instructions.

Objective C does not work on this architecture.

AIX on the RS/6000 provides support (NLS) for environments outside of the United States. Compilers and assemblers use NLS to support locale-specific representations of various objects including floating-point numbers ("." vs "," for separating decimal fractions). There have been problems reported where the library linked with GNU CC does not produce the same floating-point formats that the assembler accepts. If you have this problem, set the LANG environment variable to "C" or "En_US".

- powerpc-*-elf
- powerpc-*-sysv4 PowerPC system in big endian mode, running System V.4. This system is currently under development.
- powerpc-*-eabi Embedded PowerPC system in big endian mode. This system is currently under development.
- powerpcle-*-elf
- powerpcle-*-eabi PowerPC system in little endian mode, running System V.4. This system is currently under development.
- powerpcle-*-sysv4 Embedded PowerPC system in little endian mode. This system is currently under development.
- vax-dec-ultrix Don't try compiling with Vax C (`vcc`). It produces incorrect code in some cases (for example, when `alloca` is used).

Meanwhile, compiling `` `cp/parse.c' `` with pcc does not work because of an internal table size limitation in that compiler. To avoid this problem, compile just the GNU C compiler first, and use it to recompile building all the languages that you want to run.

- sparc-sun-* See section Installing GNU CC on the Sun, for information on installing GNU CC on Sun systems.
- vax-dec-vms See section Installing GNU CC on VMS, for details on how to install GNU CC on VMS.
- we32k-*-* These computers are also known as the 3b2, 3b5, 3b20 and other similar names. (However, the 3b1 is actually a 68000; see section Configurations Supported by GNU CC.)

Don't use `-g' when compiling with the system's compiler. The system's linker seems to be unable to handle such a large program with debugging information.

The system's compiler runs out of capacity when compiling `` `stmt.c' `` in GNU CC. You can work around this by building `` `cpp' `` in GNU CC first, then use that instead of the system's preprocessor with the system's C compiler to compile `` `stmt.c' ``. Here is how:

```
mv /lib/cpp /lib/cpp.att
cp cpp /lib/cpp.gnu
echo '/lib/cpp.gnu -traditional ${1+"$@"}' > /lib/cpp
chmod +x /lib/cpp
```

The system's compiler produces bad code for some of the GNU CC optimization files. So you must build the stage 2 compiler without optimization. Then build a stage 3 compiler with optimization. That executable should work. Here are the necessary commands:

```
make LANGUAGES=c CC=stage1/xgcc CFLAGS="-Bstage1/ -g"
make stage2
make CC=stage2/xgcc CFLAGS="-Bstage2/ -g -O"
```

You may need to raise the ULIMIT setting to build a C++ compiler, as the file `` `cc1plus' `` is larger than one

megabyte.

Compilation in a Separate Directory

If you wish to build the object files and executables in a directory other than the one containing the source files, here is what you must do differently:

1. Make sure you have a version of Make that supports the `VPATH` feature. (GNU Make supports it, as do Make versions on most BSD systems.)
2. If you have ever run `` `configure' `` in the source directory, you must undo the configuration. Do this by running:

   ```
   make distclean
   ```
3. Go to the directory in which you want to build the compiler before running `` `configure' ``:

   ```
   mkdir gcc-sun3
   cd gcc-sun3
   ```

   On systems that do not support symbolic links, this directory must be on the same file system as the source code directory.
4. Specify where to find `` `configure' `` when you run it:

   ```
   ../gcc/configure ...
   ```

   This also tells `configure` where to find the compiler sources; `configure` takes the directory from the file name that was used to invoke it. But if you want to be sure, you can specify the source directory with the `--srcdir' option, like this:

   ```
   ../gcc/configure --srcdir=../gcc other options
   ```

   The directory you specify with `--srcdir' need not be the same as the one that `configure` is found in.

Now, you can run `make` in that directory. You need not repeat the configuration steps shown above, when ordinary source files change. You must, however, run `configure` again when the configuration files change, if your system does not support symbolic links.

Building and Installing a Cross-Compiler

GNU CC can function as a cross-compiler for many machines, but not all.

- Cross-compilers for the Mips as target using the Mips assembler currently do not work, because the auxiliary programs `` `mips-tdump.c' `` and `` `mips-tfile.c' `` can't be compiled on anything but a Mips. It does work to cross compile for a Mips if you use the GNU assembler and linker.
- Cross-compilers between machines with different floating point formats have not all been made to work. GNU CC now has a floating point emulator with which these can work, but each target machine description needs to be updated to take advantage of it.
- Cross-compilation between machines of different word sizes is somewhat problematic and sometimes does not work.

Since GNU CC generates assembler code, you probably need a cross-assembler that GNU CC can run, in order to produce object files. If you want to link on other than the target machine, you need a cross-linker as well. You also need header files and libraries suitable for the target machine that you can install on the host machine.

Steps of Cross-Compilation

To compile and run a program using a cross-compiler involves several steps:

- Run the cross-compiler on the host machine to produce assembler files for the target machine. This requires header files for the target machine.
- Assemble the files produced by the cross-compiler. You can do this either with an assembler on the target machine, or with a cross-assembler on the host machine.
- Link those files to make an executable. You can do this either with a linker on the target machine, or with a cross-linker on the host machine. Whichever machine you use, you need libraries and certain startup files (typically `` `crt....o' ``) for the target machine.

It is most convenient to do all of these steps on the same host machine, since then you can do it all with a single invocation of GNU CC. This requires a suitable cross-assembler and cross-linker. For some targets, the GNU assembler and linker are available.

Configuring a Cross-Compiler

To build GNU CC as a cross-compiler, you start out by running `` `configure' ``. Use the `--target=target' to specify the target type. If `` `configure' `` was unable to correctly identify the system you are running on, also specify the `--build=build' option. For example, here is how to configure for a cross-compiler that produces code for an HP 68030 system running BSD on a system that `` `configure' `` can correctly identify:

```
./configure --target=m68k-hp-bsd4.3
```

Tools and Libraries for a Cross-Compiler

If you have a cross-assembler and cross-linker available, you should install them now. Put them in the directory `` `/usr/local/target/bin' ``. Here is a table of the tools you should put in this directory:

`` `as' ``
: This should be the cross-assembler.

`` `ld' ``
: This should be the cross-linker.

`` `ar' ``
: This should be the cross-archiver: a program which can manipulate archive files (linker libraries) in the target machine's format.

`` `ranlib' ``
: This should be a program to construct a symbol table in an archive file.

The installation of GNU CC will find these programs in that directory, and copy or link them to the proper place to for the cross-compiler to find them when run later.

The easiest way to provide these files is to build the Binutils package and GAS. Configure them with the same

`--host' and `--target' options that you use for configuring GNU CC, then build and install them. They install their executables automatically into the proper directory. Alas, they do not support all the targets that GNU supports.

If you want to install libraries to use with the cross-compiler, such as a standard C library, put them in the directory `/usr/local/target/lib'; installation of GNU CC copies all all the files in that subdirectory into the proper place for GNU CC to find them and link with them. Here's an example of copying some libraries from a target machine:

```
ftp target-machine
lcd /usr/local/target/lib
cd /lib
get libc.a
cd /usr/lib
get libg.a
get libm.a
quit
```

The precise set of libraries you'll need, and their locations on the target machine, vary depending on its operating system.

Many targets require "start files" such as `crt0.o' and `crtn.o' which are linked into each executable; these too should be placed in `/usr/local/target/lib'. There may be several alternatives for `crt0.o', for use with profiling or other compilation options. Check your target's definition of STARTFILE_SPEC to find out what start files it uses. Here's an example of copying these files from a target machine:

```
ftp target-machine
lcd /usr/local/target/lib
prompt
cd /lib
mget *crt*.o
cd /usr/lib
mget *crt*.o
quit
```

`libgcc.a' and Cross-Compilers

Code compiled by GNU CC uses certain runtime support functions implicitly. Some of these functions can be compiled successfully with GNU CC itself, but a few cannot be. These problem functions are in the source file `libgcc1.c'; the library made from them is called `libgcc1.a'.

When you build a native compiler, these functions are compiled with some other compiler--the one that you use for bootstrapping GNU CC. Presumably it knows how to open code these operations, or else knows how to call the run-time emulation facilities that the machine comes with. But this approach doesn't work for building a cross-compiler. The compiler that you use for building knows about the host system, not the target system.

So, when you build a cross-compiler you have to supply a suitable library `libgcc1.a' that does the job it is expected to do.

To compile `libgcc1.c' with the cross-compiler itself does not work. The functions in this file are supp to implement arithmetic operations that GNU CC does not know how to open code for your target machine. these functions are compiled with GNU CC itself, they will compile into infinite recursion.

On any given target, most of these functions are not needed. If GNU CC can open code an arithmetic operation, it will not call these functions to perform the operation. It is possible that on your target machine, none of these functions is needed. If so, you can supply an empty library as `libgcc1.a'.

Many targets need library support only for multiplication and division. If you are linking with a library that contains functions for multiplication and division, you can tell GNU CC to call them directly by defining the macros `MULSI3_LIBCALL`, and the like. These macros need to be defined in the target description macro file. For some targets, they are defined already. This may be sufficient to avoid the need for libgcc1.a; if so, you can supply an empty library.

Some targets do not have floating point instructions; they need other functions in `libgcc1.a', which do floating arithmetic. Recent versions of GNU CC have a file which emulates floating point. With a certain amount of work, you should be able to construct a floating point emulator that can be used as `libgcc1.a'. Perhaps future versions will contain code to do this automatically and conveniently. That depends on whether someone wants to implement it.

Some embedded targets come with all the necessary `libgcc1.a' routines written in C or assembler. These targets build `libgcc1.a' automatically and you do not need to do anything special for them. Other embedded targets do not need any `libgcc1.a' routines since all the necessary operations are supported by the hardware.

If your target system has another C compiler, you can configure GNU CC as a native compiler on that machine, build just `libgcc1.a' with `make libgcc1.a' on that machine, and use the resulting file with the cross-compiler. To do this, execute the following on the target machine:

```
cd target-build-dir
./configure --host=sparc --target=sun3
make libgcc1.a
```

And then this on the host machine:

```
ftp target-machine
binary
cd target-build-dir
get libgcc1.a
quit
```

Another way to provide the functions you need in `libgcc1.a' is to define the appropriate `perform_...` macros for those functions. If these definitions do not use the C arithmetic operators that they are meant to implement, you should be able to compile them with the cross-compiler you are building. (If these definitions already exist for your target file, then you are all set.)

To build `libgcc1.a' using the perform macros, use `LIBGCC1=libgcc1.a OLDCC=./xgcc' when building the compiler. Otherwise, you should place your replacement library under the name `libgcc1.a' in the directory in which you will build the cross-compiler, before you run `make`.

Cross-Compilers and Header Files

If you are cross-compiling a standalone program or a program for an embedded system, then you may not need any header files except the few that are part of GNU CC (and those of your program). However, if you intend to link your program with a standard C library such as `libc.a', then you probably need to compile with the header files that go with the library you use.

The GNU C compiler does not come with these files, because (1) they are system-specific, and (2) they belong in a C library, not in a compiler.

If the GNU C library supports your target machine, then you can get the header files from there (assuming you actually use the GNU library when you link your program).

If your target machine comes with a C compiler, it probably comes with suitable header files also. If you make these files accessible from the host machine, the cross-compiler can use them also.

Otherwise, you're on your own in finding header files to use when cross-compiling.

When you have found suitable header files, put them in `/usr/local/target/include', before building the cross compiler. Then installation will run fixincludes properly and install the corrected versions of the header files where the compiler will use them.

Provide the header files before you build the cross-compiler, because the build stage actually runs the cross-compiler to produce parts of `libgcc.a'. (These are the parts that *can* be compiled with GNU CC.) Some of them need suitable header files.

Here's an example showing how to copy the header files from a target machine. On the target machine, do this:

```
(cd /usr/include; tar cf - .) > tarfile
```

Then, on the host machine, do this:

```
ftp target-machine
lcd /usr/local/target/include
get tarfile
quit
tar xf tarfile
```

Actually Building the Cross-Compiler

Now you can proceed just as for compiling a single-machine compiler through the step of building stage 1. If you have not provided some sort of `libgcc1.a', then compilation will give up at the point where it needs that file, printing a suitable error message. If you do provide `libgcc1.a', then building the compiler will automatically compile and link a test program called `libgcc1-test'; if you get errors in the linking, it means that not all of the necessary routines in `libgcc1.a' are available.

You must provide the header file `float.h'. One way to do this is to compile `enquire' and run it on your target machine. The job of `enquire' is to run on the target machine and figure out by experiment the nature of its floating point representation. `enquire' records its findings in the header file `float.h'. If you can't produce this file by running `enquire' on the target machine, then you will need to come up with a suitable `float.h' in some other way (or else, avoid using it in your programs).

Do not try to build stage 2 for a cross-compiler. It doesn't work to rebuild GNU CC as a cross-compiler using the cross-compiler, because that would produce a program that runs on the target machine, not on the host. For example, if you compile a 386-to-68030 cross-compiler with itself, the result will not be right either for the 386 (because it was compiled into 68030 code) or for the 68030 (because it was configured for a 386 as the host). If you want to compile GNU CC into 68030 code, whether you compile it on a 68030 or with a cross-compiler on a 386, you must specify a 68030 as the host when you configure it.

To install the cross-compiler, use `make install', as usual.

Installing GNU CC on the Sun

On Solaris (version 2.1), do not use the linker or other tools in `/usr/ucb' to build GNU CC. Use `/usr/ccs/bin`.

Make sure the environment variable `FLOAT_OPTION` is not set when you compile `libgcc.a'. If this option were set to `f68881` when `libgcc.a' is compiled, the resulting code would demand to be linked with a special startup file and would not link properly without special pains.

There is a bug in `alloca` in certain versions of the Sun library. To avoid this bug, install the binaries of GNU CC that were compiled by GNU CC. They use `alloca` as a built-in function and never the one in the library.

Some versions of the Sun compiler crash when compiling GNU CC. The problem is a segmentation fault in cpp. This problem seems to be due to the bulk of data in the environment variables. You may be able to avoid it by using the following command to compile GNU CC with Sun CC:

```
make CC="TERMCAP=x OBJS=x LIBFUNCS=x STAGESTUFF=x cc"
```

Installing GNU CC on VMS

The VMS version of GNU CC is distributed in a backup saveset containing both source code and precompiled binaries.

To install the `gcc' command so you can use the compiler easily, in the same manner as you use the VMS C compiler, you must install the VMS CLD file for GNU CC as follows:

1. Define the VMS logical names `GNU_CC' and `GNU_CC_INCLUDE' to point to the directories where the GNU CC executables (`gcc-cpp.exe', `gcc-cc1.exe', etc.) and the C include files are kept respectively. This should be done with the commands:

   ```
   $ assign /system /translation=concealed -
     disk:[gcc.] gnu_cc
   $ assign /system /translation=concealed -
     disk:[gcc.include.] gnu_cc_include
   ```

   with the appropriate disk and directory names. These commands can be placed in your system startup file so they will be executed whenever the machine is rebooted. You may, if you choose, do this via the `GCC_INSTALL.COM' script in the `[GCC]' directory.
2. Install the `GCC' command with the command line:


```
$ set command /table=sys$common:[syslib]dcltables -
  /output=sys$common:[syslib]dcltables gnu_cc:[000000]gcc
$ install replace sys$common:[syslib]dcltables
```

3. To install the help file, do the following:

   ```
   $ library/help sys$library:helplib.hlb gcc.hlp
   ```

   Now you can invoke the compiler with a command like `gcc /verbose file.c', which is equivalent to the command `gcc -v -c file.c' in Unix.

If you wish to use GNU C++ you must first install GNU CC, and then perform the following steps:

1. Define the VMS logical name `GNU_GXX_INCLUDE' to point to the directory where the preprocessor will search for the C++ header files. This can be done with the command:

   ```
   $ assign /system /translation=concealed -
     disk:[gcc.gxx_include.] gnu_gxx_include
   ```

   with the appropriate disk and directory name. If you are going to be using libg++, this is where the libg++ install procedure will install the libg++ header files.

2. Obtain the file `gcc-cc1plus.exe', and place this in the same directory that `gcc-cc1.exe' is kept.

   The GNU C++ compiler can be invoked with a command like `gcc /plus /verbose file.cc', which is equivalent to the command `g++ -v -c file.cc' in Unix.

We try to put corresponding binaries and sources on the VMS distribution tape. But sometimes the binaries will be from an older version than the sources, because we don't always have time to update them. (Use the `/version' option to determine the version number of the binaries and compare it with the source file `version.c' to tell whether this is so.) In this case, you should use the binaries you get to recompile the sources. If you must recompile, here is how:

1. Execute the command procedure `vmsconfig.com' to set up the files `tm.h', `config.h', `aux-output.c', and `md.', and to create files `tconfig.h' and `hconfig.h'. This procedure also creates several linker option files used by `make-cc1.com' and a data file used by `make-l2.com'.

   ```
   $ @vmsconfig.com
   ```

2. Setup the logical names and command tables as defined above. In addition, define the VMS logical name `GNU_BISON' to point at the to the directories where the Bison executable is kept. This should be done with the command:

   ```
   $ assign /system /translation=concealed -
     disk:[bison.] gnu_bison
   ```

   You may, if you choose, use the `INSTALL_BISON.COM' script in the `[BISON]' directory.

3. Install the `BISON' command with the command line:

   ```
   $ set command /table=sys$common:[syslib]dcltables -
     /output=sys$common:[syslib]dcltables -
   ```


```
  gnu_bison:[000000]bison
$ install replace sys$common:[syslib]dcltables
```

4. Type `@make-gcc' to recompile everything (alternatively, submit the file `make-gcc.com' to a batch queue). If you wish to build the GNU C++ compiler as well as the GNU CC compiler, you must first edit `make-gcc.com' and follow the instructions that appear in the comments.
5. In order to use GCC, you need a library of functions which GCC compiled code will call to perform certain tasks, and these functions are defined in the file `libgcc2.c'. To compile this you should use the command procedure `make-l2.com', which will generate the library `libgcc2.olb'. `libgcc2.olb' should be built using the compiler built from the same distribution that `libgcc2.c' came from, and `make-gcc.com' will automatically do all of this for you.

   To install the library, use the following commands:

   ```
   $ library gnu_cc:[000000]gcclib/delete=(new,eprintf)
   $ library gnu_cc:[000000]gcclib/delete=L_*
   $ library libgcc2/extract=*/output=libgcc2.obj
   $ library gnu_cc:[000000]gcclib libgcc2.obj
   ```

   The first command simply removes old modules that will be replaced with modules from `libgcc2' under different module names. The modules `new` and `eprintf` may not actually be present in your `gcclib.olb'---if the VMS librarian complains about those modules not being present, simply ignore the message and continue on with the next command. The second command removes the modules that came from the previous version of the library `libgcc2.c'.

   Whenever you update the compiler on your system, you should also update the library with the above procedure.
6. You may wish to build GCC in such a way that no files are written to the directory where the source files reside. An example would be the when the source files are on a read-only disk. In these cases, execute the following DCL commands (substituting your actual path names):

   ```
   $ assign dua0:[gcc.build_dir.]/translation=concealed, -
            dua1:[gcc.source_dir.]/translation=concealed  gcc_build
   $ set default gcc_build:[000000]
   ```

   where the directory `dua1:[gcc.source_dir]' contains the source code, and the directory `dua0:[gcc.build_dir]' is meant to contain all of the generated object files and executables. Once you have done this, you can proceed building GCC as described above. (Keep in mind that `gcc_build' is a rooted logical name, and thus the device names in each element of the search list must be an actual physical device name rather than another rooted logical name).
7. **If you are building GNU CC with a previous version of GNU CC, you also should check to see that you have the newest version of the assembler**. In particular, GNU CC version 2 treats global constant variables slightly differently from GNU CC version 1, and GAS version 1.38.1 does not have the patches required to work with GCC version 2. If you use GAS 1.38.1, then `extern const` variables will not have the read-only bit set, and the linker will generate warning messages about mismatched psect attributes for these variables. These warning messages are merely a nuisance, and can safely be ignored.

   If you are compiling with a version of GNU CC older than 1.33, specify `/DEFINE=("inline=")' as an option in all the compilations. This requires editing all the `gcc` commands in `make-cc1.com'. (The older versions had problems supporting `inline`.) Once you have a working 1.33 or newer GNU CC, you

can change this file back.

8. If you want to build GNU CC with the VAX C compiler, you will need to make minor changes in `make-cccp.com' and `make-cc1.com' to choose alternate definitions of CC, CFLAGS, and LIBS. See comments in those files. However, you must also have a working version of the GNU assembler (GNU as, aka GAS) as it is used as the back-end for GNU CC to produce binary object modules and is not included in the GNU CC sources. GAS is also needed to compile `libgcc2' in order to build `gcclib' (see above); `make-l2.com' expects to be able to find it operational in `gnu_cc:[000000]gnu-as.exe'.

   To use GNU CC on VMS, you need the VMS driver programs `gcc.exe', `gcc.com', and `gcc.cld'. They are distributed with the VMS binaries (`gcc-vms') rather than the GNU CC sources. GAS is also included in `gcc-vms', as is Bison.

   Once you have successfully built GNU CC with VAX C, you should use the resulting compiler to rebuild itself. Before doing this, be sure to restore the CC, CFLAGS, and LIBS definitions in `make-cccp.com' and `make-cc1.com'. The second generation compiler will be able to take advantage of many optimizations that must be suppressed when building with other compilers.

Under previous versions of GNU CC, the generated code would occasionally give strange results when linked with the sharable `VAXCRTL' library. Now this should work.

Even with this version, however, GNU CC itself should not be linked with the sharable `VAXCRTL'. The version of `qsort' in `VAXCRTL' has a bug (known to be present in VMS versions V4.6 through V5.5) which causes the compiler to fail.

The executables are generated by `make-cc1.com' and `make-cccp.com' use the object library version of `VAXCRTL' in order to make use of the `qsort' routine in `gcclib.olb'. If you wish to link the compiler executables with the shareable image version of `VAXCRTL', you should edit the file `tm.h' (created by `vmsconfig.com') to define the macro `QSORT_WORKAROUND'.

`QSORT_WORKAROUND' is always defined when GNU CC is compiled with VAX C, to avoid a problem in case `gcclib.olb' is not yet available.

collect2

Many target systems do not have support in the assembler and linker for "constructors"---initialization functions to be called before the official "start" of `main'. On such systems, GNU CC uses a utility called `collect2' to arrange to call these functions at start time.

The program `collect2' works by linking the program once and looking through the linker output file for symbols with particular names indicating they are constructor functions. If it finds any, it creates a new temporary `.c' file containing a table of them, compiles it, and links the program a second time including that file.

The actual calls to the constructors are carried out by a subroutine called `__main', which is called (automatically) at the beginning of the body of `main' (provided `main' was compiled with GNU CC). Calling `__main' is necessary, even when compiling C code, to allow linking C and C++ object code together. (If you use `-nostdlib', you get an unresolved reference to `__main', since it's defined in the standard GCC library. Include `-lgcc' at the end of your compiler command line to resolve this reference.)

The program `collect2' is installed as `ld' in the directory where the passes of the compiler are installed. When

`collect2` needs to find the *real* `ld`, it tries the following file names:

- `` `real-ld' `` in the directories listed in the compiler's search directories.
- `` `real-ld' `` in the directories listed in the environment variable `PATH`.
- The file specified in the `REAL_LD_FILE_NAME` configuration macro, if specified.
- `` `ld' `` in the compiler's search directories, except that `collect2` will not execute itself recursively.
- `` `ld' `` in `PATH`.

"The compiler's search directories" means all the directories where `gcc` searches for passes of the compiler. This includes directories that you specify with `-B'.

Cross-compilers search a little differently:

- `` `real-ld' `` in the compiler's search directories.
- `` `target-real-ld' `` in `PATH`.
- The file specified in the `REAL_LD_FILE_NAME` configuration macro, if specified.
- `` `ld' `` in the compiler's search directories.
- `` `target-ld' `` in `PATH`.

`collect2` explicitly avoids running `ld` using the file name under which `collect2` itself was invoked. In fact, it remembers up a list of such names--in case one copy of `collect2` finds another copy (or version) of `collect2` installed as `ld` in a second place in the search path.

`collect2` searches for the utilities `nm` and `strip` using the same algorithm as above for `ld`.

Standard Header File Directories

`GCC_INCLUDE_DIR` means the same thing for native and cross. It is where GNU CC stores its private include files, and also where GNU CC stores the fixed include files. A cross compiled GNU CC runs `fixincludes` on the header files in `` `$(tooldir)/include' ``. (If the cross compilation header files need to be fixed, they must be installed before GNU CC is built. If the cross compilation header files are already suitable for ANSI C and GNU CC, nothing special need be done).

`GPLUS_INCLUDE_DIR` means the same thing for native and cross. It is where `g++` looks first for header files. `libg++` installs only target independent header files in that directory.

`LOCAL_INCLUDE_DIR` is used only for a native compiler. It is normally `` `/usr/local/include' ``. GNU CC searches this directory so that users can install header files in `` `/usr/local/include' ``.

`CROSS_INCLUDE_DIR` is used only for a cross compiler. GNU CC doesn't install anything there.

`TOOL_INCLUDE_DIR` is used for both native and cross compilers. It is the place for other packages to install header files that GNU CC will use. For a cross-compiler, this is the equivalent of `` `/usr/include' ``. When you build a cross-compiler, `fixincludes` processes any header files in this directory.

Go to the previous, next section.

Go to the previous, next section.

Extensions to the C Language Family

GNU C provides several language features not found in ANSI standard C. (The `-pedantic' option directs GNU CC to print a warning message if any of these features is used.) To test for the availability of these features in conditional compilation, check for a predefined macro `__GNUC__`, which is always defined under GNU CC.

These extensions are available in C and Objective C. Most of them are also available in C++. See section Extensions to the C++ Language, for extensions that apply *only* to C++.

Statements and Declarations in Expressions

A compound statement enclosed in parentheses may appear as an expression in GNU C. This allows you to use loops, switches, and local variables within an expression.

Recall that a compound statement is a sequence of statements surrounded by braces; in this construct, parentheses go around the braces. For example:

```
({ int y = foo (); int z;
   if (y > 0) z = y;
   else z = - y;
   z; })
```

is a valid (though slightly more complex than necessary) expression for the absolute value of `foo ()`.

The last thing in the compound statement should be an expression followed by a semicolon; the value of this subexpression serves as the value of the entire construct. (If you use some other kind of statement last within the braces, the construct has type `void`, and thus effectively no value.)

This feature is especially useful in making macro definitions "safe" (so that they evaluate each operand exactly once). For example, the "maximum" function is commonly defined as a macro in standard C as follows:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

But this definition computes either a or b twice, with bad results if the operand has side effects. In GNU C, if you know the type of the operands (here let's assume `int`), you can define the macro safely as follows:

```
#define maxint(a,b) \
  ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

Embedded statements are not allowed in constant expressions, such as the value of an enumeration

constant, the width of a bit field, or the initial value of a static variable.

If you don't know the type of the operand, you can still do this, but you must use `typeof` (see section Referring to a Type with `typeof`) or type naming (see section Naming an Expression's Type).

Locally Declared Labels

Each statement expression is a scope in which local labels can be declared. A local label is simply an identifier; you can jump to it with an ordinary `goto` statement, but only from within the statement expression it belongs to.

A local label declaration looks like this:

```
__label__ label;
```

or

```
__label__ label1, label2, ...;
```

Local label declarations must come at the beginning of the statement expression, right after the `({', before any ordinary declarations.

The label declaration defines the label *name*, but does not define the label itself. You must do this in the usual way, with `label:`, within the statements of the statement expression.

The local label feature is useful because statement expressions are often used in macros. If the macro contains nested loops, a `goto` can be useful for breaking out of them. However, an ordinary label whose scope is the whole function cannot be used: if the macro can be expanded several times in one function, the label will be multiply defined in that function. A local label avoids this problem. For example:

```
#define SEARCH(array, target)                     \
({                                                \
  __label__ found;                                \
  typeof (target) _SEARCH_target = (target);      \
  typeof (*(array)) *_SEARCH_array = (array);     \
  int i, j;                                       \
  int value;                                      \
  for (i = 0; i < max; i++)                       \
    for (j = 0; j < max; j++)                     \
      if (_SEARCH_array[i][j] == _SEARCH_target)  \
        { value = i; goto found; }                \
  value = -1;                                     \
 found:                                           \
  value;                                          \
})
```

Labels as Values

You can get the address of a label defined in the current function (or a containing function) with the unary operator `&&'. The value has type `void *`. This value is a constant and can be used wherever a constant of that type is valid. For example:

```
void *ptr;
...
ptr = &&foo;
```

To use these values, you need to be able to jump to one. This is done with the computed goto statement(2), `goto *exp;`. For example,

```
goto *ptr;
```

Any expression of type `void *` is allowed.

One way of using these constants is in initializing a static array that will serve as a jump table:

```
static void *array[] = { &&foo, &&bar, &&hack };
```

Then you can select a label with indexing, like this:

```
goto *array[i];
```

Note that this does not check whether the subscript is in bounds--array indexing in C never does that.

Such an array of label values serves a purpose much like that of the `switch` statement. The `switch` statement is cleaner, so use that rather than an array unless the problem does not fit a `switch` statement very well.

Another use of label values is in an interpreter for threaded code. The labels within the interpreter function can be stored in the threaded code for super-fast dispatching.

You can use this mechanism to jump to code in a different function. If you do that, totally unpredictable things will happen. The best way to avoid this is to store the label address only in automatic variables and never pass it as an argument.

Nested Functions

A nested function is a function defined inside another function. (Nested functions are not supported for GNU C++.) The nested function's name is local to the block where it is defined. For example, here we define a nested function named `square`, and call it twice:

```
foo (double a, double b)
{
```


```
  double square (double z) { return z * z; }

  return square (a) + square (b);
}
```

The nested function can access all the variables of the containing function that are visible at the point of its definition. This is called lexical scoping. For example, here we show a nested function which uses an inherited variable named `offset`:

```
bar (int *array, int offset, int size)
{
  int access (int *array, int index)
    { return array[index + offset]; }
  int i;
  ...
  for (i = 0; i < size; i++)
    ... access (array, i) ...
}
```

Nested function definitions are permitted within functions in the places where variable definitions are allowed; that is, in any block, before the first statement in the block.

It is possible to call the nested function from outside the scope of its name by storing its address or passing the address to another function:

```
hack (int *array, int size)
{
  void store (int index, int value)
    { array[index] = value; }

  intermediate (store, size);
}
```

Here, the function `intermediate` receives the address of `store` as an argument. If `intermediate` calls `store`, the arguments given to `store` are used to store into `array`. But this technique works only so long as the containing function (`hack`, in this example) does not exit.

If you try to call the nested function through its address after the containing function has exited, all hell will break loose. If you try to call it after a containing scope level has exited, and if it refers to some of the variables that are no longer in scope, you may be lucky, but it's not wise to take the risk. If, however, the nested function does not refer to anything that has gone out of scope, you should be safe.

GNU CC implements taking the address of a nested function using a technique called trampolines. A paper describing them is available from `maya.idiap.ch' in directory `pub/tmb', file `usenix88-lexic.ps.Z'.

A nested function can jump to a label inherited from a containing function, provided the label was

explicitly declared in the containing function (see section Locally Declared Labels). Such a jump returns instantly to the containing function, exiting the nested function which did the `goto` and any intermediate functions as well. Here is an example:

```
bar (int *array, int offset, int size)
{
  __label__ failure;
  int access (int *array, int index)
    {
      if (index > size)
        goto failure;
      return array[index + offset];
    }
  int i;
  ...
  for (i = 0; i < size; i++)
    ... access (array, i) ...
  ...
  return 0;

 /* Control comes here from access
    if it detects an error.  */
 failure:
  return -1;
}
```

A nested function always has internal linkage. Declaring one with `extern` is erroneous. If you need to declare the nested function before its definition, use `auto` (which is otherwise meaningless for function declarations).

```
bar (int *array, int offset, int size)
{
  __label__ failure;
  auto int access (int *, int);
  ...
  int access (int *array, int index)
    {
      if (index > size)
        goto failure;
      return array[index + offset];
    }
  ...
}
```

Constructing Function Calls

Using the built-in functions described below, you can record the arguments a function received, and call another function with the same arguments, without knowing the number or types of the arguments.

You can also record the return value of that function call, and later return that value, without knowing what data type the function tried to return (as long as your caller expects that data type).

`__builtin_apply_args ()`

This built-in function returns a pointer of type `void *` to data describing how to perform a call with the same arguments as were passed to the current function.

The function saves the arg pointer register, structure value address, and all registers that might be used to pass arguments to a function into a block of memory allocated on the stack. Then it returns the address of that block.

`__builtin_apply (function, arguments, size)`

This built-in function invokes function (type `void (*)()`) with a copy of the parameters described by arguments (type `void *`) and size (type `int`).

The value of arguments should be the value returned by `__builtin_apply_args`. The argument size specifies the size of the stack argument data, in bytes.

This function returns a pointer of type `void *` to data describing how to return whatever value was returned by function. The data is saved in a block of memory allocated on the stack.

It is not always simple to compute the proper value for size. The value is used by `__builtin_apply` to compute the amount of data that should be pushed on the stack and copied from the incoming argument area.

`__builtin_return (result)`

This built-in function returns the value described by result from the containing function. You should specify, for result, a value returned by `__builtin_apply`.

Naming an Expression's Type

You can give a name to the type of an expression using a `typedef` declaration with an initializer. Here is how to define name as a type name for the type of exp:

```
typedef name = exp;
```

This is useful in conjunction with the statements-within-expressions feature. Here is how the two together can be used to define a safe "maximum" macro that operates on any arithmetic type:

```
#define max(a,b) \
  ({typedef _ta = (a), _tb = (b);  \
    _ta _a = (a); _tb _b = (b);     \
```


```
_a > _b ? _a : _b; })
```

The reason for using names that start with underscores for the local variables is to avoid conflicts with variable names that occur within the expressions that are substituted for `a` and `b`. Eventually we hope to design a new form of declaration syntax that allows you to declare variables whose scopes start only after their initializers; this will be a more reliable way to prevent such conflicts.

Referring to a Type with `typeof`

Another way to refer to the type of an expression is with `typeof`. The syntax of using of this keyword looks like `sizeof`, but the construct acts semantically like a type name defined with `typedef`.

There are two ways of writing the argument to `typeof`: with an expression or with a type. Here is an example with an expression:

```
typeof (x[0](1))
```

This assumes that `x` is an array of functions; the type described is that of the values of the functions.

Here is an example with a typename as the argument:

```
typeof (int *)
```

Here the type described is that of pointers to `int`.

If you are writing a header file that must work when included in ANSI C programs, write `__typeof__` instead of `typeof`. See section Alternate Keywords.

A `typeof`-construct can be used anywhere a typedef name could be used. For example, you can use it in a declaration, in a cast, or inside of `sizeof` or `typeof`.

- This declares `y` with the type of what `x` points to.

  ```
  typeof (*x) y;
  ```

- This declares `y` as an array of such values.

  ```
  typeof (*x) y[4];
  ```

- This declares `y` as an array of pointers to characters:

  ```
  typeof (typeof (char *)[4]) y;
  ```

  It is equivalent to the following traditional C declaration:

  ```
  char *y[4];
  ```

  To see the meaning of the declaration using `typeof`, and why it might be a useful way to write, let's rewrite it with these macros:


```
#define pointer(T)  typeof(T *)
#define array(T, N) typeof(T [N])
```

Now the declaration can be rewritten this way:

```
array (pointer (char), 4) y;
```

Thus, `array (pointer (char), 4)` is the type of arrays of 4 pointers to `char`.

Generalized Lvalues

Compound expressions, conditional expressions and casts are allowed as lvalues provided their operands are lvalues. This means that you can take their addresses or store values into them.

Standard C++ allows compound expressions and conditional expressions as lvalues, and permits casts to reference type, so use of this extension is deprecated for C++ code.

For example, a compound expression can be assigned, provided the last expression in the sequence is an lvalue. These two expressions are equivalent:

```
(a, b) += 5
a, (b += 5)
```

Similarly, the address of the compound expression can be taken. These two expressions are equivalent:

```
&(a, b)
a, &b
```

A conditional expression is a valid lvalue if its type is not void and the true and false branches are both valid lvalues. For example, these two expressions are equivalent:

```
(a ? b : c) = 5
(a ? b = 5 : (c = 5))
```

A cast is a valid lvalue if its operand is an lvalue. A simple assignment whose left-hand side is a cast works by converting the right-hand side first to the specified type, then to the type of the inner left-hand side expression. After this is stored, the value is converted back to the specified type to become the value of the assignment. Thus, if `a` has type `char *`, the following two expressions are equivalent:

```
(int)a = 5
(int)(a = (char *)(int)5)
```

An assignment-with-arithmetic operation such as `+=' applied to a cast performs the arithmetic using the type resulting from the cast, and then continues as in the previous case. Therefore, these two expressions are equivalent:


```
(int)a += 5
(int)(a = (char *)(int) ((int)a + 5))
```

You cannot take the address of an lvalue cast, because the use of its address would not work out coherently. Suppose that `&(int)f` were permitted, where `f` has type `float`. Then the following statement would try to store an integer bit-pattern where a floating point number belongs:

```
*&(int)f = 1;
```

This is quite different from what `(int)f = 1` would do--that would convert 1 to floating point and store it. Rather than cause this inconsistency, we think it is better to prohibit use of `&' on a cast.

If you really do want an `int *` pointer with the address of `f`, you can simply write `(int *)&f`.

Conditionals with Omitted Operands

The middle operand in a conditional expression may be omitted. Then if the first operand is nonzero, its value is the value of the conditional expression.

Therefore, the expression

```
x ? : y
```

has the value of `x` if that is nonzero; otherwise, the value of `y`.

This example is perfectly equivalent to

```
x ? x : y
```

In this simple case, the ability to omit the middle operand is not especially useful. When it becomes useful is when the first operand does, or may (if it is a macro argument), contain a side effect. Then repeating the operand in the middle would perform the side effect twice. Omitting the middle operand uses the value already computed without the undesirable effects of recomputing it.

Double-Word Integers

GNU C supports data types for integers that are twice as long as `long int`. Simply write `long long int` for a signed integer, or `unsigned long long int` for an unsigned integer. To make an integer constant of type `long long int`, add the suffix `LL` to the integer. To make an integer constant of type `unsigned long long int`, add the suffix `ULL` to the integer.

You can use these types in arithmetic like any other integer types. Addition, subtraction, and bitwise boolean operations on these types are open-coded on all types of machines. Multiplication is open-coded if the machine supports fullword-to-doubleword a widening multiply instruction. Division and shifts are open-coded only on machines that provide special support. The operations that are not open-coded use

special library routines that come with GNU CC.

There may be pitfalls when you use `long long` types for function arguments, unless you declare function prototypes. If a function expects type `int` for its argument, and you pass a value of type `long long int`, confusion will result because the caller and the subroutine will disagree about the number of bytes for the argument. Likewise, if the function expects `long long int` and you pass `int`. The best way to avoid such problems is to use prototypes.

Complex Numbers

GNU C supports complex data types. You can declare both complex integer types and complex floating types, using the keyword `__complex__`.

For example, `__complex__ double x;' declares `x` as a variable whose real part and imaginary part are both of type `double`. `__complex__ short int y;' declares `y` to have real and imaginary parts of type `short int`; this is not likely to be useful, but it shows that the set of complex types is complete.

To write a constant with a complex data type, use the suffix `i' or `j' (either one; they are equivalent). For example, `2.5fi` has type `__complex__ float` and `3i` has type `__complex__ int`. Such a constant always has a pure imaginary value, but you can form any complex value you like by adding one to a real constant.

To extract the real part of a complex-valued expression exp, write `__real__ exp`. Likewise, use `__imag__` to extract the imaginary part.

The operator `~' performs complex conjugation when used on a value with a complex type.

GNU CC can allocate complex automatic variables in a noncontiguous fashion; it's even possible for the real part to be in a register while the imaginary part is on the stack (or vice-versa). None of the supported debugging info formats has a way to represent noncontiguous allocation like this, so GNU CC describes a noncontiguous complex variable as if it were two separate variables of noncomplex type. If the variable's actual name is `foo`, the two fictitious variables are named `foo$real` and `foo$imag`. You can examine and set these two fictitious variables with your debugger.

A future version of GDB will know how to recognize such pairs and treat them as a single variable with a complex type.

Arrays of Length Zero

Zero-length arrays are allowed in GNU C. They are very useful as the last element of a structure which is really a header for a variable-length object:

```
struct line {
  int length;
  char contents[0];
};
```


```
{
  struct line *thisline = (struct line *)
    malloc (sizeof (struct line) + this_length);
  thisline->length = this_length;
}
```

In standard C, you would have to give `contents` a length of 1, which means either you waste space or complicate the argument to `malloc`.

Arrays of Variable Length

Variable-length automatic arrays are allowed in GNU C. These arrays are declared like any other automatic arrays, but with a length that is not a constant expression. The storage is allocated at the point of declaration and deallocated when the brace-level is exited. For example:

```
FILE *
concat_fopen (char *s1, char *s2, char *mode)
{
  char str[strlen (s1) + strlen (s2) + 1];
  strcpy (str, s1);
  strcat (str, s2);
  return fopen (str, mode);
}
```

Jumping or breaking out of the scope of the array name deallocates the storage. Jumping into the scope is not allowed; you get an error message for it.

You can use the function `alloca` to get an effect much like variable-length arrays. The function `alloca` is available in many other C implementations (but not in all). On the other hand, variable-length arrays are more elegant.

There are other differences between these two methods. Space allocated with `alloca` exists until the containing *function* returns. The space for a variable-length array is deallocated as soon as the array name's scope ends. (If you use both variable-length arrays and `alloca` in the same function, deallocation of a variable-length array will also deallocate anything more recently allocated with `alloca`.)

You can also use variable-length arrays as arguments to functions:

```
struct entry
tester (int len, char data[len][len])
{
  ...
}
```

The length of an array is computed once when the storage is allocated and is remembered for the sco of the array in case you access it with `sizeof`.

If you want to pass the array first and the length afterward, you can use a forward declaration in the parameter list--another GNU extension.

```
struct entry
tester (int len; char data[len][len], int len)
{
  ...
}
```

The `int len' before the semicolon is a parameter forward declaration, and it serves the purpose of making the name `len` known when the declaration of `data` is parsed.

You can write any number of such parameter forward declarations in the parameter list. They can be separated by commas or semicolons, but the last one must end with a semicolon, which is followed by the "real" parameter declarations. Each forward declaration must match a "real" declaration in parameter name and data type.

Macros with Variable Numbers of Arguments

In GNU C, a macro can accept a variable number of arguments, much as a function can. The syntax for defining the macro looks much like that used for a function. Here is an example:

```
#define eprintf(format, args...)  \
 fprintf (stderr, format , ## args)
```

Here `args` is a rest argument: it takes in zero or more arguments, as many as the call contains. All of them plus the commas between them form the value of `args`, which is substituted into the macro body where `args` is used. Thus, we have this expansion:

```
eprintf ("%s:%d: ", input_file_name, line_number)
==>
fprintf (stderr, "%s:%d: " , input_file_name, line_number)
```

Note that the comma after the string constant comes from the definition of `eprintf`, whereas the last comma comes from the value of `args`.

The reason for using `##' is to handle the case when `args` matches no arguments at all. In this case, `args` has an empty value. In this case, the second comma in the definition becomes an embarrassment: if it got through to the expansion of the macro, we would get something like this:

```
fprintf (stderr, "success!\n" , )
```

which is invalid C syntax. `##' gets rid of the comma, so we get the following instead:


```
fprintf (stderr, "success!\n")
```

This is a special feature of the GNU C preprocessor: `##' before a rest argument that is empty discards the preceding sequence of non-whitespace characters from the macro definition. (If another macro argument precedes, none of it is discarded.)

It might be better to discard the last preprocessor token instead of the last preceding sequence of non-whitespace characters; in fact, we may someday change this feature to do so. We advise you to write the macro definition so that the preceding sequence of non-whitespace characters is just a single token, so that the meaning will not change if we change the definition of this feature.

Non-Lvalue Arrays May Have Subscripts

Subscripting is allowed on arrays that are not lvalues, even though the unary `&' operator is not. For example, this is valid in GNU C though not valid in other C dialects:

```
struct foo {int a[4];};

struct foo f();

bar (int index)
{
  return f().a[index];
}
```

Arithmetic on `void`- and Function-Pointers

In GNU C, addition and subtraction operations are supported on pointers to `void` and on pointers to functions. This is done by treating the size of a `void` or of a function as 1.

A consequence of this is that `sizeof` is also allowed on `void` and on function types, and returns 1.

The option `-Wpointer-arith' requests a warning if these extensions are used.

Non-Constant Initializers

As in standard C++, the elements of an aggregate initializer for an automatic variable are not required to be constant expressions in GNU C. Here is an example of an initializer with run-time varying elements:

```
foo (float f, float g)
{
  float beat_freqs[2] = { f-g, f+g };
  ...
```


```
}
```

Constructor Expressions

GNU C supports constructor expressions. A constructor looks like a cast containing an initializer. Its value is an object of the type specified in the cast, containing the elements specified in the initializer.

Usually, the specified type is a structure. Assume that `struct foo` and `structure` are declared as shown:

```
struct foo {int a; char b[2];} structure;
```

Here is an example of constructing a `struct foo` with a constructor:

```
structure = ((struct foo) {x + y, 'a', 0});
```

This is equivalent to writing the following:

```
{
  struct foo temp = {x + y, 'a', 0};
  structure = temp;
}
```

You can also construct an array. If all the elements of the constructor are (made up of) simple constant expressions, suitable for use in initializers, then the constructor is an lvalue and can be coerced to a pointer to its first element, as shown here:

```
char **foo = (char *[]) { "x", "y", "z" };
```

Array constructors whose elements are not simple constants are not very useful, because the constructor is not an lvalue. There are only two valid ways to use it: to subscript it, or initialize an array variable with it. The former is probably slower than a `switch` statement, while the latter does the same thing an ordinary C initializer would do. Here is an example of subscripting an array constructor:

```
output = ((int[]) { 2, x, 28 }) [input];
```

Constructor expressions for scalar types and union types are is also allowed, but then the constructor expression is equivalent to a cast.

Labeled Elements in Initializers

Standard C requires the elements of an initializer to appear in a fixed order, the same as the order of the elements in the array or structure being initialized.

In GNU C you can give the elements in any order, specifying the array indices or structure field names

they apply to. This extension is not implemented in GNU C++.

To specify an array index, write `[index]' or `[index] =' before the element value. For example,

```
int a[6] = { [4] 29, [2] = 15 };
```

is equivalent to

```
int a[6] = { 0, 0, 15, 0, 29, 0 };
```

The index values must be constant expressions, even if the array being initialized is automatic.

To initialize a range of elements to the same value, write `[first ... last] = value'. For example,

```
int widths[] = { [0 ... 9] = 1, [10 ... 99] = 2, [100] = 3 };
```

Note that the length of the array is the highest value specified plus one.

In a structure initializer, specify the name of a field to initialize with `fieldname:' before the element value. For example, given the following structure,

```
struct point { int x, y; };
```

the following initialization

```
struct point p = { y: yvalue, x: xvalue };
```

is equivalent to

```
struct point p = { xvalue, yvalue };
```

Another syntax which has the same meaning is `.fieldname =', as shown here:

```
struct point p = { .y = yvalue, .x = xvalue };
```

You can also use an element label (with either the colon syntax or the period-equal syntax) when initializing a union, to specify which element of the union should be used. For example,

```
union foo { int i; double d; };

union foo f = { d: 4 };
```

will convert 4 to a `double` to store it in the union using the second element. By contrast, casting 4 to type `union foo` would store it into the union as the integer `i`, since it is an integer. (See section Cast to a Union Type.)

You can combine this technique of naming elements with ordinary C initialization of successive elements. Each initializer element that does not have a label applies to the next consecutive element of

the array or structure. For example,

```
int a[6] = { [1] = v1, v2, [4] = v4 };
```

is equivalent to

```
int a[6] = { 0, v1, v2, 0, v4, 0 };
```

Labeling the elements of an array initializer is especially useful when the indices are characters or belong to an `enum` type. For example:

```
int whitespace[256]
  = { [' '] = 1, ['\t'] = 1, ['\h'] = 1,
      ['\f'] = 1, ['\n'] = 1, ['\r'] = 1 };
```

Case Ranges

You can specify a range of consecutive values in a single `case` label, like this:

```
case low ... high:
```

This has the same effect as the proper number of individual `case` labels, one for each integer value from low to high, inclusive.

This feature is especially useful for ranges of ASCII character codes:

```
case 'A' ... 'Z':
```

Be careful: Write spaces around the `...`, for otherwise it may be parsed wrong when you use it with integer values. For example, write this:

```
case 1 ... 5:
```

rather than this:

```
case 1...5:
```

Cast to a Union Type

A cast to union type is similar to other casts, except that the type specified is a union type. You can specify the type either with `union tag` or with a typedef name. A cast to union is actually a constructor though, not a cast, and hence does not yield an lvalue like normal casts. (See section Constructor Expressions.)

The types that may be cast to the union type are those of the members of the union. Thus, given the

following union and variables:

```
union foo { int i; double d; };
int x;
double y;
```

both `x` and `y` can be cast to type `union` foo.

Using the cast as the right-hand side of an assignment to a variable of union type is equivalent to storing in a member of the union:

```
union foo u;
...
u = (union foo) x  ==  u.i = x
u = (union foo) y  ==  u.d = y
```

You can also use the union cast as a function argument:

```
void hack (union foo);
...
hack ((union foo) x);
```

Declaring Attributes of Functions

In GNU C, you declare certain things about functions called in your program which help the compiler optimize function calls and check your code more carefully.

The keyword `__attribute__` allows you to specify special attributes when making a declaration. This keyword is followed by an attribute specification inside double parentheses. Eight attributes, `noreturn`, `const`, `format`, `section`, `constructor`, `destructor`, `unused` and `weak` are currently defined for functions. Other attributes, including `section` are supported for variables declarations (see section Specifying Attributes of Variables) and for types (see section Specifying Attributes of Types).

You may also specify attributes with `__' preceding and following each keyword. This allows you to use them in header files without being concerned about a possible macro of the same name. For example, you may use `__noreturn__` instead of `noreturn`.

`noreturn`

A few standard library functions, such as `abort` and `exit`, cannot return. GNU CC knows this automatically. Some programs define their own functions that never return. You can declare them `noreturn` to tell the compiler this fact. For example,

```
void fatal () __attribute__ ((noreturn));

void
```


```
fatal (...)
{
  ... /* Print error message. */ ...
  exit (1);
}
```

The `noreturn` keyword tells the compiler to assume that `fatal` cannot return. It can then optimize without regard to what would happen if `fatal` ever did return. This makes slightly better code. More importantly, it helps avoid spurious warnings of uninitialized variables.

Do not assume that registers saved by the calling function are restored before calling the `noreturn` function.

It does not make sense for a `noreturn` function to have a return type other than `void`.

The attribute `noreturn` is not implemented in GNU C versions earlier than 2.5. An alternative way to declare that a function does not return, which works in the current version and in some older versions, is as follows:

```
typedef void voidfn ();

volatile voidfn fatal;
```

`const`

Many functions do not examine any values except their arguments, and have no effects except the return value. Such a function can be subject to common subexpression elimination and loop optimization just as an arithmetic operator would be. These functions should be declared with the attribute `const`. For example,

```
int square (int) __attribute__ ((const));
```

says that the hypothetical function `square` is safe to call fewer times than the program says.

The attribute `const` is not implemented in GNU C versions earlier than 2.5. An alternative way to declare that a function has no side effects, which works in the current version and in some older versions, is as follows:

```
typedef int intfn ();

extern const intfn square;
```

This approach does not work in GNU C++ from 2.6.0 on, since the language specifies that the `const' must be attached to the return value.

Note that a function that has pointer arguments and examines the data pointed to must *not* be declared `const`. Likewise, a function that calls a non-`const` function usually must not be `const`. It does not make sense for a `const` function to return `void`.

`format (archetype, string-index, first-to-check)`

The `format` attribute specifies that a function takes `printf` or `scanf` style arguments whi should be type-checked against a format string. For example, the declaration:

```
extern int
my_printf (void *my_object, const char *my_format, ...)
      __attribute__ ((format (printf, 2, 3)));
```

causes the compiler to check the arguments in calls to `my_printf` for consistency with the `printf` style format string argument `my_format`.

The parameter archetype determines how the format string is interpreted, and should be either `printf` or `scanf`. The parameter string-index specifies which argument is the format string argument (starting from 1), while first-to-check is the number of the first argument to check against the format string. For functions where the arguments are not available to be checked (such as `vprintf`), specify the third parameter as zero. In this case the compiler only checks the format string for consistency.

In the example above, the format string (`my_format`) is the second argument of the function `my_print`, and the arguments to check start with the third argument, so the correct parameters for the format attribute are 2 and 3.

The `format` attribute allows you to identify your own functions which take format strings as arguments, so that GNU CC can check the calls to these functions for errors. The compiler always checks formats for the ANSI library functions `printf`, `fprintf`, `sprintf`, `scanf`, `fscanf`, `sscanf`, `vprintf`, `vfprintf` and `vsprintf` whenever such warnings are requested (using `-Wformat'), so there is no need to modify the header file `stdio.h'.

`section ("section-name")`

Normally, the compiler places the code it generates in the `text` section. Sometimes, however, you need additional sections, or you need certain particular functions to appear in special sections. The `section` attribute specifies that a function lives in a particular section. For example, the declaration:

```
extern void foobar (void) __attribute__ ((section ("bar")));
```

puts the function `foobar` in the `bar` section.

Some file formats do not support arbitrary sections so the `section` attribute is not available on all platforms. If you need to map the entire contents of a module to a particular section, consider using the facilities of the linker instead.

`constructor`

`destructor`

The `constructor` attribute causes the function to be called automatically before execution enters `main ()`. Similarly, the `destructor` attribute causes the function to be called automatically after `main ()` has completed or `exit ()` has been called. Functions with these attributes are useful for initializing data that will be used implicitly during the execution of the program.

These attributes are not currently implemented for Objective C.

`unused`

This attribute, attached to a function, means that the function is meant to be possibly unused. GNU CC will not produce a warning for this function.

`weak`

The `weak` attribute causes the declaration to be emitted as a weak symbol rather than a global. This is primarily useful in defining library functions which can be overridden in user code, though it can also be used with non-function declarations. Weak symbols are supported for ELF targets, and also for a.out targets when using the GNU assembler and linker.

`alias ("target")`

The `alias` attribute causes the declaration to be emitted as an alias for another symbol, which must be specified. For instance,

```
void __f () { /* do something */; }
void f () __attribute__ ((weak, alias ("__f")));
```

declares `f' to be a weak alias for `__f'. In C++, the mangled name for the target must be used.

`regparm (number)`

On the Intel 386, the `regparm` attribute causes the compiler to pass up to number integer arguments in registers EAX, EDX, and ECX instead of on the stack. Functions that take a variable number of arguments will continue to be passed all of their arguments on the stack.

`stdcall`

On the Intel 386, the `stdcall` attribute causes the compiler to assume that the called function will pop off the stack space used to pass arguments, unless it takes a variable number of arguments.

`cdecl`

On the Intel 386, the `cdecl` attribute causes the compiler to assume that the called function will pop off the stack space used to pass arguments, unless it takes a variable number of arguments. This is useful to override the effects of the `-mrtd' switch.

You can specify multiple attributes in a declaration by separating them by commas within the double parentheses or by immediately following an attribute declaration with another attribute declaration.

Some people object to the `__attribute__` feature, suggesting that ANSI C's `#pragma` should be used instead. There are two reasons for not doing this.

1. It is impossible to generate `#pragma` commands from a macro.
2. There is no telling what the same `#pragma` might mean in another compiler.

These two reasons apply to almost any application that might be proposed for `#pragma`. It is basically a mistake to use `#pragma` for *anything*.

Prototypes and Old-Style Function Definitions

GNU C extends ANSI C to allow a function prototype to override a later old-style non-prototype definition. Consider the following example:

```
/* Use prototypes unless the compiler is old-fashioned.  */
#if __STDC__
#define P(x) x
#else
#define P(x) ()
#endif

/* Prototype function declaration.  */
int isroot P((uid_t));

/* Old-style function definition.  */
int
isroot (x)   /* ??? lossage here ??? */
     uid_t x;
{
  return x == 0;
}
```

Suppose the type `uid_t` happens to be `short`. ANSI C does not allow this example, because subword arguments in old-style non-prototype definitions are promoted. Therefore in this example the function definition's argument is really an `int`, which does not match the prototype argument type of `short`.

This restriction of ANSI C makes it hard to write code that is portable to traditional C compilers, because the programmer does not know whether the `uid_t` type is `short`, `int`, or `long`. Therefore, in cases like these GNU C allows a prototype to override a later old-style definition. More precisely, in GNU C, a function prototype argument type overrides the argument type specified by a later old-style definition if the former type is the same as the latter type before promotion. Thus in GNU C the above example is equivalent to the following:

```
int isroot (uid_t);

int
isroot (uid_t x)
{
  return x == 0;
}
```

GNU C++ does not support old-style function definitions, so this extension is irrelevant.

Dollar Signs in Identifier Names

In GNU C, you may use dollar signs in identifier names. This is because many traditional C implementations allow such identifiers.

On some machines, dollar signs are allowed in identifiers if you specify `-traditional'. On a few systems they are allowed by default, even if you do not use `-traditional'. But they are never allowed if you specify `-ansi'.

There are certain ANSI C programs (obscure, to be sure) that would compile incorrectly if dollar signs were permitted in identifiers. For example:

```
#define foo(a) #a
#define lose(b) foo (b)
#define test$
lose (test)
```

The Character ESC in Constants

You can use the sequence `\e' in a string or character constant to stand for the ASCII character ESC.

Inquiring on Alignment of Types or Variables

The keyword `__alignof__` allows you to inquire about how an object is aligned, or the minimum alignment usually required by a type. Its syntax is just like `sizeof`.

For example, if the target machine requires a `double` value to be aligned on an 8-byte boundary, then `__alignof__ (double)` is 8. This is true on many RISC machines. On more traditional machine designs, `__alignof__ (double)` is 4 or even 2.

Some machines never actually require alignment; they allow reference to any data type even at an odd addresses. For these machines, `__alignof__` reports the *recommended* alignment of a type.

When the operand of `__alignof__` is an lvalue rather than a type, the value is the largest alignment that the lvalue is known to have. It may have this alignment as a result of its data type, or because it is part of a structure and inherits alignment from that structure. For example, after this declaration:

```
struct foo { int x; char y; } foo1;
```

the value of `__alignof__ (foo1.y)` is probably 2 or 4, the same as `__alignof__ (int)`, even though the data type of `foo1.y` does not itself demand any alignment.

A related feature which lets you specify the alignment of an object is `__attribute__ ((aligned (alignment)))`; see the following section.

Specifying Attributes of Variables

The keyword `__attribute__` allows you to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses. Eight attributes are currently defined for variables: `aligned`, `mode`, `nocommon`, `packed`, `section`, `transparent_union`, `unused`, and `weak`. Other attributes are available for functions (see section Declaring Attributes of Functions) and for types (see section Specifying Attributes of Types).

You may also specify attributes with `__' preceding and following each keyword. This allows you to use them in header files without being concerned about a possible macro of the same name. For example, you may use `__aligned__` instead of `aligned`.

`aligned (alignment)`

This attribute specifies a minimum alignment for the variable or structure field, measured in bytes. For example, the declaration:

```
int x __attribute__ ((aligned (16))) = 0;
```

causes the compiler to allocate the global variable `x` on a 16-byte boundary. On a 68040, this could be used in conjunction with an `asm` expression to access the `move16` instruction which requires 16-byte aligned operands.

You can also specify the alignment of structure fields. For example, to create a double-word aligned `int` pair, you could write:

```
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

This is an alternative to creating a union with a `double` member that forces the union to be double-word aligned.

It is not possible to specify the alignment of functions; the alignment of functions is determined by the machine's requirements and cannot be changed. You cannot specify alignment for a typedef name because such a name is just an alias, not a distinct type.

As in the preceding examples, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable or structure field. Alternatively, you can leave out the alignment factor and just ask the compiler to align a variable or field to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
short array[3] __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an `aligned` attribute specification, the compiler automatically sets the alignment for the declared variable or field to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables or fields that you have aligned this way.

The `aligned` attribute can only increase the alignment; but you can decrease it by specifying `packed` as well. See below.

Note that the effectiveness of `aligned` attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying `aligned(16)` in an `__attribute__` will still only provide you with 8 byte alignment. See your linker documentation for further information.

`mode (mode)`

This attribute specifies the data type for the declaration--whichever type corresponds to the mode mode. This in effect lets you request an integer or floating point type according to its width.

You may also specify a mode of `byte' or `__byte__' to indicate the mode corresponding to a one-byte integer, `word' or `__word__' for the mode of a one-word integer, and `pointer' or `__pointer__' for the mode used to represent pointers.

`nocommon`

This attribute specifies requests GNU CC not to place a variable "common" but instead to allocate space for it directly. If you specify the `-fno-common' flag, GNU CC will do this for all variables.

Specifying the `nocommon` attribute for a variable provides an initialization of zeros. A variable may only be initialized in one source file.

`packed`

The `packed` attribute specifies that a variable or structure field should have the smallest possible alignment--one byte for a variable, and one bit for a field, unless you specify a larger value with the `aligned` attribute.

Here is a structure in which the field `x` is packed, so that it immediately follows `a`:

```
struct foo
{
  char a;
  int x[2] __attribute__ ((packed));
};
```

`section ("section-name")`

Normally, the compiler places the objects it generates in sections like `data` and `bss`. Sometimes, however, you need additional sections, or you need certain particular variables to appear in special sections, for example to map to special hardware. The `section` attribute specifies that a variable (or function) lives in a particular section. For example, this small program uses several specific section names:

```
struct duart a __attribute__ ((section ("DUART_A"))) = { 0 };
struct duart b __attribute__ ((section ("DUART_B"))) = { 0 };
char stack[10000] __attribute__ ((section ("STACK"))) = { 0 };
```


```
int init_data_copy __attribute__ ((section ("INITDATACOPY"))) =

main()
{
  /* Initialize stack pointer */
  init_sp (stack + sizeof (stack));

  /* Initialize initialized data */
  memcpy (&init_data_copy, &data, &edata - &data);

  /* Turn on the serial ports */
  init_duart (&a);
  init_duart (&b);
}
```

Use the `section` attribute with an *initialized* definition of a *global* variable, as shown in the example. GNU CC issues a warning and otherwise ignores the `section` attribute in uninitialized variable declarations.

You may only use the `section` attribute with a fully initialized global definition because of the way linkers work. The linker requires each object be defined once, with the exception that uninitialized variables tentatively go in the `common` (or `bss`) section and can be multiply "defined". You can force a variable to be initialized with the `-fno-common' flag or the `nocommon` attribute.

Some file formats do not support arbitrary sections so the `section` attribute is not available on all platforms. If you need to map the entire contents of a module to a particular section, consider using the facilities of the linker instead.

`transparent_union`

This attribute, attached to a function argument variable which is a union, means to pass the argument in the same way that the first union member would be passed. You can also use this attribute on a `typedef` for a union data type; then it applies to all function arguments with that type.

`unused`

This attribute, attached to a variable, means that the variable is meant to be possibly unused. GNU CC will not produce a warning for this variable.

`weak`

The `weak` attribute is described in See section Declaring Attributes of Functions.

To specify multiple attributes, separate them by commas within the double parentheses: for example, `__attribute__ ((aligned (16), packed))'.

Specifying Attributes of Types

The keyword `__attribute__` allows you to specify special attributes of `struct` and `union` types when you define such types. This keyword is followed by an attribute specification inside double parentheses. Three attributes are currently defined for types: `aligned`, `packed`, and `transparent_union`. Other attributes are defined for functions (see section Declaring Attributes of Functions) and for variables (see section Specifying Attributes of Variables).

You may also specify any one of these attributes with `__' preceding and following its keyword. This allows you to use these attributes in header files without being concerned about a possible macro of the same name. For example, you may use `__aligned__` instead of `aligned`.

You may specify the `aligned` and `transparent_union` attributes either in a `typedef` declaration or just past the closing curly brace of a complete enum, struct or union type *definition* and the `packed` attribute only past the closing brace of a definition.

`aligned (alignment)`

This attribute specifies a minimum alignment (in bytes) for variables of the specified type. For example, the declarations:

```
struct S { short f[3]; } __attribute__ ((aligned (8));
typedef int more_aligned_int __attribute__ ((aligned (8));
```

force the compiler to insure (as fas as it can) that each variable whose type is `struct S` or `more_aligned_int` will be allocated and aligned *at least* on a 8-byte boundary. On a Sparc, having all variables of type `struct S` aligned to 8-byte boundaries allows the compiler to use the `ldd` and `std` (doubleword load and store) instructions when copying one variable of type `struct S` to another, thus improving run-time efficiency.

Note that the alignment of any given `struct` or `union` type is required by the ANSI C standard to be at least a perfect multiple of the lowest common multiple of the alignments of all of the members of the `struct` or `union` in question. This means that you *can* effectively adjust the alignment of a `struct` or `union` type by attaching an `aligned` attribute to any one of the members of such a type, but the notation illustrated in the example above is a more obvious, intuitive, and readable way to request the compiler to adjust the alignment of an entire `struct` or `union` type.

As in the preceding example, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given `struct` or `union` type. Alternatively, you can leave out the alignment factor and just ask the compiler to align a type to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
struct S { short f[3]; } __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an `aligned` attribute specification, the compiler automatically sets the alignment for the type to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations

more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables which have types that you have aligned this way.

In the example above, if the size of each `short` is 2 bytes, then the size of the entire `struct S` type is 6 bytes. The smallest power of two which is greater than or equal to that is 8, so the compiler sets the alignment for the entire `struct S` type to 8 bytes.

Note that although you can ask the compiler to select a time-efficient alignment for a given type and then declare only individual stand-alone objects of that type, the compiler's ability to select a time-efficient alignment is primarily useful only when you plan to create arrays of variables having the relevant (efficiently aligned) type. If you declare or use arrays of variables of an efficiently-aligned type, then it is likely that your program will also be doing pointer arithmetic (or subscripting, which amounts to the same thing) on pointers to the relevant type, and the code that the compiler generates for these pointer arithmetic operations will often be more efficient for efficiently-aligned types than for other types.

The `aligned` attribute can only increase the alignment; but you can decrease it by specifying `packed` as well. See below.

Note that the effectiveness of `aligned` attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying `aligned(16)` in an `__attribute__` will still only provide you with 8 byte alignment. See your linker documentation for further information.

`packed`

This attribute, attached to an `enum`, `struct`, or `union` type definition, specified that the minimum required memory be used to represent the type.

Specifying this attribute for `struct` and `union` types is equivalent to specifying the `packed` attribute on each of the structure or union members. Specifying the `-fshort-enums' flag on the line is equivalent to specifying the `packed` attribute on all `enum` definitions.

You may only specify this attribute after a closing curly brace on an `enum` definition, not in a `typedef` declaration.

`transparent_union`

This attribute, attached to a `union` type definition, indicates that any variable having that union type should, if passed to a function, be passed in the same way that the first union member would be passed. For example:

```
union foo
{
  char a;
  int x[2];
} __attribute__ ((transparent_union));
```

To specify multiple attributes, separate them by commas within the double parentheses: for example, `__attribute__ ((aligned (16), packed))'.

An Inline Function is As Fast As a Macro

By declaring a function `inline`, you can direct GNU CC to integrate that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included. The effect on code size is less predictable; object code may be larger or smaller with function inlining, depending on the particular case. Inlining of functions is an optimization and it really "works" only in optimizing compilation. If you don't use `-O', no function is really inline.

To declare a function inline, use the `inline` keyword in its declaration, like this:

```
inline int
inc (int *a)
{
  (*a)++;
}
```

(If you are writing a header file to be included in ANSI C programs, write `__inline__` instead of `inline`. See section Alternate Keywords.)

You can also make all "simple enough" functions inline with the option `-finline-functions'. Note that certain usages in a function definition can make it unsuitable for inline substitution.

Note that in C and Objective C, unlike C++, the `inline` keyword does not affect the linkage of the function.

GNU CC automatically inlines member functions defined within the class body of C++ programs even if they are not explicitly declared `inline`. (You can override this with `-fno-default-inline'; see section Options Controlling C++ Dialect.)

When a function is both inline and `static`, if all calls to the function are integrated into the caller, and the function's address is never used, then the function's own assembler code is never referenced. In this case, GNU CC does not actually output assembler code for the function, unless you specify the option `-fkeep-inline-functions'. Some calls cannot be integrated for various reasons (in particular, calls that precede the function's definition cannot be integrated, and neither can recursive calls within the definition). If there is a nonintegrated call, then the function is compiled to assembler code as usual. The function must also be compiled as usual if the program refers to its address, because that can't be inlined.

When an inline function is not `static`, then the compiler must assume that there may be calls from other source files; since a global symbol can be defined only once in any program, the function must not be defined in the other source files, so the calls therein cannot be integrated. Therefore, a non-`static` inline function is always compiled on its own in the usual fashion.

If you specify both `inline` and `extern` in the function definition, then the definition is used only for inlining. In no case is the function compiled on its own, not even if you refer to its address explicitly. Such an address becomes an external reference, as if you had only declared the function, and had not defined it.

This combination of `inline` and `extern` has almost the effect of a macro. The way to use it is to put a function definition in a header file with these keywords, and put another copy of the definition (lacking `inline` and `extern`) in a library file. The definition in the header file will cause most calls to the function to be inlined. If any uses of the function remain, they will refer to the single copy in the library.

GNU C does not inline any functions when not optimizing. It is not clear whether it is better to inline or not, in this case, but we found that a correct implementation when not optimizing was difficult. So we did the easy thing, and turned it off.

Assembler Instructions with C Expression Operands

In an assembler instruction using `asm`, you can now specify the operands of the instruction using C expressions. This means no more guessing which registers or memory locations will contain the data you want to use.

You must specify an assembler instruction template much like what appears in a machine description, plus an operand constraint string for each operand.

For example, here is how to use the 68881's `fsinx` instruction:

```
asm ("fsinx %1,%0" : "=f" (result) : "f" (angle));
```

Here `angle` is the C expression for the input operand while `result` is that of the output operand. Each has `"f"' as its operand constraint, saying that a floating point register is required. The `=' in `=f' indicates that the operand is an output; all output operands' constraints must use `='. The constraints use the same language used in the machine description (see section Operand Constraints).

Each operand is described by an operand-constraint string followed by the C expression in parentheses. A colon separates the assembler template from the first output operand, and another separates the last output operand from the first input, if any. Commas separate output operands and separate inputs. The total number of operands is limited to ten or to the maximum number of operands in any instruction pattern in the machine description, whichever is greater.

If there are no output operands, and there are input operands, then there must be two consecutive colons surrounding the place where the output operands would go.

Output operand expressions must be lvalues; the compiler can check this. The input operands need not be lvalues. The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. It does not parse the assembler instruction template and does not know what it means, or whether it is valid assembler input. The extended `asm` feature is most often used for machine instructions that the compiler itself does not know exist. If the output expression cannot be directly

addressed (for example, it is a bit field), your constraint must allow a register. In that case, GNU CC may use the register as the output of the `asm`, and then store that register into the output.

The output operands must be write-only; GNU CC will assume that the values in these operands before the instruction are dead and need not be generated. Extended asm does not support input-output or read-write operands. For this reason, the constraint character `+', which indicates such an operand, may not be used.

When the assembler instruction has a read-write operand, or an operand in which only some of the bits are to be changed, you must logically split its function into two separate operands, one input operand and one write-only output operand. The connection between them is expressed by constraints which say they need to be in the same location when the instruction executes. You can use the same C expression for both operands, or different expressions. For example, here we write the (fictitious) `combine' instruction with `bar` as its read-only source operand and `foo` as its read-write destination:

```
asm ("combine %2,%0" : "=r" (foo) : "0" (foo), "g" (bar));
```

The constraint `"0"' for operand 1 says that it must occupy the same location as operand 0. A digit in constraint is allowed only in an input operand, and it must refer to an output operand.

Only a digit in the constraint can guarantee that one operand will be in the same place as another. The mere fact that `foo` is the value of both operands is not enough to guarantee that they will be in the same place in the generated assembler code. The following would not work:

```
asm ("combine %2,%0" : "=r" (foo) : "r" (foo), "g" (bar));
```

Various optimizations or reloading could cause operands 0 and 1 to be in different registers; GNU CC knows no reason not to do so. For example, the compiler might find a copy of the value of `foo` in one register and use it for operand 1, but generate the output operand 0 in a different register (copying it afterward to `foo`'s own address). Of course, since the register for operand 1 is not even mentioned in the assembler code, the result will not work, but GNU CC can't tell that.

Some instructions clobber specific hard registers. To describe this, write a third colon after the input operands, followed by the names of the clobbered hard registers (given as strings). Here is a realistic example for the Vax:

```
asm volatile ("movc3 %0,%1,%2"
              : /* no outputs */
              : "g" (from), "g" (to), "g" (count)
              : "r0", "r1", "r2", "r3", "r4", "r5");
```

If you refer to a particular hardware register from the assembler code, then you will probably have to list the register after the third colon to tell the compiler that the register's value is modified. In many assemblers, the register names begin with `%'; to produce one `%' in the assembler code, you must write `%%' in the input.

If your assembler instruction can alter the condition code register, add `cc' to the list of clobbered registers. GNU CC on some machines represents the condition codes as a specific hardware register; `cc'

serves to name this register. On other machines, the condition code is handled differently, and specifying `cc' has no effect. But it is valid no matter what the machine.

If your assembler instruction modifies memory in an unpredictable fashion, add `memory' to the list of clobbered registers. This will cause GNU CC to not keep memory values cached in registers across the assembler instruction.

You can put multiple assembler instructions together in a single `asm` template, separated either with newlines (written as `\n') or with semicolons if the assembler allows such semicolons. The GNU assembler allows semicolons and all Unix assemblers seem to do so. The input operands are guaranteed not to use any of the clobbered registers, and neither will the output operands' addresses, so you can read and write the clobbered registers as many times as you like. Here is an example of multiple instructions in a template; it assumes that the subroutine `_foo` accepts arguments in registers 9 and 10:

```
asm ("movl %0,r9;movl %1,r10;call _foo"
     : /* no outputs */
     : "g" (from), "g" (to)
     : "r9", "r10");
```

Unless an output operand has the `&' constraint modifier, GNU CC may allocate it in the same register as an unrelated input operand, on the assumption that the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use `&' for each output operand that may not overlap an input. See section Constraint Modifier Characters.

If you want to test the condition code produced by an assembler instruction, you must include a branch and a label in the `asm` construct, as follows:

```
asm ("clr %0;frob %1;beq 0f;mov #1,%0;0:"
     : "g" (result)
     : "g" (input));
```

This assumes your assembler supports local labels, as the GNU assembler and most Unix assemblers do.

Speaking of labels, jumps from one `asm` to another are not supported. The compiler's optimizers do not know about these jumps, and therefore they cannot take account of them when deciding how to optimize.

Usually the most convenient way to use these `asm` instructions is to encapsulate them in macros that look like functions. For example,

```
#define sin(x)       \
({ double __value, __arg = (x);   \
   asm ("fsinx %1,%0": "=f" (__value): "f" (__arg));  \
   __value; })
```

Here the variable `__arg` is used to make sure that the instruction operates on a proper `double` value, and to accept only those arguments `x` which can convert automatically to a `double`.

Another way to make sure the instruction operates on the correct data type is to use a cast in the `asm`. This is different from using a variable `__arg` in that it converts more different types. For example, if the desired type were `int`, casting the argument to `int` would accept a pointer with no complaint, while assigning the argument to an `int` variable named `__arg` would warn about using a pointer unless the caller explicitly casts it.

If an `asm` has output operands, GNU CC assumes for optimization purposes that the instruction has no side effects except to change the output operands. This does not mean that instructions with a side effect cannot be used, but you must be careful, because the compiler may eliminate them if the output operands aren't used, or move them out of loops, or replace two with one if they constitute a common subexpression. Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

You can prevent an `asm` instruction from being deleted, moved significantly, or combined, by writing the keyword `volatile` after the `asm`. For example:

```
#define set_priority(x)  \
asm volatile ("set_priority %0": /* no outputs */ : "g" (x))
```

An instruction without output operands will not be deleted or moved significantly, regardless, unless it is unreachable.

Note that even a volatile `asm` instruction can be moved in ways that appear insignificant to the compiler, such as across jump instructions. You can't expect a sequence of volatile `asm` instructions to remain perfectly consecutive. If you want consecutive output, use a single `asm`.

It is a natural idea to look for a way to give access to the condition code left by the assembler instruction. However, when we attempted to implement this, we found no way to make it work reliably. The problem is that output operands might need reloading, which would result in additional following "store" instructions. On most machines, these instructions would alter the condition code before there was time to test it. This problem doesn't arise for ordinary "test" and "compare" instructions because they don't have any output operands.

If you are writing a header file that should be includable in ANSI C programs, write `__asm__` instead of `asm`. See section Alternate Keywords.

Controlling Names Used in Assembler Code

You can specify the name to be used in the assembler code for a C function or variable by writing the `asm` (or `__asm__`) keyword after the declarator as follows:

```
int foo asm ("myfoo") = 2;
```

This specifies that the name to be used for the variable `foo` in the assembler code should be `myfoo' rather than the usual `_foo'.

On systems where an underscore is normally prepended to the name of a C function or variable, this

feature allows you to define names for the linker that do not start with an underscore.

You cannot use `asm` in this way in a function *definition*; but you can get the same effect by writing a declaration for the function before its definition and putting `asm` there, like this:

```
extern func () asm ("FUNC");

func (x, y)
     int x, y;
...
```

It is up to you to make sure that the assembler names you choose do not conflict with any other assembler symbols. Also, you must not use a register name; that would produce completely invalid assembler code. GNU CC does not as yet have the ability to store static variables in registers. Perhaps that will be added.

Variables in Specified Registers

GNU C allows you to put a few global variables into specified hardware registers. You can also specify the register in which an ordinary register variable should be allocated.

- Global register variables reserve registers throughout the program. This may be useful in programs such as programming language interpreters which have a couple of global variables that are accessed very often.
- Local register variables in specific registers do not reserve the registers. The compiler's data flow analysis is capable of determining where the specified registers contain live values, and where they are available for other uses.

  These local variables are sometimes convenient for use with the extended `asm` feature (see section Assembler Instructions with C Expression Operands), if you want to write one output of the assembler instruction directly into a particular register. (This will work provided the register you specify fits the constraints specified for that operand in the `asm`.)

Defining Global Register Variables

You can define a global register variable in GNU C like this:

```
register int *foo asm ("a5");
```

Here `a5` is the name of the register which should be used. Choose a register which is normally saved and restored by function calls on your machine, so that library routines will not clobber it.

Naturally the register name is cpu-dependent, so you would need to conditionalize your program according to cpu type. The register `a5` would be a good choice on a 68000 for a variable of pointer type. On machines with register windows, be sure to choose a "global" register that is not affected magically by the function call mechanism.

In addition, operating systems on one type of cpu may differ in how they name the registers; then you would need additional conditionals. For example, some 68000 operating systems call this register %a

Eventually there may be a way of asking the compiler to choose a register automatically, but first we need to figure out how it should choose and how to enable you to guide the choice. No solution is evident.

Defining a global register variable in a certain register reserves that register entirely for this use, at least within the current compilation. The register will not be allocated for any other purpose in the functions in the current compilation. The register will not be saved and restored by these functions. Stores into this register are never deleted even if they would appear to be dead, but references may be deleted or moved or simplified.

It is not safe to access the global register variables from signal handlers, or from more than one thread of control, because the system library routines may temporarily use the register for other things (unless you recompile them specially for the task at hand).

It is not safe for one function that uses a global register variable to call another such function `foo` by way of a third function `lose` that was compiled without knowledge of this variable (i.e. in a different source file in which the variable wasn't declared). This is because `lose` might save the register and put some other value there. For example, you can't expect a global register variable to be available in the comparison-function that you pass to `qsort`, since `qsort` might have put something else in that register. (If you are prepared to recompile `qsort` with the same global register variable, you can solve this problem.)

If you want to recompile `qsort` or other source files which do not actually use your global register variable, so that they will not use that register for any other purpose, then it suffices to specify the compiler option `-ffixed-reg'. You need not actually add a global register declaration to their source code.

A function which can alter the value of a global register variable cannot safely be called from a function compiled without this variable, because it could clobber the value the caller expects to find there on return. Therefore, the function which is the entry point into the part of the program that uses the global register variable must explicitly save and restore the value which belongs to its caller.

On most machines, `longjmp` will restore to each global register variable the value it had at the time of the `setjmp`. On some machines, however, `longjmp` will not change the value of global register variables. To be portable, the function that called `setjmp` should make other arrangements to save the values of the global register variables, and to restore them in a `longjmp`. This way, the same thing will happen regardless of what `longjmp` does.

All global register variable declarations must precede all function definitions. If such a declaration could appear after function definitions, the declaration would be too late to prevent the register from being used for other purposes in the preceding functions.

Global register variables may not have initial values, because an executable file has no means to supply initial contents for a register.

On the Sparc, there are reports that g3 ... g7 are suitable registers, but certain library functions, such as `getwd`, as well as the subroutines for division and remainder, modify g3 and g4. g1 and g2 are local

temporaries.

On the 68000, a2 ... a5 should be suitable, as should d2 ... d7. Of course, it will not do to use more than a few of those.

Specifying Registers for Local Variables

You can define a local register variable with a specified register like this:

```
register int *foo asm ("a5");
```

Here `a5` is the name of the register which should be used. Note that this is the same syntax used for defining global register variables, but for a local variable it would appear within a function.

Naturally the register name is cpu-dependent, but this is not a problem, since specific registers are most often useful with explicit assembler instructions (see section Assembler Instructions with C Expression Operands). Both of these things generally require that you conditionalize your program according to cpu type.

In addition, operating systems on one type of cpu may differ in how they name the registers; then you would need additional conditionals. For example, some 68000 operating systems call this register `%a5`.

Eventually there may be a way of asking the compiler to choose a register automatically, but first we need to figure out how it should choose and how to enable you to guide the choice. No solution is evident.

Defining such a register variable does not reserve the register; it remains available for other uses in places where flow control determines the variable's value is not live. However, these registers are made unavailable for use in the reload pass. I would not be surprised if excessive use of this feature leaves the compiler too few available registers to compile certain functions.

Alternate Keywords

The option `-traditional' disables certain keywords; `-ansi' disables certain others. This causes trouble when you want to use GNU C extensions, or ANSI C features, in a general-purpose header file that should be usable by all programs, including ANSI C programs and traditional ones. The keywords `asm`, `typeof` and `inline` cannot be used since they won't work in a program compiled with `-ansi', while the keywords `const`, `volatile`, `signed`, `typeof` and `inline` won't work in a program compiled with `-traditional'.

The way to solve these problems is to put `__' at the beginning and end of each problematical keyword. For example, use `__asm__` instead of `asm`, `__const__` instead of `const`, and `__inline__` instead of `inline`.

Other C compilers won't accept these alternative keywords; if you want to compile with another compiler, you can define the alternate keywords as macros to replace them with the customary keywords. It looks like this:


```
#ifndef __GNUC__
#define __asm__ asm
#endif
```

`-pedantic' causes warnings for many GNU C extensions. You can prevent such warnings within one expression by writing `__extension__` before the expression. `__extension__` has no effect aside from this.

Incomplete `enum` Types

You can define an `enum` tag without specifying its possible values. This results in an incomplete type, much like what you get if you write `struct foo` without describing the elements. A later declaration which does specify the possible values completes the type.

You can't allocate variables or storage using the type while it is incomplete. However, you can work with pointers to that type.

This extension may not be very useful, but it makes the handling of `enum` more consistent with the way `struct` and `union` are handled.

This extension is not supported by GNU C++.

Function Names as Strings

GNU CC predefines two string variables to be the name of the current function. The variable `__FUNCTION__` is the name of the function as it appears in the source. The variable `__PRETTY_FUNCTION__` is the name of the function pretty printed in a language specific fashion.

These names are always the same in a C function, but in a C++ function they may be different. For example, this program:

```
extern "C" {
extern int printf (char *, ...);
}

class a {
 public:
  sub (int i)
    {
      printf ("__FUNCTION__ = %s\n", __FUNCTION__);
      printf ("__PRETTY_FUNCTION__ = %s\n", __PRETTY_FUNCTION__);
    }
};

int
```


```
main (void)
{
  a ax;
  ax.sub (0);
  return 0;
}
```

gives this output:

```
__FUNCTION__ = sub
__PRETTY_FUNCTION__ = int  a::sub (int)
```

Go to the previous, next section.

Go to the previous, next section.

Extensions to the C++ Language

The GNU compiler provides these extensions to the C++ language (and you can also use most of the C language extensions in your C++ programs). If you want to write code that checks whether these features are available, you can test for the GNU compiler the same way as for C programs: check for a predefined macro __GNUC__. You can also use __GNUG__ to test specifically for GNU C++ (see section 'Standard Predefined Macros' in The C Preprocessor).

Named Return Values in C++

GNU C++ extends the function-definition syntax to allow you to specify a name for the result of a function outside the body of the definition, in C++ programs:

```
type
functionname (args) return resultname;
{
  ...
  body
  ...
}
```

You can use this feature to avoid an extra constructor call when a function result has a class type. For example, consider a function `m`, declared as `X v = m ();', whose result is of class `X`:

```
X
m ()
{
  X b;
  b.a = 23;
  return b;
}
```

Although `m` appears to have no arguments, in fact it has one implicit argument: the address of the return value. At invocation, the address of enough space to hold `v` is sent in as the implicit argument. Then `b` is constructed and its `a` field is set to the value 23. Finally, a copy constructor (a constructor of the form `X(X&)') is applied to `b`, with the (implicit) return value location as the target, so that `v` is now bound to the return value.

But this is wasteful. The local `b` is declared just to hold something that will be copied right out. While a compiler that combined an "elision" algorithm with interprocedural data flow analysis could conceivably eliminate all of this, it is much more practical to allow you to assist the compiler in generating efficient code by manipulating the return value explicitly, thus avoiding the local variable and copy constructor

altogether.

Using the extended GNU C++ function-definition syntax, you can avoid the temporary allocation and copying by naming `r` as your return value at the outset, and assigning to its `a` field directly:

```
X
m () return r;
{
  r.a = 23;
}
```

The declaration of `r` is a standard, proper declaration, whose effects are executed **before** any of the body of `m`.

Functions of this type impose no additional restrictions; in particular, you can execute `return` statements, or return implicitly by reaching the end of the function body ("falling off the edge"). Cases like

```
X
m () return r (23);
{
  return;
}
```

(or even `X m () return r (23); { }') are unambiguous, since the return value `r` has been initialized in either case. The following code may be hard to read, but also works predictably:

```
X
m () return r;
{
  X b;
  return b;
}
```

The return value slot denoted by `r` is initialized at the outset, but the statement `return b;' overrides this value. The compiler deals with this by destroying `r` (calling the destructor if there is one, or doing nothing if there is not), and then reinitializing `r` with `b`.

This extension is provided primarily to help people who use overloaded operators, where there is a great need to control not just the arguments, but the return values of functions. For classes where the copy constructor incurs a heavy performance penalty (especially in the common case where there is a quick default constructor), this is a major savings. The disadvantage of this extension is that you do not control when the default constructor for the return value is called: it is always called at the beginning.

Minimum and Maximum Operators in C++

It is very convenient to have operators which return the "minimum" or the "maximum" of two arguments. In GNU C++ (but not in GNU C),

`a <? b`

is the minimum, returning the smaller of the numeric values a and b;

`a >? b`

is the maximum, returning the larger of the numeric values a and b.

These operations are not primitive in ordinary C++, since you can use a macro to return the minimum of two things in C++, as in the following example.

```
#define MIN(X,Y) ((X) < (Y) ? : (X) : (Y))
```

You might then use `int min = MIN (i, j);' to set min to the minimum value of variables i and j.

However, side effects in `X` or `Y` may cause unintended behavior. For example, `MIN (i++, j++)` will fail, incrementing the smaller counter twice. A GNU C extension allows you to write safe macros that avoid this kind of problem (see section Naming an Expression's Type). However, writing `MIN` and `MAX` as macros also forces you to use function-call notation notation for a fundamental arithmetic operation. Using GNU C++ extensions, you can write `int min = i <? j;' instead.

Since `<?` and `>?` are built into the compiler, they properly handle expressions with side-effects; `int min = i++ <? j++;' works correctly.

`goto` and Destructors in GNU C++

In C++ programs, you can safely use the `goto` statement. When you use it to exit a block which contains aggregates requiring destructors, the destructors will run before the `goto` transfers control. (In ANSI C++, `goto` is restricted to targets within the current block.)

The compiler still forbids using `goto` to *enter* a scope that requires constructors.

Declarations and Definitions in One Header

C++ object definitions can be quite complex. In principle, your source code will need two kinds of things for each object that you use across more than one source file. First, you need an interface specification, describing its structure with type declarations and function prototypes. Second, you need the implementation itself. It can be tedious to maintain a separate interface description in a header file, in parallel to the actual implementation. It is also dangerous, since separate interface and implementation definitions may not remain parallel.

With GNU C++, you can use a single header file for both purposes.

Warning: The mechanism to specify this is in transition. For the nonce, you must use one of

two `#pragma` commands; in a future release of GNU C++, an alternative mechanism will make these `#pragma` commands unnecessary.

The header file contains the full definitions, but is marked with `#pragma interface' in the source code. This allows the compiler to use the header file only as an interface specification when ordinary source files incorporate it with `#include`. In the single source file where the full implementation belongs, you can use either a naming convention or `#pragma implementation' to indicate this alternate use of the header file.

`#pragma interface`

`#pragma interface "subdir/objects.h"`

Use this directive in *header files* that define object classes, to save space in most of the object files that use those classes. Normally, local copies of certain information (backup copies of inline member functions, debugging information, and the internal tables that implement virtual functions) must be kept in each object file that includes class definitions. You can use this pragma to avoid such duplication. When a header file containing `#pragma interface' is included in a compilation, this auxiliary information will not be generated (unless the main input source file itself uses `#pragma implementation'). Instead, the object files will contain references to be resolved at link time.

The second form of this directive is useful for the case where you have multiple headers with the same name in different directories. If you use this form, you must specify the same string to `#pragma implementation'.

`#pragma implementation`

`#pragma implementation "objects.h"`

Use this pragma in a *main input file*, when you want full output from included header files to be generated (and made globally visible). The included header file, in turn, should use `#pragma interface'. Backup copies of inline member functions, debugging information, and the internal tables used to implement virtual functions are all generated in implementation files.

If you use `#pragma implementation' with no argument, it applies to an include file with the same basename(3) as your source file. For example, in `allclass.cc', `#pragma implementation' by itself is equivalent to `#pragma implementation "allclass.h"'.

In versions of GNU C++ prior to 2.6.0 `allclass.h' was treated as an implementation file whenever you would include it from `allclass.cc' even if you never specified `#pragma implementation'. This was deemed to be more trouble than it was worth, however, and disabled.

If you use an explicit `#pragma implementation', it must appear in your source file *before* you include the affected header files.

Use the string argument if you want a single implementation file to include code from multiple header files. (You must also use `#include' to include the header file; `#pragma implementation' only specifies how to use the file--it doesn't actually include it.)

There is no way to split up the contents of a single header file into multiple implementation files.

`#pragma implementation' and `#pragma interface' also have an effect on function inlining.

If you define a class in a header file marked with `#pragma interface', the effect on a function defined in that class is similar to an explicit `extern` declaration--the compiler emits no code at all to define an independent version of the function. Its definition is used only for inlining with its callers.

Conversely, when you include the same header file in a main source file that declares it as `#pragma implementation', the compiler emits code for the function itself; this defines a version of the function that can be found via pointers (or by callers compiled without inlining). If all calls to the function can be inlined, you can avoid emitting the function by compiling with `-fno-implement-inlines'. If any calls were not inlined, you will get linker errors.

Where's the Template?

C++ templates are the first language feature to require more intelligence from the environment than one usually finds on a UNIX system. Somehow the compiler and linker have to make sure that each template instance occurs exactly once in the executable if it is needed, and not at all otherwise. There are two basic approaches to this problem, which I will refer to as the Borland model and the Cfront model.

Borland model

> Borland C++ solved the template instantiation problem by adding the code equivalent of common blocks to their linker; template instances are emitted in each translation unit that uses them, and they are collapsed together at run time. The advantage of this model is that the linker only has to consider the object files themselves; there is no external complexity to worry about. This disadvantage is that compilation time is increased because the template code is being compiled repeatedly. Code written for this model tends to include definitions of all member templates in the header file, since they must be seen to be compiled.

Cfront model

> The AT&T C++ translator, Cfront, solved the template instantiation problem by creating the notion of a template repository, an automatically maintained place where template instances are stored. As individual object files are built, notes are placed in the repository to record where templates and potential type arguments were seen so that the subsequent instantiation step knows where to find them. At link time, any needed instances are generated and linked in. The advantages of this model are more optimal compilation speed and the ability to use the system linker; to implement the Borland model a compiler vendor also needs to replace the linker. The disadvantages are vastly increased complexity, and thus potential for error; theoretically, this should be just as transparent, but in practice it has been very difficult to build multiple programs in one directory and one program in multiple directories using Cfront. Code written for this model tends to separate definitions of non-inline member templates into a separate file, which is magically found by the link preprocessor when a template needs to be instantiated.

Currently, g++ implements neither automatic model. The g++ team hopes to have a repository working for 2.7.0. In the mean time, you have three options for dealing with template instantiations:

1. Do nothing. Pretend g++ does implement automatic instantiation management. Code written for the Borland model will work fine, but each translation unit will contain instances of each of the templates it uses. In a large program, this can lead to an unacceptable amount of code duplication.
2. Add `#pragma interface' to all files containing template definitions. For each of these files, add

`#pragma implementation "filename"' to the top of some `.C' file which `#include's it. Then compile everything with -fexternal-templates. The templates will then only be expanded in the translation unit which implements them (i.e. has a `#pragma implementation' line for the file where they live); all other files will use external references. If you're lucky, everything should work properly. If you get undefined symbol errors, you need to make sure that each template instance which is used in the program is used in the file which implements that template. If you don't have any use for a particular instance in that file, you can just instantiate it explicitly, using the syntax from the latest C++ working paper:

```
template class A<int>;
template ostream& operator << (ostream&, const A<int>&);
```

This strategy will work with code written for either model. If you are using code written for the Cfront model, the file containing a class template and the file containing its member templates should be implemented in the same translation unit.

A slight variation on this approach is to use the flag -falt-external-templates instead; this flag causes template instances to be emitted in the translation unit that implements the header where they are first instantiated, rather than the one which implements the file where the templates are defined. This header must be the same in all translation units, or things are likely to break.

See section Declarations and Definitions in One Header, for more discussion of these pragmas.

3. Explicitly instantiate all the template instances you use, and compile with -fno-implicit-templates. This is probably your best bet; it may require more knowledge of exactly which templates you are using, but it's less mysterious than the previous approach, and it doesn't require any `#pragma's or other g++-specific code. You can scatter the instantiations throughout your program, you can create one big file to do all the instantiations, or you can create tiny files like

```
#include "Foo.h"
#include "Foo.cc"

template class Foo<int>;
```

for each instance you need, and create a template instantiation library from those. I'm partial to the last, but your mileage may vary. If you are using Cfront-model code, you can probably get away with not using -fno-implicit-templates when compiling files that don't `#include' the member template definitions.

Type Abstraction using Signatures

In GNU C++, you can use the keyword `signature` to define a completely abstract class interface as a datatype. You can connect this abstraction with actual classes using signature pointers. If you want to use signatures, run the GNU compiler with the `-fhandle-signatures' command-line option. (With this option, the compiler reserves a second keyword `sigof` as well, for a future extension.)

Roughly, signatures are type abstractions or interfaces of classes. Some other languages have similar

facilities. C++ signatures are related to ML's signatures, Haskell's type classes, definition modules in Modula-2, interface modules in Modula-3, abstract types in Emerald, type modules in Trellis/Owl, categories in Scratchpad II, and types in POOL-I. For a more detailed discussion of signatures, see Signatures: A C++ Extension for Type Abstraction and Subtype Polymorphism by Gerald Baumgartner and Vincent F. Russo (Tech report CSD--TR--93--059, Dept. of Computer Sciences, Purdue University, December 1994, to appear in *Software Practice & Experience*). You can get the tech report by anonymous FTP from `ftp.cs.purdue.edu` in `` `pub/reports/TR93-059.PS.Z' ``.

Syntactically, a signature declaration is a collection of member function declarations and nested type declarations. For example, this signature declaration defines a new abstract type `S` with member functions `int foo ()' and `int bar (int)':

```
signature S
{
  int foo ();
  int bar (int);
};
```

Since signature types do not include implementation definitions, you cannot write an instance of a signature directly. Instead, you can define a pointer to any class that contains the required interfaces as a signature pointer. Such a class implements the signature type.

To use a class as an implementation of `S`, you must ensure that the class has public member functions `int foo ()' and `int bar (int)'. The class can have other member functions as well, public or not; as long as it offers what's declared in the signature, it is suitable as an implementation of that signature type.

For example, suppose that `C` is a class that meets the requirements of signature `S` (`C` conforms to `S`). Then

```
C obj;
S * p = &obj;
```

defines a signature pointer `p` and initializes it to point to an object of type `C`. The member function call `int i = p->foo ();' executes `obj.foo ()'.

Abstract virtual classes provide somewhat similar facilities in standard C++. There are two main advantages to using signatures instead:

1. Subtyping becomes independent from inheritance. A class or signature type `T` is a subtype of a signature type `S` independent of any inheritance hierarchy as long as all the member functions declared in `S` are also found in `T`. So you can define a subtype hierarchy that is completely independent from any inheritance (implementation) hierarchy, instead of being forced to use types that mirror the class inheritance hierarchy.
2. Signatures allow you to work with existing class hierarchies as implementations of a signature type. If those class hierarchies are only available in compiled form, you're out of luck with abstract virtual classes, since an abstract virtual class cannot be retrofitted on top of existing class hierarchies. So you would be required to write interface classes as subtypes of the abstract virtual class.

There is one more detail about signatures. A signature declaration can contain member function *definitions* as well as member function declarations. A signature member function with a full definition is called a *default implementation*; classes need not contain that particular interface in order to conform. For example, a class `C` can conform to the signature

```
signature T
{
  int f (int);
  int f0 () { return f (0); };
};
```

whether or not `C` implements the member function `int f0 ()'. If you define `C::f0`, that definition takes precedence; otherwise, the default implementation `S::f0` applies.

Go to the previous, next section.

Go to the previous, next section.

Known Causes of Trouble with GNU CC

This section describes known problems that affect users of GNU CC. Most of these are not GNU CC bugs per se--if they were, we would fix them. But the result for a user may be like the result of a bug.

Some of these problems are due to bugs in other software, some are missing features that are too much work to add, and some are places where people's opinions differ as to what is best.

Actual Bugs We Haven't Fixed Yet

- The `fixincludes` script interacts badly with automounters; if the directory of system header files is automounted, it tends to be unmounted while `fixincludes` is running. This would seem to be a bug in the automounter. We don't know any good way to work around it.
- The `fixproto` script will sometimes add prototypes for the `sigsetjmp` and `siglongjmp` functions that reference the `jmp_buf` type before that type is defined. To work around this, edit the offending file and place the typedef in front of the prototypes.
- There are several obscure case of mis-using struct, union, and enum tags that are not detected as errors by the compiler.
- When `-pedantic-errors' is specified, GNU C will incorrectly give an error message when a function name is specified in an expression involving the comma operator.
- Loop unrolling doesn't work properly for certain C++ programs. This is a bug in the C++ front end. It sometimes emits incorrect debug info, and the loop unrolling code is unable to recover from this error.

Installation Problems

This is a list of problems (and some apparent problems which don't really mean anything is wrong) that show up during installation of GNU CC.

- On certain systems, defining certain environment variables such as `CC` can interfere with the functioning of `make`.
- If you encounter seemingly strange errors when trying to build the compiler in a directory other than the source directory, it could be because you have previously configured the compiler in the source directory. Make sure you have done all the necessary preparations. See section Compilation in a Separate Directory.
- If you build GNU CC on a BSD system using a directory stored in a System V file system, problems may occur in running `fixincludes` if the System V file system doesn't support symbolic links. These problems result in a failure to fix the declaration of `size_t` in `` `sys/types.h' ``. If you find that `size_t` is a signed type and that type mismatches occur, this could be the cause.

The solution is not to use such a directory for building GNU CC.

- In previous versions of GNU CC, the `gcc` driver program looked for `as` and `ld` in various places; for example, in files beginning with `/usr/local/lib/gcc-'. GNU CC version 2 looks for them in the directory `/usr/local/lib/gcc-lib/target/version'.

  Thus, to use a version of `as` or `ld` that is not the system default, for example `gas` or GNU `ld`, you must put them in that directory (or make links to them from that directory).
- Some commands executed when making the compiler may fail (return a non-zero status) and be ignored by `make`. These failures, which are often due to files that were not found, are expected, and can safely be ignored.
- It is normal to have warnings in compiling certain files about unreachable code and about enumeration type clashes. These files' names begin with `insn-'. Also, `real.c' may get some warnings that you can ignore.
- Sometimes `make` recompiles parts of the compiler when installing the compiler. In one case, this was traced down to a bug in `make`. Either ignore the problem or switch to GNU Make.
- If you have installed a program known as purify, you may find that it causes errors while linking `enquire`, which is part of building GNU CC. The fix is to get rid of the file `real-ld` which purify installs--so that GNU CC won't try to use it.
- On Linux SLS 1.01, there is a problem with `libc.a': it does not contain the obstack functions. However, GNU CC assumes that the obstack functions are in `libc.a' when it is the GNU C library. To work around this problem, change the __GNU_LIBRARY__ conditional around line 31 to `#if 1'.
- On some 386 systems, building the compiler never finishes because `enquire` hangs due to a hardware problem in the motherboard--it reports floating point exceptions to the kernel incorrectly. You can install GNU CC except for `float.h' by patching out the command to run `enquire`. You may also be able to fix the problem for real by getting a replacement motherboard. This problem was observed in Revision E of the Micronics motherboard, and is fixed in Revision F. It has also been observed in the MYLEX MXA-33 motherboard.

  If you encounter this problem, you may also want to consider removing the FPU from the socket during the compilation. Alternatively, if you are running SCO Unix, you can reboot and force the FPU to be ignored. To do this, type `hd(40)unix auto ignorefpu'.
- On some 386 systems, GNU CC crashes trying to compile `enquire.c'. This happens on machines that don't have a 387 FPU chip. On 386 machines, the system kernel is supposed to emulate the 387 when you don't have one. The crash is due to a bug in the emulator.

  One of these systems is the Unix from Interactive Systems: 386/ix. On this system, an alternate emulator is provided, and it does work. To use it, execute this command as super-user:

  ```
  ln /etc/emulator.rel1 /etc/emulator
  ```

  and then reboot the system. (The default emulator file remains present under the name `emulator.dflt'.)

  Try using `/etc/emulator.att', if you have such a problem on the SCO system.

Another system which has this problem is Esix. We don't know whether it has an alternate emulator that works.

On NetBSD 0.8, a similar problem manifests itself as these error messages:

```
enquire.c: In function `fprop':
enquire.c:2328: floating overflow
```

- On SCO systems, when compiling GNU CC with the system's compiler, do not use `-O'. Some versions of the system's compiler miscompile GNU CC with `-O'.
- Sometimes on a Sun 4 you may observe a crash in the program `genflags` or `genoutput` while building GNU CC. This is said to be due to a bug in `sh`. You can probably get around it by running `genflags` or `genoutput` manually and then retrying the `make`.
- On Solaris 2, executables of GNU CC version 2.0.2 are commonly available, but they have a bug that shows up when compiling current versions of GNU CC: undefined symbol errors occur during assembly if you use `-g'.

  The solution is to compile the current version of GNU CC without `-g'. That makes a working compiler which you can use to recompile with `-g'.
- Solaris 2 comes with a number of optional OS packages. Some of these packages are needed to use GNU CC fully. If you did not install all optional packages when installing Solaris, you will need to verify that the packages that GNU CC needs are installed.

  To check whether an optional package is installed, use the `pkginfo` command. To add an optional package, use the `pkgadd` command. For further details, see the Solaris documentation.

  For Solaris 2.0 and 2.1, GNU CC needs six packages: `SUNWarc', `SUNWbtool', `SUNWesu', `SUNWhea', `SUNWlibm', and `SUNWtoo'.

  For Solaris 2.2, GNU CC needs an additional seventh package: `SUNWsprot'.
- On Solaris 2, trying to use the linker and other tools in `` `/usr/ucb' `` to install GNU CC has been observed to cause trouble. For example, the linker may hang indefinitely. The fix is to remove `` `/usr/ucb' `` from your `PATH`.
- If you use the 1.31 version of the MIPS assembler (such as was shipped with Ultrix 3.1), you will need to use the -fno-delayed-branch switch when optimizing floating point code. Otherwise, the assembler will complain when the GCC compiler fills a branch delay slot with a floating point instruction, such as `add.d`.
- If on a MIPS system you get an error message saying "does not have gp sections for all it's [sic] sectons [sic]", don't worry about it. This happens whenever you use GAS with the MIPS linker, but there is not really anything wrong, and it is okay to use the output file. You can stop such warnings by installing the GNU linker.

  It would be nice to extend GAS to produce the gp tables, but they are optional, and there should not be a warning about their absence.
- In Ultrix 4.0 on the MIPS machine, `` `stdio.h' `` does not work with GNU CC at all unless it has been fixed with `fixincludes`. This causes problems in building GNU CC. Once GNU CC is

installed, the problems go away.

To work around this problem, when making the stage 1 compiler, specify this option to Make:

```
GCC_FOR_TARGET="./xgcc -B./ -I./include"
```

When making stage 2 and stage 3, specify this option:

```
CFLAGS="-g -I./include"
```

- Users have reported some problems with version 2.0 of the MIPS compiler tools that were shipped with Ultrix 4.1. Version 2.10 which came with Ultrix 4.2 seems to work fine.

  Users have also reported some problems with version 2.20 of the MIPS compiler tools that were shipped with RISC/os 4.x. The earlier version 2.11 seems to work fine.
- Some versions of the MIPS linker will issue an assertion failure when linking code that uses `alloca` against shared libraries on RISC-OS 5.0, and DEC's OSF/1 systems. This is a bug in the linker, that is supposed to be fixed in future revisions. To protect against this, GNU CC passes `-non_shared' to the linker unless you pass an explicit `-shared' or `-call_shared' switch.
- On System V release 3, you may get this error message while linking:

  ```
  ld fatal: failed to write symbol name something
   in strings table for file whatever
  ```

  This probably indicates that the disk is full or your ULIMIT won't allow the file to be as large as it needs to be.

  This problem can also result because the kernel parameter `MAXUMEM` is too small. If so, you must regenerate the kernel and make the value much larger. The default value is reported to be 1024; a value of 32768 is said to work. Smaller values may also work.
- On System V, if you get an error like this,

  ```
  /usr/local/lib/bison.simple: In function `yyparse':
  /usr/local/lib/bison.simple:625: virtual memory exhausted
  ```

  that too indicates a problem with disk space, ULIMIT, or `MAXUMEM`.
- Current GNU CC versions probably do not work on version 2 of the NeXT operating system.
- On NeXTStep 3.0, the Objective C compiler does not work, due, apparently, to a kernel bug that it happens to trigger. This problem does not happen on 3.1.
- On the Tower models 4n0 and 6n0, by default a process is not allowed to have more than one megabyte of memory. GNU CC cannot compile itself (or many other programs) with `-O' in that much memory.

  To solve this problem, reconfigure the kernel adding the following line to the configuration file:

  ```
  MAXUMEM = 4096
  ```

- On HP 9000 series 300 or 400 running HP-UX release 8.0, there is a bug in the assembler that must be fixed before GNU CC can be built. This bug manifests itself during the first stage of compilation, while building `libgcc2.a':

```
_floatdisf
cc1: warning: `-g' option not supported on this version of GCC
cc1: warning: `-g1' option not supported on this version of GCC
./xgcc: Internal compiler error: program as got fatal signal 11
```

  A patched version of the assembler is available by anonymous ftp from `altdorf.ai.mit.edu` as the file `archive/cph/hpux-8.0-assembler'. If you have HP software support, the patch can also be obtained directly from HP, as described in the following note:

  > This is the patched assembler, to patch SR#1653-010439, where the assembler aborts on floating point constants.
  >
  > The bug is not really in the assembler, but in the shared library version of the function "cvtnum(3c)". The bug on "cvtnum(3c)" is SR#4701-078451. Anyway, the attached assembler uses the archive library version of "cvtnum(3c)" and thus does not exhibit the bug.

  This patch is also known as PHCO_4484.
- On HP-UX version 8.05, but not on 8.07 or more recent versions, the `fixproto` shell script triggers a bug in the system shell. If you encounter this problem, upgrade your operating system or use BASH (the GNU shell) to run `fixproto`.
- Some versions of the Pyramid C compiler are reported to be unable to compile GNU CC. You must use an older version of GNU CC for bootstrapping. One indication of this problem is if you get a crash when GNU CC compiles the function `muldi3` in file `libgcc2.c'.

  You may be able to succeed by getting GNU CC version 1, installing it, and using it to compile GNU CC version 2. The bug in the Pyramid C compiler does not seem to affect GNU CC version 1.
- There may be similar problems on System V Release 3.1 on 386 systems.
- On the Intel Paragon (an i860 machine), if you are using operating system version 1.0, you will get warnings or errors about redefinition of `va_arg` when you build GNU CC.

  If this happens, then you need to link most programs with the library `iclib.a'. You must also modify `stdio.h' as follows: before the lines

```
#if     defined(__i860__) && !defined(_VA_LIST)
#include <va_list.h>
```

  insert the line

```
#if __PGC__
```

  and after the lines


```
extern int  vprintf(const char *, va_list );
extern int  vsprintf(char *, const char *, va_list );
#endif
```

insert the line

```
#endif /* __PGC__ */
```

These problems don't exist in operating system version 1.1.

- On the Altos 3068, programs compiled with GNU CC won't work unless you fix a kernel bug. This happens using system versions V.2.2 1.0gT1 and V.2.2 1.0e and perhaps later versions as well. See the file `README.ALTOS'.
- You will get several sorts of compilation and linking errors on the we32k if you don't follow the special instructions. See section Configurations Supported by GNU CC.
- A bug in the HP-UX 8.05 (and earlier) shell will cause the fixproto program to report an error of the form:

  ```
  ./fixproto: sh internal 1K buffer overflow
  ```

  To fix this, change the first line of the fixproto script to look like:

  ```
  #!/bin/ksh
  ```

Cross-Compiler Problems

You may run into problems with cross compilation on certain machines, for several reasons.

- Cross compilation can run into trouble for certain machines because some target machines' assemblers require floating point numbers to be written as *integer* constants in certain contexts.

  The compiler writes these integer constants by examining the floating point value as an integer and printing that integer, because this is simple to write and independent of the details of the floating point representation. But this does not work if the compiler is running on a different machine with an incompatible floating point format, or even a different byte-ordering.

  In addition, correct constant folding of floating point values requires representing them in the target machine's format. (The C standard does not quite require this, but in practice it is the only way to win.)

  It is now possible to overcome these problems by defining macros such as REAL_VALUE_TYPE. But doing so is a substantial amount of work for each target machine. See section Cross Compilation and Floating Point.

- At present, the program `mips-tfile' which adds debug support to object files on MIPS systems does not work in a cross compile environment.

Interoperation

This section lists various difficulties encountered in using GNU C or GNU C++ together with other compilers or with the assemblers, linkers, libraries and debuggers on certain systems.

- Objective C does not work on the RS/6000.
- GNU C++ does not do name mangling in the same way as other C++ compilers. This means that object files compiled with one compiler cannot be used with another.

  This effect is intentional, to protect you from more subtle problems. Compilers differ as to many internal details of C++ implementation, including: how class instances are laid out, how multiple inheritance is implemented, and how virtual function calls are handled. If the name encoding were made the same, your programs would link against libraries provided from other compilers--but the programs would then crash when run. Incompatible libraries are then detected at link time, rather than at run time.
- Older GDB versions sometimes fail to read the output of GNU CC version 2. If you have trouble, get GDB version 4.4 or later.
- DBX rejects some files produced by GNU CC, though it accepts similar constructs in output from PCC. Until someone can supply a coherent description of what is valid DBX input and what is not, there is nothing I can do about these problems. You are on your own.
- The GNU assembler (GAS) does not support PIC. To generate PIC code, you must use some other assembler, such as `` `/bin/as' ``.
- On some BSD systems, including some versions of Ultrix, use of profiling causes static variable destructors (currently used only in C++) not to be run.
- Use of `-I/usr/include' may cause trouble.

  Many systems come with header files that won't work with GNU CC unless corrected by `fixincludes`. The corrected header files go in a new directory; GNU CC searches this directory before `` `/usr/include' ``. If you use `-I/usr/include', this tells GNU CC to search `` `/usr/include' `` earlier on, before the corrected headers. The result is that you get the uncorrected header files.

  Instead, you should use these options (when compiling C programs):

  ```
  -I/usr/local/lib/gcc-lib/target/version/include -I/usr/include
  ```

  For C++ programs, GNU CC also uses a special directory that defines C++ interfaces to standard C subroutines. This directory is meant to be searched *before* other standard include directories, so that it takes precedence. If you are compiling C++ programs and specifying include directories explicitly, use this option first, then the two options above:

  ```
  -I/usr/local/lib/g++-include
  ```
- On some SGI systems, when you use `-lgl_s' as an option, it gets translated magically to `-lgl_s -lX11_s -lc_s'. Naturally, this does not happen when you use GNU CC. You must specify all three options explicitly.

- On a Sparc, GNU CC aligns all values of type `double` on an 8-byte boundary, and it expects every `double` to be so aligned. The Sun compiler usually gives `double` values 8-byte alignment, with one exception: function arguments of type `double` may not be aligned.

  As a result, if a function compiled with Sun CC takes the address of an argument of type `double` and passes this pointer of type `double *` to a function compiled with GNU CC, dereferencing the pointer may cause a fatal signal.

  One way to solve this problem is to compile your entire program with GNU CC. Another solution is to modify the function that is compiled with Sun CC to copy the argument into a local variable; local variables are always properly aligned. A third solution is to modify the function that uses the pointer to dereference it via the following function `access_double` instead of directly with `*':

```
inline double
access_double (double *unaligned_ptr)
{
  union d2i { double d; int i[2]; };

  union d2i *p = (union d2i *) unaligned_ptr;
  union d2i u;

  u.i[0] = p->i[0];
  u.i[1] = p->i[1];

  return u.d;
}
```

  Storing into the pointer can be done likewise with the same union.

- On Solaris, the `malloc` function in the `libmalloc.a' library may allocate memory that is only 4 byte aligned. Since GNU CC on the Sparc assumes that doubles are 8 byte aligned, this may result in a fatal signal if doubles are stored in memory allocated by the `libmalloc.a' library.

  The solution is to not use the `libmalloc.a' library. Use instead `malloc` and related functions from `libc.a'; they do not have this problem.

- Sun forgot to include a static version of `libdl.a' with some versions of SunOS (mainly 4.1). This results in undefined symbols when linking static binaries (that is, if you use `-static'). If you see undefined symbols `_dlclose`, `_dlsym` or `_dlopen` when linking, compile and link against the file `mit/util/misc/dlsym.c' from the MIT version of X windows.

- The 128-bit long double format that the Sparc port supports currently works by using the architecturally defined quad-word floating point instructions. Since there is no hardware that supports these instructions they must be emulated by the operating system. Long doubles do not work in Sun OS versions 4.0.3 and earlier, because the kernel eumulator uses an obsolete and incompatible format. Long doubles do not work in Sun OS version 4.1.1 due to a problem in a Sun library. Long doubles do work on Sun OS versions 4.1.2 and higher, but GNU CC does not enable them by default. Long doubles appear to work in Sun OS 5.x (Solaris 2.x).

- On HP-UX version 9.01 on the HP PA, the HP compiler `cc` does not compile GNU CC correctly.

We do not yet know why. However, GNU CC compiled on earlier HP-UX versions works properly on HP-UX 9.01 and can compile itself properly on 9.01.

- On the HP PA machine, ADB sometimes fails to work on functions compiled with GNU CC. Specifically, it fails to work on functions that use `alloca` or variable-size arrays. This is because GNU CC doesn't generate HP-UX unwind descriptors for such functions. It may even be impossible to generate them.
- Debugging (`-g') is not supported on the HP PA machine, unless you use the preliminary GNU tools (see section Installing GNU CC).
- Taking the address of a label may generate errors from the HP-UX PA assembler. GAS for the PA does not have this problem.
- Using floating point parameters for indirect calls to static functions will not work when using the HP assembler. There simply is no way for GCC to specify what registers hold arguments for static functions when using the HP assembler. GAS for the PA does not have this problem.
- In extremely rare cases involvving some very large functions you may receive errors from the HP linker complaining about an out of bounds unconditional branch offset. This used to occur more often in previous versions of GNU CC, but is now exceptionally rare. If you should run into it, you can work around by making your function smaller.
- GNU CC compiled code sometimes emits warnings from the HP-UX assembler of the form:

```
(warning) Use of GR3 when
  frame >= 8192 may cause conflict.
```

  These warnings are harmless and can be safely ignored.
- The current version of the assembler (`/bin/as') for the RS/6000 has certain problems that prevent the `-g' option in GCC from working. Note that `Makefile.in' uses `-g' by default when compiling `libgcc2.c'.

  IBM has produced a fixed version of the assembler. The upgraded assembler unfortunately was not included in any of the AIX 3.2 update PTF releases (3.2.2, 3.2.3, or 3.2.3e). Users of AIX 3.1 should request PTF U403044 from IBM and users of AIX 3.2 should request PTF U416277. See the file `README.RS6000' for more details on these updates.

  You can test for the presense of a fixed assembler by using the command

```
as -u < /dev/null
```

  If the command exits normally, the assembler fix already is installed. If the assembler complains that "-u" is an unknown flag, you need to order the fix.
- On the IBM RS/6000, compiling code of the form

```
extern int foo;

... foo ...
```


```
static int foo;
```

will cause the linker to report an undefined symbol `foo`. Although this behavior differs from most other systems, it is not a bug because redefining an `extern` variable as `static` is undefined in ANSI C.

- AIX on the RS/6000 provides support (NLS) for environments outside of the United States. Compilers and assemblers use NLS to support locale-specific representations of various objects including floating-point numbers ("." vs "," for separating decimal fractions). There have been problems reported where the library linked with GCC does not produce the same floating-point formats that the assembler accepts. If you have this problem, set the LANG environment variable to "C" or "En_US".
- Even if you specify `-fdollars-in-identifiers', you cannot successfully use `$' in identifiers on the RS/6000 due to a restriction in the IBM assembler. GAS supports these identifiers.
- On the RS/6000, XLC version 1.3.0.0 will miscompile `` `jump.c' ``. XLC version 1.3.0.1 or later fixes this problem. You can obtain XLC-1.3.0.2 by requesting PTF 421749 from IBM.
- There is an assembler bug in versions of DG/UX prior to 5.4.2.01 that occurs when the `fldcr' instruction is used. GNU CC uses `fldcr' on the 88100 to serialize volatile memory references. Use the option `-mno-serialize-volatile' if your version of the assembler has this bug.
- On VMS, GAS versions 1.38.1 and earlier may cause spurious warning messages from the linker. These warning messages complain of mismatched psect attributes. You can ignore them. See section Installing GNU CC on VMS.
- On NewsOS version 3, if you include both of the files `` `stddef.h' `` and `` `sys/types.h' ``, you get an error because there are two typedefs of `size_t`. You should change `` `sys/types.h' `` by adding these lines around the definition of `size_t`:

  ```
  #ifndef _SIZE_T
  #define _SIZE_T
  actual typedef here
  #endif
  ```

- On the Alliant, the system's own convention for returning structures and unions is unusual, and is not compatible with GNU CC no matter what options are used.
- On the IBM RT PC, the MetaWare HighC compiler (hc) uses a different convention for structure and union returning. Use the option `-mhc-struct-return' to tell GNU CC to use a convention compatible with it.
- On Ultrix, the Fortran compiler expects registers 2 through 5 to be saved by function calls. However, the C compiler uses conventions compatible with BSD Unix: registers 2 through 5 may be clobbered by function calls.

  GNU CC uses the same convention as the Ultrix C compiler. You can use these options to produce code compatible with the Fortran compiler:

  ```
  -fcall-saved-r2 -fcall-saved-r3 -fcall-saved-r4 -fcall-saved-r5
  ```

- On the WE32k, you may find that programs compiled with GNU CC do not work with the

standard shared C library. You may need to link with the ordinary C compiler. If you do so, yc must specify the following options:

```
-L/usr/local/lib/gcc-lib/we32k-att-sysv/2.7.0 -lgcc -lc_s
```

The first specifies where to find the library `libgcc.a' specified with the `-lgcc' option.

GNU CC does linking by invoking `ld`, just as `cc` does, and there is no reason why it *should* matter which compilation program you use to invoke `ld`. If someone tracks this problem down, it can probably be fixed easily.

- On the Alpha, you may get assembler errors about invalid syntax as a result of floating point constants. This is due to a bug in the C library functions `ecvt`, `fcvt` and `gcvt`. Given valid floating point numbers, they sometimes print `NaN'.
- On Irix 4.0.5F (and perhaps in some other versions), an assembler bug sometimes reorders instructions incorrectly when optimization is turned on. If you think this may be happening to you, try using the GNU assembler; GAS version 2.1 supports ECOFF on Irix.

  Or use the `-noasmopt' option when you compile GNU CC with itself, and then again when you compile your program. (This is a temporary kludge to turn off assembler optimization on Irix.) If this proves to be what you need, edit the assembler spec in the file `specs` so that it unconditionally passes `-O0' to the assembler, and never passes `-O2' or `-O3'.

Problems Compiling Certain Programs

Certain programs have problems compiling.

- Parse errors may occur compiling X11 on a Decstation running Ultrix 4.2 because of problems in DEC's versions of the X11 header files `X11/Xlib.h` and `X11/Xutil.h`. People recommend adding `-I/usr/include/mit' to use the MIT versions of the header files, using the `-traditional' switch to turn off ANSI C, or fixing the header files by adding this:

  ```
  #ifdef __STDC__
  #define NeedFunctionPrototypes 0
  #endif
  ```

- If you have trouble compiling Perl on a SunOS 4 system, it may be because Perl specifies `-I/usr/ucbinclude'. This accesses the unfixed header files. Perl specifies the options

  ```
  -traditional -Dvolatile=__volatile__
  -I/usr/include/sun -I/usr/ucbinclude
  -fpcc-struct-return
  ```

  most of which are unnecessary with GCC 2.4.5 and newer versions. You can make a properly working Perl by setting `ccflags` to `-fwritable-strings' (implied by the `-traditional' in the original options) and `cppflags` to empty in `config.sh`, then typing `./doSH; make depend; make'.

- On various 386 Unix systems derived from System V, including SCO, ISC, and ESIX, you ma get error messages about running out of virtual memory while compiling certain programs.

  You can prevent this problem by linking GNU CC with the GNU malloc (which thus replaces the malloc that comes with the system). GNU malloc is available as a separate package, and also in the file `src/gmalloc.c' in the GNU Emacs 19 distribution.

  If you have installed GNU malloc as a separate library package, use this option when you relink GNU CC:

  ```
  MALLOC=/usr/local/lib/libgmalloc.a
  ```

  Alternatively, if you have compiled `gmalloc.c' from Emacs 19, copy the object file to `gmalloc.o' and use this option when you relink GNU CC:

  ```
  MALLOC=gmalloc.o
  ```

Incompatibilities of GNU CC

There are several noteworthy incompatibilities between GNU C and most existing (non-ANSI) versions of C. The `-traditional' option eliminates many of these incompatibilities, *but not all*, by telling GNU C to behave like the other C compilers.

- GNU CC normally makes string constants read-only. If several identical-looking string constants are used, GNU CC stores only one copy of the string.

  One consequence is that you cannot call `mktemp` with a string constant argument. The function `mktemp` always alters the string its argument points to.

  Another consequence is that `sscanf` does not work on some systems when passed a string constant as its format control string or input. This is because `sscanf` incorrectly tries to write into the string constant. Likewise `fscanf` and `scanf`.

  The best solution to these problems is to change the program to use `char`-array variables with initialization strings for these purposes instead of string constants. But if this is not possible, you can use the `-fwritable-strings' flag, which directs GNU CC to handle string constants the same way most C compilers do. `-traditional' also has this effect, among others.
- `-2147483648` is positive.

  This is because 2147483648 cannot fit in the type `int`, so (following the ANSI C rules) its data type is `unsigned long int`. Negating this value yields 2147483648 again.
- GNU CC does not substitute macro arguments when they appear inside of string constants. For example, the following macro in GNU CC

  ```
  #define foo(a) "a"
  ```

  will produce output `"a"` regardless of what the argument a is.

The `-traditional' option directs GNU CC to handle such cases (among others) in the old-fashioned (non-ANSI) fashion.

- When you use `setjmp` and `longjmp`, the only automatic variables guaranteed to remain valid are those declared `volatile`. This is a consequence of automatic register allocation. Consider this function:

```
jmp_buf j;

foo ()
{
  int a, b;

  a = fun1 ();
  if (setjmp (j))
    return a;

  a = fun2 ();
  /* longjmp (j) may occur in fun3. */
  return a + fun3 ();
}
```

  Here `a` may or may not be restored to its first value when the `longjmp` occurs. If `a` is allocated in a register, then its first value is restored; otherwise, it keeps the last value stored in it.

  If you use the `-W' option with the `-O' option, you will get a warning when GNU CC thinks such a problem might be possible.

  The `-traditional' option directs GNU C to put variables in the stack by default, rather than in registers, in functions that call `setjmp`. This results in the behavior found in traditional C compilers.

- Programs that use preprocessing directives in the middle of macro arguments do not work with GNU CC. For example, a program like this will not work:

```
foobar (
#define luser
        hack)
```

  ANSI C does not permit such a construct. It would make sense to support it when `-traditional' is used, but it is too much work to implement.

- Declarations of external variables and functions within a block apply only to the block containing the declaration. In other words, they have the same scope as any other declaration in the same place.

  In some other C compilers, a `extern` declaration affects all the rest of the file even if it happens within a block.

The `-traditional' option directs GNU C to treat all `extern` declarations as global, like traditional compilers.

- In traditional C, you can combine `long`, etc., with a typedef name, as shown here:

```
typedef int foo;
typedef long foo bar;
```

  In ANSI C, this is not allowed: `long` and other type modifiers require an explicit `int`. Because this criterion is expressed by Bison grammar rules rather than C code, the `-traditional' flag cannot alter it.

- PCC allows typedef names to be used as function parameters. The difficulty described immediately above applies here too.
- PCC allows whitespace in the middle of compound assignment operators such as `+='. GNU CC, following the ANSI standard, does not allow this. The difficulty described immediately above applies here too.
- GNU CC complains about unterminated character constants inside of preprocessing conditionals that fail. Some programs have English comments enclosed in conditionals that are guaranteed to fail; if these comments contain apostrophes, GNU CC will probably report an error. For example, this code would produce an error:

```
#if 0
You can't expect this to work.
#endif
```

  The best solution to such a problem is to put the text into an actual C comment delimited by `/*...*/'. However, `-traditional' suppresses these error messages.

- Many user programs contain the declaration `long time ();'. In the past, the system header files on many systems did not actually declare `time`, so it did not matter what type your program declared it to return. But in systems with ANSI C headers, `time` is declared to return `time_t`, and if that is not the same as `long`, then `long time ();' is erroneous.

  The solution is to change your program to use `time_t` as the return type of `time`.

- When compiling functions that return `float`, PCC converts it to a double. GNU CC actually returns a `float`. If you are concerned with PCC compatibility, you should declare your functions to return `double`; you might as well say what you mean.
- When compiling functions that return structures or unions, GNU CC output code normally uses a method different from that used on most versions of Unix. As a result, code compiled with GNU CC cannot call a structure-returning function compiled with PCC, and vice versa.

  The method used by GNU CC is as follows: a structure or union which is 1, 2, 4 or 8 bytes long is returned like a scalar. A structure or union with any other size is stored into an address supplied by the caller (usually in a special, fixed register, but on some machines it is passed on the stack). The machine-description macros `STRUCT_VALUE` and `STRUCT_INCOMING_VALUE` tell GNU CC where to pass this address.

By contrast, PCC on most target machines returns structures and unions of any size by copying data into an area of static storage, and then returning the address of that storage as if it were a pointer value. The caller must copy the data from that memory area to the place where the value is wanted. GNU CC does not use this method because it is slower and nonreentrant.

On some newer machines, PCC uses a reentrant convention for all structure and union returning. GNU CC on most of these machines uses a compatible convention when returning structures and unions in memory, but still returns small structures and unions in registers.

You can tell GNU CC to use a compatible convention for all structure and union returning with the option `-fpcc-struct-return'.

- GNU C complains about program fragments such as `0x74ae-0x4000' which appear to be two hexadecimal constants separated by the minus operator. Actually, this string is a single preprocessing token. Each such token must correspond to one token in C. Since this does not, GNU C prints an error message. Although it may appear obvious that what is meant is an operator and two values, the ANSI C standard specifically requires that this be treated as erroneous.

  A preprocessing token is a preprocessing number if it begins with a digit and is followed by letters, underscores, digits, periods and `e+', `e-', `E+', or `E-' character sequences.

  To make the above program fragment valid, place whitespace in front of the minus sign. This whitespace will end the preprocessing number.

Fixed Header Files

GNU CC needs to install corrected versions of some system header files. This is because most target systems have some header files that won't work with GNU CC unless they are changed. Some have bugs, some are incompatible with ANSI C, and some depend on special features of other compilers.

Installing GNU CC automatically creates and installs the fixed header files, by running a program called `fixincludes` (or for certain targets an alternative such as `fixinc.svr4`). Normally, you don't need to pay attention to this. But there are cases where it doesn't do the right thing automatically.

- If you update the system's header files, such as by installing a new system version, the fixed header files of GNU CC are not automatically updated. The easiest way to update them is to reinstall GNU CC. (If you want to be clever, look in the makefile and you can find a shortcut.)
- On some systems, in particular SunOS 4, header file directories contain machine-specific symbolic links in certain places. This makes it possible to share most of the header files among hosts running the same version of SunOS 4 on different machine models.

  The programs that fix the header files do not understand this special way of using symbolic links; therefore, the directory of fixed header files is good only for the machine model used to build it.

  In SunOS 4, only programs that look inside the kernel will notice the difference between machine models. Therefore, for most purposes, you need not be concerned about this.

  It is possible to make separate sets of fixed header files for the different machine models, and arrange a structure of symbolic links so as to use the proper set, but you'll have to do this by hand.

- On Lynxos, GNU CC by default does not fix the header files. This is because bugs in the shell cause the `fixincludes` script to fail.

  This means you will encounter problems due to bugs in the system header files. It may be no comfort that they aren't GNU CC's fault, but it does mean that there's nothing for us to do about them.

Standard Libraries

GNU CC by itself attempts to be what the ISO/ANSI C standard calls a conforming freestanding implementation. This means all ANSI C language features are available, as well as the contents of `` `float.h' ``, `` `limits.h' ``, `` `stdarg.h' ``, and `` `stddef.h' ``. The rest of the C library is supplied by the vendor of the operating system. If that C library doesn't conform to the C standards, then your programs might get warnings (especially when using `-Wall') that you don't expect.

For example, the `sprintf` function on SunOS 4.1.3 returns `char *` while the C standard says that `sprintf` returns an `int`. The `fixincludes` program could make the prototype for this function match the Standard, but that would be wrong, since the function will still return `char *`.

If you need a Standard compliant library, then you need to find one, as GNU CC does not provide one. The GNU C library (called `glibc`) has been ported to a number of operating systems, and provides ANSI/ISO, POSIX, BSD and SystemV compatibility. You could also ask your operating system vendor if newer libraries are available.

Disappointments and Misunderstandings

These problems are perhaps regrettable, but we don't know any practical way around them.

- Certain local variables aren't recognized by debuggers when you compile with optimization.

  This occurs because sometimes GNU CC optimizes the variable out of existence. There is no way to tell the debugger how to compute the value such a variable "would have had", and it is not clear that would be desirable anyway. So GNU CC simply does not mention the eliminated variable when it writes debugging information.

  You have to expect a certain amount of disagreement between the executable and your source code, when you use optimization.

- Users often think it is a bug when GNU CC reports an error for code like this:

```
int foo (struct mumble *);

struct mumble { ... };

int foo (struct mumble *x)
{ ... }
```

This code really is erroneous, because the scope of `struct mumble` in the prototype is limi to the argument list containing it. It does not refer to the `struct mumble` defined with file scope immediately below--they are two unrelated types with similar names in different scopes.

But in the definition of `foo`, the file-scope type is used because that is available to be inherited. Thus, the definition and the prototype do not match, and you get an error.

This behavior may seem silly, but it's what the ANSI standard specifies. It is easy enough for you to make your code work by moving the definition of `struct mumble` above the prototype. It's not worth being incompatible with ANSI C just to avoid an error for the example shown above.

- Accesses to bitfields even in volatile objects works by accessing larger objects, such as a byte or a word. You cannot rely on what size of object is accessed in order to read or write the bitfield; it may even vary for a given bitfield according to the precise usage.

  If you care about controlling the amount of memory that is accessed, use volatile but do not use bitfields.

- GNU CC comes with shell scripts to fix certain known problems in system header files. They install corrected copies of various header files in a special directory where only GNU CC will normally look for them. The scripts adapt to various systems by searching all the system header files for the problem cases that we know about.

  If new system header files are installed, nothing automatically arranges to update the corrected header files. You will have to reinstall GNU CC to fix the new header files. More specifically, go to the build directory and delete the files `` `stmp-fixinc' `` and `` `stmp-headers' ``, and the subdirectory `include`; then do `make install' again.

- On 68000 systems, you can get paradoxical results if you test the precise values of floating point numbers. For example, you can find that a floating point value which is not a NaN is not equal to itself. This results from the fact that the the floating point registers hold a few more bits of precision than fit in a `double` in memory. Compiled code moves values between memory and floating point registers at its convenience, and moving them into memory truncates them.

  You can partially avoid this problem by using the `-ffloat-store' option (see section Options That Control Optimization).

- On the MIPS, variable argument functions using `` `varargs.h' `` cannot have a floating point value for the first argument. The reason for this is that in the absence of a prototype in scope, if the first argument is a floating point, it is passed in a floating point register, rather than an integer register.

  If the code is rewritten to use the ANSI standard `` `stdarg.h' `` method of variable arguments, and the prototype is in scope at the time of the call, everything will work fine.

Common Misunderstandings with GNU C++

C++ is a complex language and an evolving one, and its standard definition (the ANSI C++ draft standard) is also evolving. As a result, your C++ compiler may occasionally surprise you, even when its behavior is correct. This section discusses some areas that frequently give rise to questions of this sort.

Declare *and* Define Static Members

When a class has static data members, it is not enough to *declare* the static member; you must also *define* it. For example:

```
class Foo
{
  ...
  void method();
  static int bar;
};
```

This declaration only establishes that the class `Foo` has an `int` named `Foo::bar`, and a member function named `Foo::method`. But you still need to define *both* `method` and `bar` elsewhere. According to the draft ANSI standard, you must supply an initializer in one (and only one) source file, such as:

```
int Foo::bar = 0;
```

Other C++ compilers may not correctly implement the standard behavior. As a result, when you switch to `g++` from one of these compilers, you may discover that a program that appeared to work correctly in fact does not conform to the standard: `g++` reports as undefined symbols any static data members that lack definitions.

Temporaries May Vanish Before You Expect

It is dangerous to use pointers or references to *portions* of a temporary object. The compiler may very well delete the object before you expect it to, leaving a pointer to garbage. The most common place where this problem crops up is in classes like the libg++ `String` class, that define a conversion function to type `char *` or `const char *`. However, any class that returns a pointer to some internal structure is potentially subject to this problem.

For example, a program may use a function `strfunc` that returns `String` objects, and another function `charfunc` that operates on pointers to `char`:

```
String strfunc ();
void charfunc (const char *);
```

In this situation, it may seem natural to write `charfunc (strfunc ());' based on the knowledge that class `String` has an explicit conversion to `char` pointers. However, what really happens is akin to `charfunc (strfunc ().convert ());', where the `convert` method is a function to do the same data conversion normally performed by a cast. Since the last use of the temporary `String` object is the call to the conversion function, the compiler may delete that object before actually calling `charfunc`. The compiler has no way of knowing that deleting the `String` object will invalidate the pointer. The pointer then points to garbage, so that by the time `charfunc` is called, it gets an invalid argument.

Code like this may run successfully under some other compilers, especially those that delete tempora relatively late. However, the GNU C++ behavior is also standard-conformant, so if your program depends on late destruction of temporaries it is not portable.

If you think this is surprising, you should be aware that the ANSI C++ committee continues to debate the lifetime-of-temporaries problem.

For now, at least, the safe way to write such code is to give the temporary a name, which forces it to remain until the end of the scope of the name. For example:

```
String& tmp = strfunc ();
charfunc (tmp);
```

Caveats of using `protoize`

The conversion programs `protoize` and `unprotoize` can sometimes change a source file in a way that won't work unless you rearrange it.

- `protoize` can insert references to a type name or type tag before the definition, or in a file where they are not defined.

  If this happens, compiler error messages should show you where the new references are, so fixing the file by hand is straightforward.
- There are some C constructs which `protoize` cannot figure out. For example, it can't determine argument types for declaring a pointer-to-function variable; this you must do by hand. `protoize` inserts a comment containing `???' each time it finds such a variable; so you can find all such variables by searching for this string. ANSI C does not require declaring the argument types of pointer-to-function types.
- Using `unprotoize` can easily introduce bugs. If the program relied on prototypes to bring about conversion of arguments, these conversions will not take place in the program without prototypes. One case in which you can be sure `unprotoize` is safe is when you are removing prototypes that were made with `protoize`; if the program worked before without any prototypes, it will work again without them.

  You can find all the places where this problem might occur by compiling the program with the `-Wconversion' option. It prints a warning whenever an argument is converted.
- Both conversion programs can be confused if there are macro calls in and around the text to be converted. In other words, the standard syntax for a declaration or definition must not result from expanding a macro. This problem is inherent in the design of C and cannot be fixed. If only a few functions have confusing macro calls, you can easily convert them manually.
- `protoize` cannot get the argument types for a function whose definition was not actually compiled due to preprocessing conditionals. When this happens, `protoize` changes nothing in regard to such a function. `protoize` tries to detect such instances and warn about them.

  You can generally work around this problem by using `protoize` step by step, each time specifying a different set of `-D' options for compilation, until all of the functions have been

converted. There is no automatic way to verify that you have got them all, however.

- Confusion may result if there is an occasion to convert a function declaration or definition in a region of source code where there is more than one formal parameter list present. Thus, attempts to convert code containing multiple (conditionally compiled) versions of a single function header (in the same vicinity) may not produce the desired (or expected) results.

  If you plan on converting source files which contain such code, it is recommended that you first make sure that each conditionally compiled region of source code which contains an alternative function header also contains at least one additional follower token (past the final right parenthesis of the function header). This should circumvent the problem.
- `unprotoize` can become confused when trying to convert a function definition or declaration which contains a declaration for a pointer-to-function formal argument which has the same name as the function being defined or declared. We recommand you avoid such choices of formal parameter names.
- You might also want to correct some of the indentation by hand and break long lines. (The conversion programs don't write lines longer than eighty characters in any case.)

Certain Changes We Don't Want to Make

This section lists changes that people frequently request, but which we do not make because we think GNU CC is better without them.

- Checking the number and type of arguments to a function which has an old-fashioned definition and no prototype.

  Such a feature would work only occasionally--only for calls that appear in the same file as the called function, following the definition. The only way to check all calls reliably is to add a prototype for the function. But adding a prototype eliminates the motivation for this feature. So the feature is not worthwhile.
- Warning about using an expression whose type is signed as a shift count.

  Shift count operands are probably signed more often than unsigned. Warning about this would cause far more annoyance than good.
- Warning about assigning a signed value to an unsigned variable.

  Such assignments must be very common; warning about them would cause more annoyance than good.
- Warning about unreachable code.

  It's very common to have unreachable code in machine-generated programs. For example, this happens normally in some files of GNU C itself.
- Warning when a non-void function value is ignored.

  Coming as I do from a Lisp background, I balk at the idea that there is something dangerous about discarding a value. There are functions that return values which some callers may find useful; it makes no sense to clutter the program with a cast to `void` whenever the value isn't useful.

- Assuming (for optimization) that the address of an external symbol is never zero.

  This assumption is false on certain systems when `#pragma weak' is used.

- Making `-fshort-enums' the default.

  This would cause storage layout to be incompatible with most other C compilers. And it doesn't seem very important, given that you can get the same result in other ways. The case where it matters most is when the enumeration-valued object is inside a structure, and in that case you can specify a field width explicitly.

- Making bitfields unsigned by default on particular machines where "the ABI standard" says to do so.

  The ANSI C standard leaves it up to the implementation whether a bitfield declared plain `int` is signed or not. This in effect creates two alternative dialects of C.

  The GNU C compiler supports both dialects; you can specify the signed dialect with `-fsigned-bitfields' and the unsigned dialect with `-funsigned-bitfields'. However, this leaves open the question of which dialect to use by default.

  Currently, the preferred dialect makes plain bitfields signed, because this is simplest. Since `int` is the same as `signed int` in every other context, it is cleanest for them to be the same in bitfields as well.

  Some computer manufacturers have published Application Binary Interface standards which specify that plain bitfields should be unsigned. It is a mistake, however, to say anything about this issue in an ABI. This is because the handling of plain bitfields distinguishes two dialects of C. Both dialects are meaningful on every type of machine. Whether a particular object file was compiled using signed bitfields or unsigned is of no concern to other object files, even if they access the same bitfields in the same data structures.

  A given program is written in one or the other of these two dialects. The program stands a chance to work on most any machine if it is compiled with the proper dialect. It is unlikely to work at all if compiled with the wrong dialect.

  Many users appreciate the GNU C compiler because it provides an environment that is uniform across machines. These users would be inconvenienced if the compiler treated plain bitfields differently on certain machines.

  Occasionally users write programs intended only for a particular machine type. On these occasions, the users would benefit if the GNU C compiler were to support by default the same dialect as the other compilers on that machine. But such applications are rare. And users writing a program to run on more than one type of machine cannot possibly benefit from this kind of compatibility.

  This is why GNU CC does and will treat plain bitfields in the same fashion on all types of machines (by default).

  There are some arguments for making bitfields unsigned by default on all machines. If, for example, this becomes a universal de facto standard, it would make sense for GNU CC to go along

with it. This is something to be considered in the future.

(Of course, users strongly concerned about portability should indicate explicitly in each bitfield whether it is signed or not. In this way, they write programs which have the same meaning in both C dialects.)

- Undefining `__STDC__` when `-ansi' is not used.

  Currently, GNU CC defines `__STDC__` as long as you don't use `-traditional'. This provides good results in practice.

  Programmers normally use conditionals on `__STDC__` to ask whether it is safe to use certain features of ANSI C, such as function prototypes or ANSI token concatenation. Since plain `gcc' supports all the features of ANSI C, the correct answer to these questions is "yes".

  Some users try to use `__STDC__` to check for the availability of certain library facilities. This is actually incorrect usage in an ANSI C program, because the ANSI C standard says that a conforming freestanding implementation should define `__STDC__` even though it does not have the library facilities. `gcc -ansi -pedantic' is a conforming freestanding implementation, and it is therefore required to define `__STDC__`, even though it does not come with an ANSI C library.

  Sometimes people say that defining `__STDC__` in a compiler that does not completely conform to the ANSI C standard somehow violates the standard. This is illogical. The standard is a standard for compilers that claim to support ANSI C, such as `gcc -ansi'---not for other compilers such as plain `gcc'. Whatever the ANSI C standard says is relevant to the design of plain `gcc' without `-ansi' only for pragmatic reasons, not as a requirement.

- Undefining `__STDC__` in C++.

  Programs written to compile with C++-to-C translators get the value of `__STDC__` that goes with the C compiler that is subsequently used. These programs must test `__STDC__` to determine what kind of C preprocessor that compiler uses: whether they should concatenate tokens in the ANSI C fashion or in the traditional fashion.

  These programs work properly with GNU C++ if `__STDC__` is defined. They would not work otherwise.

  In addition, many header files are written to provide prototypes in ANSI C but not in traditional C. Many of these header files can work without change in C++ provided `__STDC__` is defined. If `__STDC__` is not defined, they will all fail, and will all need to be changed to test explicitly for C++ as well.

- Deleting "empty" loops.

  GNU CC does not delete "empty" loops because the most likely reason you would put one in a program is to have a delay. Deleting them will not make real programs run any faster, so it would be pointless.

  It would be different if optimization of a nonempty loop could produce an empty one. But this generally can't happen.

- Making side effects happen in the same order as in some other compiler.

It is never safe to depend on the order of evaluation of side effects. For example, a function ca like this may very well behave differently from one compiler to another:

```
void func (int, int);

int i = 2;
func (i++, i++);
```

There is no guarantee (in either the C or the C++ standard language definitions) that the increments will be evaluated in any particular order. Either increment might happen first. `func` might get the arguments `2, 3', or it might get `3, 2', or even `2, 2'.

- Not allowing structures with volatile fields in registers.

  Strictly speaking, there is no prohibition in the ANSI C standard against allowing structures with volatile fields in registers, but it does not seem to make any sense and is probably not what you wanted to do. So the compiler will give an error message in this case.

Warning Messages and Error Messages

The GNU compiler can produce two kinds of diagnostics: errors and warnings. Each kind has a different purpose:

- *Errors* report problems that make it impossible to compile your program. GNU CC reports errors with the source file name and line number where the problem is apparent.
- *Warnings* report other unusual conditions in your code that *may* indicate a problem, although compilation can (and does) proceed. Warning messages also report the source file name and line number, but include the text `warning:' to distinguish them from error messages.

Warnings may indicate danger points where you should check to make sure that your program really does what you intend; or the use of obsolete features; or the use of nonstandard features of GNU C or C++. Many warnings are issued only if you ask for them, with one of the `-W' options (for instance, `-Wall' requests a variety of useful warnings).

GNU CC always tries to compile your program if possible; it never gratuitously rejects a program whose meaning is clear merely because (for instance) it fails to conform to a standard. In some cases, however, the C and C++ standards specify that certain extensions are forbidden, and a diagnostic *must* be issued by a conforming compiler. The `-pedantic' option tells GNU CC to issue warnings in such cases; `-pedantic-errors' says to make them errors instead. This does not mean that *all* non-ANSI constructs get warnings or errors.

See section Options to Request or Suppress Warnings, for more detail on these and related command-line options.

Go to the previous, next section.

Go to the previous, next section.

Reporting Bugs

Your bug reports play an essential role in making GNU CC reliable.

When you encounter a problem, the first thing to do is to see if it is already known. See section Known Causes of Trouble with GNU CC. If it isn't known, then you should report the problem.

Reporting a bug may help you by bringing a solution to your problem, or it may not. (If it does not, look in the service directory; see section How To Get Help with GNU CC.) In any case, the principal function of a bug report is to help the entire community by making the next version of GNU CC work better. Bug reports are your contribution to the maintenance of GNU CC.

Since the maintainers are very overloaded, we cannot respond to every bug report. However, if the bug has not been fixed, we are likely to send you a patch and ask you to tell us whether it works.

In order for a bug report to serve its purpose, you must include the information that makes for fixing the bug.

Have You Found a Bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the compiler gets a fatal signal, for any input whatever, that is a compiler bug. Reliable compilers never crash.
- If the compiler produces invalid assembly code, for any input whatever (except an `asm` statement), that is a compiler bug, unless the compiler reports errors (not just warnings) which would ordinarily prevent the assembler from being run.
- If the compiler produces valid assembly code that does not correctly execute the input source code, that is a compiler bug.

  However, you must double-check to make sure, because you may have run into an incompatibility between GNU C and traditional C (see section Incompatibilities of GNU CC). These incompatibilities might be considered bugs, but they are inescapable consequences of valuable features.

  Or you may have a program whose behavior is undefined, which happened by chance to give the desired results with another C or C++ compiler.

  For example, in many nonoptimizing compilers, you can write `x;' at the end of a function instead of `return x;', with the same results. But the value of the function is undefined if `return` is omitted; it is not a bug when GNU CC produces different results.

  Problems often result from expressions with two increment operators, as in `f (*p++, *p++)`. Your previous compiler might have interpreted that expression the way you intended; GNU CC

might interpret it another way. Neither compiler is wrong. The bug is in your code.

After you have localized the error to a single source line, it should be easy to check for these things. If your program is correct and well defined, you have found a compiler bug.

- If the compiler produces an error message for valid input, that is a compiler bug.
- If the compiler does not produce an error message for invalid input, that is a compiler bug. However, you should note that your idea of "invalid input" might be my idea of "an extension" or "support for traditional practice".
- If you are an experienced user of C or C++ compilers, your suggestions for improvement of GNU CC or GNU C++ are welcome in any case.

Where to Report Bugs

Send bug reports for GNU C to `bug-gcc@prep.ai.mit.edu'.

Send bug reports for GNU C++ to `bug-g++@prep.ai.mit.edu'. If your bug involves the C++ class library libg++, send mail to `bug-lib-g++@prep.ai.mit.edu'. If you're not sure, you can send the bug report to both lists.

Do not send bug reports to `help-gcc@prep.ai.mit.edu' or to the newsgroup `gnu.gcc.help'. Most users of GNU CC do not want to receive bug reports. Those that do, have asked to be on `bug-gcc' and/or `bug-g++'.

The mailing lists `bug-gcc' and `bug-g++' both have newsgroups which serve as repeaters: `gnu.gcc.bug' and `gnu.g++.bug'. Each mailing list and its newsgroup carry exactly the same messages.

Often people think of posting bug reports to the newsgroup instead of mailing them. This appears to work, but it has one problem which can be crucial: a newsgroup posting does not contain a mail path back to the sender. Thus, if maintainers need more information, they may be unable to reach you. For this reason, you should always send bug reports by mail to the proper mailing list.

As a last resort, send bug reports on paper to:

```
GNU Compiler Bugs
Free Software Foundation
675 Mass Ave
Cambridge, MA 02139
```

How to Report Bugs

The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to state a fact or leave it out, state it!

Often people omit facts because they think they know what causes the problem and they conclude that some details don't matter. Thus, you might assume that the name of the variable you use in an example does not matter. Well, probably it doesn't, but one cannot be sure. Perhaps the bug is a stray memory

reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name were different, the contents of that location would fool the compiler into doing the right thing despite the bug. Play it safe and give a specific, complete example. That is the easiest thing for you to do, and the most helpful.

Keep in mind that the purpose of a bug report is to enable someone to fix the bug if it is not known. It isn't very important what happens if the bug is already known. Therefore, always write your bug reports on the assumption that the bug is not known.

Sometimes people give a few sketchy facts and ask, "Does this ring a bell?" This cannot help us fix a bug, so it is basically useless. We respond by asking for enough details to enable us to investigate. You might as well expedite matters by sending them to begin with.

Try to make your bug report self-contained. If we have to ask you for more information, it is best if you include all the previous information in your response, as well as the information that was missing.

Please report each bug in a separate message. This makes it easier for us to track which bugs have been fixed and to forward your bugs reports to the appropriate maintainer.

Do not compress and encode any part of your bug report using programs such as `uuencode'. If you do so it will slow down the processing of your bug. If you must submit multiple large files, use `shar', which allows us to read your message without having to run any decompression programs.

To enable someone to investigate the bug, you should include all these things:

- The version of GNU CC. You can get this by running it with the `-v' option.

  Without this, we won't know whether there is any point in looking for the bug in the current version of GNU CC.
- A complete input file that will reproduce the bug. If the bug is in the C preprocessor, send a source file and any header files that it requires. If the bug is in the compiler proper (`cc1'), run your source file through the C preprocessor by doing `gcc -E sourcefile > outfile', then include the contents of outfile in the bug report. (When you do this, use the same `-I', `-D' or `-U' options that you used in actual compilation.)

  A single statement is not enough of an example. In order to compile it, it must be embedded in a complete file of compiler input; and the bug might depend on the details of how this is done.

  Without a real example one can compile, all anyone can do about your bug report is wish you luck. It would be futile to try to guess how to provoke the bug. For example, bugs in register allocation and reloading frequently depend on every little detail of the function they happen in.

  Even if the input file that fails comes from a GNU program, you should still send the complete test case. Don't ask the GNU CC maintainers to do the extra work of obtaining the program in question--they are all overworked as it is. Also, the problem may depend on what is in the header files on your system; it is unreliable for the GNU CC maintainers to try the problem with the header files available to them. By sending CPP output, you can eliminate this source of uncertainty and save us a certain percentage of wild goose chases.
- The command arguments you gave GNU CC or GNU C++ to compile that example and observe

the bug. For example, did you use `-O'? To guarantee you won't omit something important, list the options.

If we were to try to guess the arguments, we would probably guess wrong and then we would not encounter the bug.

- The type of machine you are using, and the operating system name and version number.
- The operands you gave to the `configure` command when you installed the compiler.
- A complete list of any modifications you have made to the compiler source. (We don't promise to investigate the bug unless it happens in an unmodified compiler. But if you've made modifications and don't tell us, then you are sending us on a wild goose chase.)

  Be precise about these changes. A description in English is not enough--send a context diff for them.

  Adding files of your own (such as a machine description for a machine we don't support) is a modification of the compiler source.
- Details of any other deviations from the standard procedure for installing GNU CC.
- A description of what behavior you observe that you believe is incorrect. For example, "The compiler gets a fatal signal," or, "The assembler instruction at line 208 in the output is incorrect."

  Of course, if the bug is that the compiler gets a fatal signal, then one can't miss it. But if the bug is incorrect output, the maintainer might not notice unless it is glaringly wrong. None of us has time to study all the assembler code from a 50-line C program just on the chance that one instruction might be wrong. We need *you* to do this part!

  Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of the compiler is out of synch, or you have encountered a bug in the C library on your system. (This has happened!) Your copy might crash and the copy here would not. If you *said* to expect a crash, then when the compiler here fails to crash, we would know that the bug was not happening. If you don't say to expect a crash, then we would not know whether the bug was happening. We would not be able to draw any conclusion from our observations.

  If the problem is a diagnostic when compiling GNU CC with some other compiler, say whether it is a warning or an error.

  Often the observed symptom is incorrect output when your program is run. Sad to say, this is not enough information unless the program is short and simple. None of us has time to study a large program to figure out how it would work if compiled correctly, much less which line of it was compiled wrong. So you will have to do that. Tell us which source line it is, and what incorrect result happens when that line is executed. A person who understands the program can find this as easily as finding a bug in the program itself.
- If you send examples of assembler code output from GNU CC or GNU C++, please use `-g' when you make them. The debugging information includes source line numbers which are essential for correlating the output with the input.
- If you wish to mention something in the GNU CC source, refer to it by context, not by line

number.

The line numbers in the development sources don't match those in your sources. Your line numbers would convey no useful information to the maintainers.

- Additional information from a debugger might enable someone to find a problem on a machine which he does not have available. However, you need to think when you collect this information if you want it to have any chance of being useful.

  For example, many people send just a backtrace, but that is never useful by itself. A simple backtrace with arguments conveys little about GNU CC because the compiler is largely data-driven; the same functions are called over and over for different RTL insns, doing different things depending on the details of the insn.

  Most of the arguments listed in the backtrace are useless because they are pointers to RTL list structure. The numeric values of the pointers, which the debugger prints in the backtrace, have no significance whatever; all that matters is the contents of the objects they point to (and most of the contents are other such pointers).

  In addition, most compiler passes consist of one or more loops that scan the RTL insn sequence. The most vital piece of information about such a loop--which insn it has reached--is usually in a local variable, not in an argument.

  What you need to provide in addition to a backtrace are the values of the local variables for several stack frames up. When a local variable or an argument is an RTX, first print its value and then use the GDB command `pr` to print the RTL expression that it points to. (If GDB doesn't run on your machine, use your debugger to call the function `debug_rtx` with the RTX as an argument.) In general, whenever a variable is a pointer, its value is no use without the data it points to.

Here are some things that are not necessary:

- A description of the envelope of the bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

  This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. You might as well save your time for something else.

  Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience. Errors in the output will be easier to spot, running under the debugger will take less time, etc. Most GNU CC bugs involve just one function, so the most straightforward way to simplify an example is to delete all the function definitions except the one where the bug occurs. Those earlier in the file may be replaced by external declarations if the crucial function depends on them. (Exception: inline functions may affect compilation of functions defined later in the file.)

  However, simplification is not vital; if you don't want to do this, report the bug anyway and send the entire test case you used.

- In particular, some people insert conditionals `#ifdef BUG' around a statement which, if removed,

makes the bug not happen. These are just clutter; we won't pay any attention to them anyway. Besides, you should send us cpp output, and that can't have conditionals.

- A patch for the bug.

  A patch for the bug is useful if it is a good one. But don't omit the necessary information, such as the test case, on the assumption that a patch is all we need. We might see problems with your patch and decide to fix the problem another way, or we might not understand it at all.

  Sometimes with a program as complicated as GNU CC it is very hard to construct an example that will make the program follow a certain path through the code. If you don't send the example, we won't be able to construct one, so we won't be able to verify that the bug is fixed.

  And if we can't understand what bug you are trying to fix, or why your patch should be an improvement, we won't install it. A test case will help us to understand.

  See section Sending Patches for GNU CC, for guidelines on how to make it easy for us to understand and install your patches.

- A guess about what the bug is or what it depends on.

  Such guesses are usually wrong. Even I can't guess right about such things without first using the debugger to find the facts.

- A core dump file.

  We have no way of examining a core dump for your type of machine unless we have an identical system--and if we do have one, we should be able to reproduce the crash ourselves.

Sending Patches for GNU CC

If you would like to write bug fixes or improvements for the GNU C compiler, that is very helpful. Send suggested fixes to the bug report mailing list, `bug-gcc@prep.ai.mit.edu`.

Please follow these guidelines so we can study your patches efficiently. If you don't follow these guidelines, your information might still be useful, but using it will take extra work. Maintaining GNU C is a lot of work in the best of circumstances, and we can't keep up unless you do your best to help.

- Send an explanation with your changes of what problem they fix or what improvement they bring about. For a bug fix, just include a copy of the bug report, and explain why the change fixes the bug.

  (Referring to a bug report is not as good as including it, because then we will have to look it up, and we have probably already deleted it if we've already fixed the bug.)

- Always include a proper bug report for the problem you think you have fixed. We need to convince ourselves that the change is right before installing it. Even if it is right, we might have trouble judging it if we don't have a way to reproduce the problem.
- Include all the comments that are appropriate to help people reading the source in the future understand why this change was needed.
- Don't mix together changes made for different reasons. Send them *individually*.

If you make two changes for separate reasons, then we might not want to install them both. We might want to install just one. If you send them all jumbled together in a single set of diffs, we have to do extra work to disentangle them--to figure out which parts of the change serve which purpose. If we don't have time for this, we might have to ignore your changes entirely.

If you send each change as soon as you have written it, with its own explanation, then the two changes never get tangled up, and we can consider each one properly without any extra work to disentangle them.

Ideally, each change you send should be impossible to subdivide into parts that we might want to consider separately, because each of its parts gets its motivation from the other parts.

- Send each change as soon as that change is finished. Sometimes people think they are helping us by accumulating many changes to send them all together. As explained above, this is absolutely the worst thing you could do.

  Since you should send each change separately, you might as well send it right away. That gives us the option of installing it immediately if it is important.

- Use `diff -c' to make your diffs. Diffs without context are hard for us to install reliably. More than that, they make it hard for us to study the diffs to decide whether we want to install them. Unidiff format is better than contextless diffs, but not as easy to read as `-c' format.

  If you have GNU diff, use `diff -cp', which shows the name of the function that each change occurs in.

- Write the change log entries for your changes. We get lots of changes, and we don't have time to do all the change log writing ourselves.

  Read the `ChangeLog' file to see what sorts of information to put in, and to learn the style that we use. The purpose of the change log is to show people where to find what was changed. So you need to be specific about what functions you changed; in large functions, it's often helpful to indicate where within the function the change was.

  On the other hand, once you have shown people where to find the change, you need not explain its purpose. Thus, if you add a new function, all you need to say about it is that it is new. If you feel that the purpose needs explaining, it probably does--but the explanation will be much more useful if you put it in comments in the code.

  If you would like your name to appear in the header line for who made the change, send us the header line.

- When you write the fix, keep in mind that we can't install a change that would break other systems.

  People often suggest fixing a problem by changing machine-independent files such as `toplev.c' to do something special that a particular system needs. Sometimes it is totally obvious that such changes would break GNU CC for almost all users. We can't possibly make a change like that. At best it might tell us how to write another patch that would solve the problem acceptably.

  Sometimes people send fixes that *might* be an improvement in general--but it is hard to be sure of

this. It's hard to install such changes because we have to study them very carefully. Of course, a good explanation of the reasoning by which you concluded the change was correct can help convince us.

The safest changes are changes to the configuration files for a particular machine. These are safe because they can't create new bugs on other machines.

Please help us keep up with the workload by designing the patch in a form that is good to install.

Go to the previous, next section.

Go to the previous, next section.

How To Get Help with GNU CC

If you need help installing, using or changing GNU CC, there are two ways to find it:

- Send a message to a suitable network mailing list. First try `bug-gcc@prep.ai.mit.edu`, and if that brings no response, try `help-gcc@prep.ai.mit.edu`.
- Look in the service directory for someone who might help you for a fee. The service directory is found in the file named `` `SERVICE' `` in the GNU CC distribution.

Go to the previous, next section.

Go to the previous, next section.

Using GNU CC on VMS

Here is how to use GNU CC on VMS.

Include Files and VMS

Due to the differences between the filesystems of Unix and VMS, GNU CC attempts to translate file names in `#include' into names that VMS will understand. The basic strategy is to prepend a prefix to the specification of the include file, convert the whole filename to a VMS filename, and then try to open the file. GNU CC tries various prefixes one by one until one of them succeeds:

1. The first prefix is the `GNU_CC_INCLUDE:' logical name: this is where GNU C header files are traditionally stored. If you wish to store header files in non-standard locations, then you can assign the logical `GNU_CC_INCLUDE' to be a search list, where each element of the list is suitable for use with a rooted logical.
2. The next prefix tried is `SYS$SYSROOT:[SYSLIB.]'. This is where VAX-C header files are traditionally stored.
3. If the include file specification by itself is a valid VMS filename, the preprocessor then uses this name with no prefix in an attempt to open the include file.
4. If the file specification is not a valid VMS filename (i.e. does not contain a device or a directory specifier, and contains a `/' character), the preprocessor tries to convert it from Unix syntax to VMS syntax.

   Conversion works like this: the first directory name becomes a device, and the rest of the directories are converted into VMS-format directory names. For example, the name `X11/foobar.h' is translated to `X11:[000000]foobar.h' or `X11:foobar.h', whichever one can be opened. This strategy allows you to assign a logical name to point to the actual location of the header files.
5. If none of these strategies succeeds, the `#include' fails.

Include directives of the form:

```
#include foobar
```

are a common source of incompatibility between VAX-C and GNU CC. VAX-C treats this much like a standard `#include <foobar.h>` directive. That is incompatible with the ANSI C behavior implemented by GNU CC: to expand the name `foobar` as a macro. Macro expansion should eventually yield one of the two standard formats for `#include`:

```
#include "file"
#include <file>
```

If you have this problem, the best solution is to modify the source to convert the `#include` directiv one of the two standard forms. That will work with either compiler. If you want a quick and dirty fix, define the file names as macros with the proper expansion, like this:

```
#define stdio <stdio.h>
```

This will work, as long as the name doesn't conflict with anything else in the program.

Another source of incompatibility is that VAX-C assumes that:

```
#include "foobar"
```

is actually asking for the file `` `foobar.h' ``. GNU CC does not make this assumption, and instead takes what you ask for literally; it tries to read the file `` `foobar' ``. The best way to avoid this problem is to always specify the desired file extension in your include directives.

GNU CC for VMS is distributed with a set of include files that is sufficient to compile most general purpose programs. Even though the GNU CC distribution does not contain header files to define constants and structures for some VMS system-specific functions, there is no reason why you cannot use GNU CC with any of these functions. You first may have to generate or create header files, either by using the public domain utility `UNSDL` (which can be found on a DECUS tape), or by extracting the relevant modules from one of the system macro libraries, and using an editor to construct a C header file.

A `#include` file name cannot contain a DECNET node name. The preprocessor reports an I/O error if you attempt to use a node name, whether explicitly, or implicitly via a logical name.

Global Declarations and VMS

GNU CC does not provide the `globalref`, `globaldef` and `globalvalue` keywords of VAX-C. You can get the same effect with an obscure feature of GAS, the GNU assembler. (This requires GAS version 1.39 or later.) The following macros allow you to use this feature in a fairly natural way:

```
#ifdef __GNUC__
#define GLOBALREF(TYPE,NAME)                            \
  TYPE NAME                                             \
  asm ("_$$PsectAttributes_GLOBALSYMBOL$$" #NAME)
#define GLOBALDEF(TYPE,NAME,VALUE)                      \
  TYPE NAME                                             \
  asm ("_$$PsectAttributes_GLOBALSYMBOL$$" #NAME)       \
    = VALUE
#define GLOBALVALUEREF(TYPE,NAME)                       \
  const TYPE NAME[1]                                    \
  asm ("_$$PsectAttributes_GLOBALVALUE$$" #NAME)
#define GLOBALVALUEDEF(TYPE,NAME,VALUE)                 \
  const TYPE NAME[1]                                    \
  asm ("_$$PsectAttributes_GLOBALVALUE$$" #NAME)        \
```


```
    = {VALUE}
#else
#define GLOBALREF(TYPE,NAME) \
  globalref TYPE NAME
#define GLOBALDEF(TYPE,NAME,VALUE) \
  globaldef TYPE NAME = VALUE
#define GLOBALVALUEDEF(TYPE,NAME,VALUE) \
  globalvalue TYPE NAME = VALUE
#define GLOBALVALUEREF(TYPE,NAME) \
  globalvalue TYPE NAME
#endif
```

(The `_$$PsectAttributes_GLOBALSYMBOL` prefix at the start of the name is removed by the assembler, after it has modified the attributes of the symbol). These macros are provided in the VMS binaries distribution in a header file `` `GNU_HACKS.H' ``. An example of the usage is:

```
GLOBALREF (int, ijk);
GLOBALDEF (int, jkl, 0);
```

The macros `GLOBALREF` and `GLOBALDEF` cannot be used straightforwardly for arrays, since there is no way to insert the array dimension into the declaration at the right place. However, you can declare an array with these macros if you first define a typedef for the array type, like this:

```
typedef int intvector[10];
GLOBALREF (intvector, foo);
```

Array and structure initializers will also break the macros; you can define the initializer to be a macro of its own, or you can expand the `GLOBALDEF` macro by hand. You may find a case where you wish to use the `GLOBALDEF` macro with a large array, but you are not interested in explicitly initializing each element of the array. In such cases you can use an initializer like: `{0,}`, which will initialize the entire array to `0`.

A shortcoming of this implementation is that a variable declared with `GLOBALVALUEREF` or `GLOBALVALUEDEF` is always an array. For example, the declaration:

```
GLOBALVALUEREF(int, ijk);
```

declares the variable `ijk` as an array of type `int [1]`. This is done because a globalvalue is actually a constant; its "value" is what the linker would normally consider an address. That is not how an integer value works in C, but it is how an array works. So treating the symbol as an array name gives consistent results--with the exception that the value seems to have the wrong type. **Don't try to access an element of the array.** It doesn't have any elements. The array "address" may not be the address of actual storage.

The fact that the symbol is an array may lead to warnings where the variable is used. Insert type casts to avoid the warnings. Here is an example; it takes advantage of the ANSI C feature allowing macros that expand to use the same name as the macro itself.


```
GLOBALVALUEREF (int, ss$_normal);
GLOBALVALUEDEF (int, xyzzy,123);
#ifdef __GNUC__
#define ss$_normal ((int) ss$_normal)
#define xyzzy ((int) xyzzy)
#endif
```

Don't use `globaldef` or `globalref` with a variable whose type is an enumeration type; this is not implemented. Instead, make the variable an integer, and use a `globalvaluedef` for each of the enumeration values. An example of this would be:

```
#ifdef __GNUC__
GLOBALDEF (int, color, 0);
GLOBALVALUEDEF (int, RED, 0);
GLOBALVALUEDEF (int, BLUE, 1);
GLOBALVALUEDEF (int, GREEN, 3);
#else
enum globaldef color {RED, BLUE, GREEN = 3};
#endif
```

Other VMS Issues

GNU CC automatically arranges for `main` to return 1 by default if you fail to specify an explicit return value. This will be interpreted by VMS as a status code indicating a normal successful completion. Version 1 of GNU CC did not provide this default.

GNU CC on VMS works only with the GNU assembler, GAS. You need version 1.37 or later of GAS in order to produce value debugging information for the VMS debugger. Use the ordinary VMS linker with the object files produced by GAS.

Under previous versions of GNU CC, the generated code would occasionally give strange results when linked to the sharable `` `VAXCRTL' `` library. Now this should work.

A caveat for use of `const` global variables: the `const` modifier must be specified in every external declaration of the variable in all of the source files that use that variable. Otherwise the linker will issue warnings about conflicting attributes for the variable. Your program will still work despite the warnings, but the variable will be placed in writable storage.

Although the VMS linker does distinguish between upper and lower case letters in global symbols, most VMS compilers convert all such symbols into upper case and most run-time library routines also have upper case names. To be able to reliably call such routines, GNU CC (by means of the assembler GAS) converts global symbols into upper case like other VMS compilers. However, since the usual practice in C is to distinguish case, GNU CC (via GAS) tries to preserve usual C behavior by augmenting each name that is not all lower case. This means truncating the name to at most 23 characters and then adding more characters at the end which encode the case pattern of those 23. Names which contain at least one dollar sign are an exception; they are converted directly into upper case without augmentation.

Name augmentation yields bad results for programs that use precompiled libraries (such as Xlib) which were generated by another compiler. You can use the compiler option `/NOCASE_HACK' to inhibit augmentation; it makes external C functions and variables case-independent as is usual on VMS. Alternatively, you could write all references to the functions and variables in such libraries using lower case; this will work on VMS, but is not portable to other systems. The compiler option `/NAMES' also provides control over global name handling.

Function and variable names are handled somewhat differently with GNU C++. The GNU C++ compiler performs name mangling on function names, which means that it adds information to the function name to describe the data types of the arguments that the function takes. One result of this is that the name of a function can become very long. Since the VMS linker only recognizes the first 31 characters in a name, special action is taken to ensure that each function and variable has a unique name that can be represented in 31 characters.

If the name (plus a name augmentation, if required) is less than 32 characters in length, then no special action is performed. If the name is longer than 31 characters, the assembler (GAS) will generate a hash string based upon the function name, truncate the function name to 23 characters, and append the hash string to the truncated name. If the `/VERBOSE' compiler option is used, the assembler will print both the full and truncated names of each symbol that is truncated.

The `/NOCASE_HACK' compiler option should not be used when you are compiling programs that use libg++. libg++ has several instances of objects (i.e. `Filebuf` and `filebuf`) which become indistinguishable in a case-insensitive environment. This leads to cases where you need to inhibit augmentation selectively (if you were using libg++ and Xlib in the same program, for example). There is no special feature for doing this, but you can get the result by defining a macro for each mixed case symbol for which you wish to inhibit augmentation. The macro should expand into the lower case equivalent of itself. For example:

```
#define StuDlyCapS studlycaps
```

These macro definitions can be placed in a header file to minimize the number of changes to your source code.

Go to the previous, next section.

Go to the previous, next section.

GNU CC and Portability

The main goal of GNU CC was to make a good, fast compiler for machines in the class that the GNU system aims to run on: 32-bit machines that address 8-bit bytes and have several general registers. Elegance, theoretical power and simplicity are only secondary.

GNU CC gets most of the information about the target machine from a machine description which gives an algebraic formula for each of the machine's instructions. This is a very clean way to describe the target. But when the compiler needs information that is difficult to express in this fashion, I have not hesitated to define an ad-hoc parameter to the machine description. The purpose of portability is to reduce the total work needed on the compiler; it was not of interest for its own sake.

GNU CC does not contain machine dependent code, but it does contain code that depends on machine parameters such as endianness (whether the most significant byte has the highest or lowest address of the bytes in a word) and the availability of autoincrement addressing. In the RTL-generation pass, it is often necessary to have multiple strategies for generating code for a particular kind of syntax tree, strategies that are usable for different combinations of parameters. Often I have not tried to address all possible cases, but only the common ones or only the ones that I have encountered. As a result, a new target may require additional strategies. You will know if this happens because the compiler will call `abort`. Fortunately, the new strategies can be added in a machine-independent fashion, and will affect only the target machines that need them.

Go to the previous, next section.

Go to the previous, next section.

Interfacing to GNU CC Output

GNU CC is normally configured to use the same function calling convention normally in use on the target system. This is done with the machine-description macros described (see section Target Description Macros).

However, returning of structure and union values is done differently on some target machines. As a result, functions compiled with PCC returning such types cannot be called from code compiled with GNU CC, and vice versa. This does not cause trouble often because few Unix library routines return structures or unions.

GNU CC code returns structures and unions that are 1, 2, 4 or 8 bytes long in the same registers used for `int` or `double` return values. (GNU CC typically allocates variables of such types in registers also.) Structures and unions of other sizes are returned by storing them into an address passed by the caller (usually in a register). The machine-description macros `STRUCT_VALUE` and `STRUCT_INCOMING_VALUE` tell GNU CC where to pass this address.

By contrast, PCC on most target machines returns structures and unions of any size by copying the data into an area of static storage, and then returning the address of that storage as if it were a pointer value. The caller must copy the data from that memory area to the place where the value is wanted. This is slower than the method used by GNU CC, and fails to be reentrant.

On some target machines, such as RISC machines and the 80386, the standard system convention is to pass to the subroutine the address of where to return the value. On these machines, GNU CC has been configured to be compatible with the standard compiler, when this method is used. It may not be compatible for structures of 1, 2, 4 or 8 bytes.

GNU CC uses the system's standard convention for passing arguments. On some machines, the first few arguments are passed in registers; in others, all are passed on the stack. It would be possible to use registers for argument passing on any machine, and this would probably result in a significant speedup. But the result would be complete incompatibility with code that follows the standard convention. So this change is practical only if you are switching to GNU CC as the sole C compiler for the system. We may implement register argument passing on certain machines once we have a complete GNU system so that we can compile the libraries with GNU CC.

On some machines (particularly the Sparc), certain types of arguments are passed "by invisible reference". This means that the value is stored in memory, and the address of the memory location is passed to the subroutine.

If you use `longjmp`, beware of automatic variables. ANSI C says that automatic variables that are not declared `volatile` have undefined values after a `longjmp`. And this is all GNU CC promises to do, because it is very difficult to restore register variables correctly, and one of GNU CC's features is that it can put variables in registers without your asking it to.

If you want a variable to be unaltered by `longjmp`, and you don't want to write `volatile` because old

C compilers don't accept it, just take the address of the variable. If a variable's address is ever taken, even if just to compute it and ignore it, then the variable cannot go in a register:

```
{
  int careful;
  &careful;
  ...
}
```

Code compiled with GNU CC may call certain library routines. Most of them handle arithmetic for which there are no instructions. This includes multiply and divide on some machines, and floating point operations on any machine for which floating point support is disabled with `-msoft-float'. Some standard parts of the C library, such as `bcopy` or `memcpy`, are also called automatically. The usual function call interface is used for calling the library routines.

These library routines should be defined in the library `` `libgcc.a' ``, which GNU CC automatically searches whenever it links a program. On machines that have multiply and divide instructions, if hardware floating point is in use, normally `` `libgcc.a' `` is not needed, but it is searched just in case.

Each arithmetic function is defined in `` `libgcc1.c' `` to use the corresponding C arithmetic operator. As long as the file is compiled with another C compiler, which supports all the C arithmetic operators, this file will work portably. However, `` `libgcc1.c' `` does not work if compiled with GNU CC, because each arithmetic function would compile into a call to itself!

Go to the previous, next section.

Go to the previous, next section.

Passes and Files of the Compiler

The overall control structure of the compiler is in `toplev.c'. This file is responsible for initialization, decoding arguments, opening and closing files, and sequencing the passes.

The parsing pass is invoked only once, to parse the entire input. The RTL intermediate code for a function is generated as the function is parsed, a statement at a time. Each statement is read in as a syntax tree and then converted to RTL; then the storage for the tree for the statement is reclaimed. Storage for types (and the expressions for their sizes), declarations, and a representation of the binding contours and how they nest, remain until the function is finished being compiled; these are all needed to output the debugging information.

Each time the parsing pass reads a complete function definition or top-level declaration, it calls either the function `rest_of_compilation`, or the function `rest_of_decl_compilation` in `toplev.c', which are responsible for all further processing necessary, ending with output of the assembler language. All other compiler passes run, in sequence, within `rest_of_compilation`. When that function returns from compiling a function definition, the storage used for that function definition's compilation is entirely freed, unless it is an inline function (see section An Inline Function is As Fast As a Macro).

Here is a list of all the passes of the compiler and their source files. Also included is a description of where debugging dumps can be requested with `-d' options.

- Parsing. This pass reads the entire text of a function definition, constructing partial syntax trees. This and RTL generation are no longer truly separate passes (formerly they were), but it is easier to think of them as separate.

  The tree representation does not entirely follow C syntax, because it is intended to support other languages as well.

  Language-specific data type analysis is also done in this pass, and every tree node that represents an expression has a data type attached. Variables are represented as declaration nodes.

  Constant folding and some arithmetic simplifications are also done during this pass.

  The language-independent source files for parsing are `stor-layout.c', `fold-const.c', and `tree.c'. There are also header files `tree.h' and `tree.def' which define the format of the tree representation.

  The source files to parse C are `c-parse.in', `c-decl.c', `c-typeck.c', `c-aux-info.c', `c-convert.c', and `c-lang.c' along with header files `c-lex.h', and `c-tree.h'.

  The source files for parsing C++ are `cp-parse.y', `cp-class.c', `cp-cvt.c', `cp-decl.c', `cp-decl2.c', `cp-dem.c', `cp-except.c', `cp-expr.c', `cp-init.c', `cp-lex.c', `cp-method.c', `cp-ptree.c',

`cp-search.c', `cp-tree.c', `cp-type2.c', and `cp-typeck.c', along with header files `cp-tree.def', `cp-tree.h', and `cp-decl.h'.

The special source files for parsing Objective C are `objc-parse.y', `objc-actions.c', `objc-tree.def', and `objc-actions.h'. Certain C-specific files are used for this as well.

The file `c-common.c' is also used for all of the above languages.

- RTL generation. This is the conversion of syntax tree into RTL code. It is actually done statement-by-statement during parsing, but for most purposes it can be thought of as a separate pass.

  This is where the bulk of target-parameter-dependent code is found, since often it is necessary for strategies to apply only when certain standard kinds of instructions are available. The purpose of named instruction patterns is to provide this information to the RTL generation pass.

  Optimization is done in this pass for `if`-conditions that are comparisons, boolean operations or conditional expressions. Tail recursion is detected at this time also. Decisions are made about how best to arrange loops and how to output `switch` statements.

  The source files for RTL generation include `stmt.c', `calls.c', `expr.c', `explow.c', `expmed.c', `function.c', `optabs.c' and `emit-rtl.c'. Also, the file `insn-emit.c', generated from the machine description by the program `genemit`, is used in this pass. The header file `expr.h' is used for communication within this pass.

  The header files `insn-flags.h' and `insn-codes.h', generated from the machine description by the programs `genflags` and `gencodes`, tell this pass which standard names are available for use and which patterns correspond to them.

  Aside from debugging information output, none of the following passes refers to the tree structure representation of the function (only part of which is saved).

  The decision of whether the function can and should be expanded inline in its subsequent callers is made at the end of rtl generation. The function must meet certain criteria, currently related to the size of the function and the types and number of parameters it has. Note that this function may contain loops, recursive calls to itself (tail-recursive functions can be inlined!), gotos, in short, all constructs supported by GNU CC. The file `integrate.c' contains the code to save a function's rtl for later inlining and to inline that rtl when the function is called. The header file `integrate.h' is also used for this purpose.

  The option `-dr' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending `.rtl' to the input file name.

- Jump optimization. This pass simplifies jumps to the following instruction, jumps across jumps, and jumps to jumps. It deletes unreferenced labels and unreachable code, except that unreachable code that contains a loop is not recognized as unreachable in this pass. (Such loops are deleted later in the basic block analysis.) It also converts some code originally written with jumps into sequences of instructions that directly set values from the results of comparisons, if the machine has such instructions.

Jump optimization is performed two or three times. The first time is immediately following RTL generation. The second time is after CSE, but only if CSE says repeated jump optimization is needed. The last time is right before the final pass. That time, cross-jumping and deletion of no-op move instructions are done together with the optimizations described above.

The source file of this pass is `jump.c'.

The option `-dj' causes a debugging dump of the RTL code after this pass is run for the first time. This dump file's name is made by appending `.jump' to the input file name.

- Register scan. This pass finds the first and last use of each register, as a guide for common subexpression elimination. Its source is in `regclass.c'.
- Jump threading. This pass detects a condition jump that branches to an identical or inverse test. Such jumps can be `threaded' through the second conditional test. The source code for this pass is in `jump.c'. This optimization is only performed if `-fthread-jumps' is enabled.
- Common subexpression elimination. This pass also does constant propagation. Its source file is `cse.c'. If constant propagation causes conditional jumps to become unconditional or to become no-ops, jump optimization is run again when CSE is finished.

  The option `-ds' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending `.cse' to the input file name.
- Loop optimization. This pass moves constant expressions out of loops, and optionally does strength-reduction and loop unrolling as well. Its source files are `loop.c' and `unroll.c', plus the header `loop.h' used for communication between them. Loop unrolling uses some functions in `integrate.c' and the header `integrate.h'.

  The option `-dL' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending `.loop' to the input file name.
- If `-frerun-cse-after-loop' was enabled, a second common subexpression elimination pass is performed after the loop optimization pass. Jump threading is also done again at this time if it was specified.

  The option `-dt' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending `.cse2' to the input file name.
- Stupid register allocation is performed at this point in a nonoptimizing compilation. It does a little data flow analysis as well. When stupid register allocation is in use, the next pass executed is the reloading pass; the others in between are skipped. The source file is `stupid.c'.
- Data flow analysis (`flow.c'). This pass divides the program into basic blocks (and in the process deletes unreachable loops); then it computes which pseudo-registers are live at each point in the program, and makes the first instruction that uses a value point at the instruction that computed the value.

  This pass also deletes computations whose results are never used, and combines memory references with add or subtract instructions to make autoincrement or autodecrement addressing.

  The option `-df' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending `.flow' to the input file name. If stupid register allocation is in use, this dump

file reflects the full results of such allocation.

- Instruction combination (`combine.c'). This pass attempts to combine groups of two or three instructions that are related by data flow into single instructions. It combines the RTL expressions for the instructions by substitution, simplifies the result using algebra, and then attempts to match the result against the machine description.

  The option `-dc' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending `.combine' to the input file name.

- Instruction scheduling (`sched.c'). This pass looks for instructions whose output will not be available by the time that it is used in subsequent instructions. (Memory loads and floating point instructions often have this behavior on RISC machines). It re-orders instructions within a basic block to try to separate the definition and use of items that otherwise would cause pipeline stalls.

  Instruction scheduling is performed twice. The first time is immediately after instruction combination and the second is immediately after reload.

  The option `-dS' causes a debugging dump of the RTL code after this pass is run for the first time. The dump file's name is made by appending `.sched' to the input file name.

- Register class preferencing. The RTL code is scanned to find out which register class is best for each pseudo register. The source file is `regclass.c'.

- Local register allocation (`local-alloc.c'). This pass allocates hard registers to pseudo registers that are used only within one basic block. Because the basic block is linear, it can use fast and powerful techniques to do a very good job.

  The option `-dl' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending `.lreg' to the input file name.

- Global register allocation (`global.c'). This pass allocates hard registers for the remaining pseudo registers (those whose life spans are not contained in one basic block).

- Reloading. This pass renumbers pseudo registers with the hardware registers numbers they were allocated. Pseudo registers that did not get hard registers are replaced with stack slots. Then it finds instructions that are invalid because a value has failed to end up in a register, or has ended up in a register of the wrong kind. It fixes up these instructions by reloading the problematical values temporarily into registers. Additional instructions are generated to do the copying.

  The reload pass also optionally eliminates the frame pointer and inserts instructions to save and restore call-clobbered registers around calls.

  Source files are `reload.c' and `reload1.c', plus the header `reload.h' used for communication between them.

  The option `-dg' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending `.greg' to the input file name.

- Instruction scheduling is repeated here to try to avoid pipeline stalls due to memory loads generated for spilled pseudo registers.

  The option `-dR' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending `.sched2' to the input file name.

- Jump optimization is repeated, this time including cross-jumping and deletion of no-op move instructions.

  The option `-dJ' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending `.jump2' to the input file name.
- Delayed branch scheduling. This optional pass attempts to find instructions that can go into the delay slots of other instructions, usually jumps and calls. The source file name is `reorg.c'.

  The option `-dd' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending `.dbr' to the input file name.
- Conversion from usage of some hard registers to usage of a register stack may be done at this point. Currently, this is supported only for the floating-point registers of the Intel 80387 coprocessor. The source file name is `reg-stack.c'.

  The options `-dk' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending `.stack' to the input file name.
- Final. This pass outputs the assembler code for the function. It is also responsible for identifying spurious test and compare instructions. Machine-specific peephole optimizations are performed at the same time. The function entry and exit sequences are generated directly as assembler code in this pass; they never exist as RTL.

  The source files are `final.c' plus `insn-output.c'; the latter is generated automatically from the machine description by the tool `genoutput'. The header file `conditions.h' is used for communication between these files.
- Debugging information output. This is run after final because it must output the stack slot offsets for pseudo registers that did not get hard registers. Source files are `dbxout.c' for DBX symbol table format, `sdbout.c' for SDB symbol table format, and `dwarfout.c' for DWARF symbol table format.

Some additional files are used by all or many passes:

- Every pass uses `machmode.def' and `machmode.h' which define the machine modes.
- Several passes use `real.h', which defines the default representation of floating point constants and how to operate on them.
- All the passes that work with RTL use the header files `rtl.h' and `rtl.def', and subroutines in file `rtl.c'. The tools `gen*` also use these files to read and work with the machine description RTL.
- Several passes refer to the header file `insn-config.h' which contains a few parameters (C macro definitions) generated automatically from the machine description RTL by the tool `genconfig`.
- Several passes use the instruction recognizer, which consists of `recog.c' and `recog.h', plus the files `insn-recog.c' and `insn-extract.c' that are generated automatically from the machine description by the tools `genrecog' and `genextract'.
- Several passes use the header files `regs.h' which defines the information recorded about pseudo register usage, and `basic-block.h' which defines the information recorded about basic blocks.

- `hard-reg-set.h' defines the type HARD_REG_SET, a bit-vector with a bit for each hard register, and some macros to manipulate it. This type is just int if the machine has few enough hard registers; otherwise it is an array of int and some of the macros expand into loops.
- Several passes use instruction attributes. A definition of the attributes defined for a particular machine is in file `insn-attr.h', which is generated from the machine description by the program `genattr'. The file `insn-attrtab.c' contains subroutines to obtain the attribute values for insns. It is generated from the machine description by the program `genattrtab'.

Go to the previous, next section.

Go to the previous, next section.

RTL Representation

Most of the work of the compiler is done on an intermediate representation called register transfer language. In this language, the instructions to be output are described, pretty much one by one, in an algebraic form that describes what the instruction does.

RTL is inspired by Lisp lists. It has both an internal form, made up of structures that point at other structures, and a textual form that is used in the machine description and in printed debugging dumps. The textual form uses nested parentheses to indicate the pointers in the internal form.

RTL Object Types

RTL uses five kinds of objects: expressions, integers, wide integers, strings and vectors. Expressions are the most important ones. An RTL expression ("RTX", for short) is a C structure, but it is usually referred to with a pointer; a type that is given the typedef name `rtx`.

An integer is simply an `int`; their written form uses decimal digits. A wide integer is an integral object whose type is `HOST_WIDE_INT` (see section The Configuration File); their written form uses decimal digits.

A string is a sequence of characters. In core it is represented as a `char *` in usual C fashion, and it is written in C syntax as well. However, strings in RTL may never be null. If you write an empty string in a machine description, it is represented in core as a null pointer rather than as a pointer to a null character. In certain contexts, these null pointers instead of strings are valid. Within RTL code, strings are most commonly found inside `symbol_ref` expressions, but they appear in other contexts in the RTL expressions that make up machine descriptions.

A vector contains an arbitrary number of pointers to expressions. The number of elements in the vector is explicitly present in the vector. The written form of a vector consists of square brackets (`[...]') surrounding the elements, in sequence and with whitespace separating them. Vectors of length zero are not created; null pointers are used instead.

Expressions are classified by expression codes (also called RTX codes). The expression code is a name defined in `rtl.def'`, which is also (in upper case) a C enumeration constant. The possible expression codes and their meanings are machine-independent. The code of an RTX can be extracted with the macro `GET_CODE (x)` and altered with `PUT_CODE (x, newcode)`.

The expression code determines how many operands the expression contains, and what kinds of objects they are. In RTL, unlike Lisp, you cannot tell by looking at an operand what kind of object it is. Instead, you must know from its context--from the expression code of the containing expression. For example, in an expression of code `subreg`, the first operand is to be regarded as an expression and the second operand as an integer. In an expression of code `plus`, there are two operands, both of which are to be regarded as expressions. In a `symbol_ref` expression, there is one operand, which is to be regarded as

a string.

Expressions are written as parentheses containing the name of the expression type, its flags and machine mode if any, and then the operands of the expression (separated by spaces).

Expression code names in the `md' file are written in lower case, but when they appear in C code they are written in upper case. In this manual, they are shown as follows: `const_int`.

In a few contexts a null pointer is valid where an expression is normally wanted. The written form of this is `(nil)`.

Access to Operands

For each expression type `rtl.def' specifies the number of contained objects and their kinds, with four possibilities: `e' for expression (actually a pointer to an expression), `i' for integer, `w' for wide integer, `s' for string, and `E' for vector of expressions. The sequence of letters for an expression code is called its format. Thus, the format of `subreg` is `ei'.

A few other format characters are used occasionally:

`u`

`u' is equivalent to `e' except that it is printed differently in debugging dumps. It is used for pointers to insns.

`n`

`n' is equivalent to `i' except that it is printed differently in debugging dumps. It is used for the line number or code number of a `note` insn.

`S`

`S' indicates a string which is optional. In the RTL objects in core, `S' is equivalent to `s', but when the object is read, from an `md' file, the string value of this operand may be omitted. An omitted string is taken to be the null string.

`V`

`V' indicates a vector which is optional. In the RTL objects in core, `V' is equivalent to `E', but when the object is read from an `md' file, the vector value of this operand may be omitted. An omitted vector is effectively the same as a vector of no elements.

`0`

`0' means a slot whose contents do not fit any normal category. `0' slots are not printed at all in dumps, and are often used in special ways by small parts of the compiler.

There are macros to get the number of operands, the format, and the class of an expression code:

`GET_RTX_LENGTH (code)`

Number of operands of an RTX of code code.

`GET_RTX_FORMAT (code)`

The format of an RTX of code code, as a C string.

`GET_RTX_CLASS (code)`

A single character representing the type of RTX operation that code code performs.

The following classes are defined:

`o`

An RTX code that represents an actual object, such as `reg` or `mem`. `subreg` is not in this class.

`<`

An RTX code for a comparison. The codes in this class are `NE`, `EQ`, `LE`, `LT`, `GE`, `GT`, `LEU`, `LTU`, `GEU`, `GTU`.

`1`

An RTX code for a unary arithmetic operation, such as `neg`.

`c`

An RTX code for a commutative binary operation, other than `NE` and `EQ` (which have class `<').

`2`

An RTX code for a noncommutative binary operation, such as `MINUS`.

`b`

An RTX code for a bitfield operation, either `ZERO_EXTRACT` or `SIGN_EXTRACT`.

`3`

An RTX code for other three input operations, such as `IF_THEN_ELSE`.

`i`

An RTX code for a machine insn (`INSN`, `JUMP_INSN`, and `CALL_INSN`).

`m`

An RTX code for something that matches in insns, such as `MATCH_DUP`.

`x`

All other RTX codes.

Operands of expressions are accessed using the macros `XEXP`, `XINT`, `XWINT` and `XSTR`. Each of these macros takes two arguments: an expression-pointer (RTX) and an operand number (counting from zero). Thus,

```
XEXP (x, 2)
```

accesses operand 2 of expression x, as an expression.

```
XINT (x, 2)
```

accesses the same operand as an integer. `XSTR`, used in the same fashion, would access it as a string.

Any operand can be accessed as an integer, as an expression or as a string. You must choose the correct method of access for the kind of value actually stored in the operand. You would do this based on the expression code of the containing expression. That is also how you would know how many operands

there are.

For example, if x is a `subreg` expression, you know that it has two operands which can be correctly accessed as `XEXP (x, 0)` and `XINT (x, 1)`. If you did `XINT (x, 0)`, you would get the address of the expression operand but cast as an integer; that might occasionally be useful, but it would be cleaner to write `(int) XEXP (x, 0)`. `XEXP (x, 1)` would also compile without error, and would return the second, integer operand cast as an expression pointer, which would probably result in a crash when accessed. Nothing stops you from writing `XEXP (x, 28)` either, but this will access memory past the end of the expression with unpredictable results.

Access to operands which are vectors is more complicated. You can use the macro `XVEC` to get the vector-pointer itself, or the macros `XVECEXP` and `XVECLEN` to access the elements and length of a vector.

`XVEC (exp, idx)`

Access the vector-pointer which is operand number idx in exp.

`XVECLEN (exp, idx)`

Access the length (number of elements) in the vector which is in operand number idx in exp. This value is an `int`.

`XVECEXP (exp, idx, eltnum)`

Access element number eltnum in the vector which is in operand number idx in exp. This value is an RTX.

It is up to you to make sure that eltnum is not negative and is less than `XVECLEN (exp, idx)`.

All the macros defined in this section expand into lvalues and therefore can be used to assign the operands, lengths and vector elements as well as to access them.

Flags in an RTL Expression

RTL expressions contain several flags (one-bit bitfields) that are used in certain types of expression. Most often they are accessed with the following macros:

`MEM_VOLATILE_P (x)`

In `mem` expressions, nonzero for volatile memory references. Stored in the `volatil` field and printed as `/v'.

`MEM_IN_STRUCT_P (x)`

In `mem` expressions, nonzero for reference to an entire structure, union or array, or to a component of one. Zero for references to a scalar variable or through a pointer to a scalar. Stored in the `in_struct` field and printed as `/s'.

`REG_LOOP_TEST_P`

In `reg` expressions, nonzero if this register's entire life is contained in the exit test code for some loop. Stored in the `in_struct` field and printed as `/s'.

`REG_USERVAR_P (x)`

In a `reg`, nonzero if it corresponds to a variable present in the user's source code. Zero for

temporaries generated internally by the compiler. Stored in the `volatil` field and printed as

`REG_FUNCTION_VALUE_P (x)`

Nonzero in a `reg` if it is the place in which this function's value is going to be returned. (This happens only in a hard register.) Stored in the `integrated` field and printed as `/i'.

The same hard register may be used also for collecting the values of functions called by this one, but `REG_FUNCTION_VALUE_P` is zero in this kind of use.

`SUBREG_PROMOTED_VAR_P`

Nonzero in a `subreg` if it was made when accessing an object that was promoted to a wider mode in accord with the `PROMOTED_MODE` machine description macro (see section Storage Layout). In this case, the mode of the `subreg` is the declared mode of the object and the mode of `SUBREG_REG` is the mode of the register that holds the object. Promoted variables are always either sign- or zero-extended to the wider mode on every assignment. Stored in the `in_struct` field and printed as `/s'.

`SUBREG_PROMOTED_UNSIGNED_P`

Nonzero in a `subreg` that has `SUBREG_PROMOTED_VAR_P` nonzero if the object being referenced is kept zero-extended and zero if it is kept sign-extended. Stored in the `unchanging` field and printed as `/u'.

`RTX_UNCHANGING_P (x)`

Nonzero in a `reg` or `mem` if the value is not changed. (This flag is not set for memory references via pointers to constants. Such pointers only guarantee that the object will not be changed explicitly by the current function. The object might be changed by other functions or by aliasing.) Stored in the `unchanging` field and printed as `/u'.

`RTX_INTEGRATED_P (insn)`

Nonzero in an insn if it resulted from an in-line function call. Stored in the `integrated` field and printed as `/i'. This may be deleted; nothing currently depends on it.

`SYMBOL_REF_USED (x)`

In a `symbol_ref`, indicates that x has been used. This is normally only used to ensure that x is only declared external once. Stored in the `used` field.

`SYMBOL_REF_FLAG (x)`

In a `symbol_ref`, this is used as a flag for machine-specific purposes. Stored in the `volatil` field and printed as `/v'.

`LABEL_OUTSIDE_LOOP_P`

In `label_ref` expressions, nonzero if this is a reference to a label that is outside the innermost loop containing the reference to the label. Stored in the `in_struct` field and printed as `/s'.

`INSN_DELETED_P (insn)`

In an insn, nonzero if the insn has been deleted. Stored in the `volatil` field and printed as `/v'.

`INSN_ANNULLED_BRANCH_P (insn)`

In an `insn` in the delay slot of a branch insn, indicates that an annulling branch should be used. See the discussion under `sequence` below. Stored in the `unchanging` field and printed as `/u'.

`INSN_FROM_TARGET_P (insn)`

In an `insn` in a delay slot of a branch, indicates that the insn is from the target of the branch. If the branch insn has `INSN_ANNULLED_BRANCH_P` set, this insn should only be executed if the branch is taken. For annulled branches with this bit clear, the insn should be executed only if the branch is not taken. Stored in the `in_struct` field and printed as `/s'.

`CONSTANT_POOL_ADDRESS_P (x)`

Nonzero in a `symbol_ref` if it refers to part of the current function's "constants pool". These are addresses close to the beginning of the function, and GNU CC assumes they can be addressed directly (perhaps with the help of base registers). Stored in the `unchanging` field and printed as `/u'.

`CONST_CALL_P (x)`

In a `call_insn`, indicates that the insn represents a call to a const function. Stored in the `unchanging` field and printed as `/u'.

`LABEL_PRESERVE_P (x)`

In a `code_label`, indicates that the label can never be deleted. Labels referenced by a non-local goto will have this bit set. Stored in the `in_struct` field and printed as `/s'.

`SCHED_GROUP_P (insn)`

During instruction scheduling, in an insn, indicates that the previous insn must be scheduled together with this insn. This is used to ensure that certain groups of instructions will not be split up by the instruction scheduling pass, for example, `use` insns before a `call_insn` may not be separated from the `call_insn`. Stored in the `in_struct` field and printed as `/s'.

These are the fields which the above macros refer to:

`used`

Normally, this flag is used only momentarily, at the end of RTL generation for a function, to count the number of times an expression appears in insns. Expressions that appear more than once are copied, according to the rules for shared structure (see section Structure Sharing Assumptions).

In a `symbol_ref`, it indicates that an external declaration for the symbol has already been written.

In a `reg`, it is used by the leaf register renumbering code to ensure that each register is only renumbered once.

`volatil`

This flag is used in `mem`, `symbol_ref` and `reg` expressions and in insns. In RTL dump files, it is printed as `/v'.

In a `mem` expression, it is 1 if the memory reference is volatile. Volatile memory references may not be deleted, reordered or combined.

In a `symbol_ref` expression, it is used for machine-specific purposes.

In a `reg` expression, it is 1 if the value is a user-level variable. 0 indicates an internal compiler temporary.

In an insn, 1 means the insn has been deleted.

`in_struct`

In `mem` expressions, it is 1 if the memory datum referred to is all or part of a structure or array; 0 if it is (or might be) a scalar variable. A reference through a C pointer has 0 because the pointer might point to a scalar variable. This information allows the compiler to determine something about possible cases of aliasing.

In an insn in the delay slot of a branch, 1 means that this insn is from the target of the branch.

During instruction scheduling, in an insn, 1 means that this insn must be scheduled as part of a group together with the previous insn.

In `reg` expressions, it is 1 if the register has its entire life contained within the test expression of some loop.

In `subreg` expressions, 1 means that the `subreg` is accessing an object that has had its mode promoted from a wider mode.

In `label_ref` expressions, 1 means that the referenced label is outside the innermost loop containing the insn in which the `label_ref` was found.

In `code_label` expressions, it is 1 if the label may never be deleted. This is used for labels which are the target of non-local gotos.

In an RTL dump, this flag is represented as `/s'.

`unchanging`

In `reg` and `mem` expressions, 1 means that the value of the expression never changes.

In `subreg` expressions, it is 1 if the `subreg` references an unsigned object whose mode has been promoted to a wider mode.

In an insn, 1 means that this is an annulling branch.

In a `symbol_ref` expression, 1 means that this symbol addresses something in the per-function constants pool.

In a `call_insn`, 1 means that this instruction is a call to a const function.

In an RTL dump, this flag is represented as `/u'.

`integrated`

In some kinds of expressions, including insns, this flag means the rtl was produced by procedure integration.

In a `reg` expression, this flag indicates the register containing the value to be returned by the current function. On machines that pass parameters in registers, the same register number may be used for parameters as well, but this flag is not set on such uses.

Machine Modes

A machine mode describes a size of data object and the representation used for it. In the C code, machine modes are represented by an enumeration type, `enum machine_mode`, defined in `machmode.def'`. Each RTL expression has room for a machine mode and so do certain kinds of tree expressions (declarations and types, to be precise).

In debugging dumps and machine descriptions, the machine mode of an RTL expression is written after the expression code with a colon to separate them. The letters `mode' which appear at the end of each machine mode name are omitted. For example, `(reg:SI 38)` is a `reg` expression with machine mode `SImode`. If the mode is `VOIDmode`, it is not written at all.

Here is a table of machine modes. The term "byte" below refers to an object of `BITS_PER_UNIT` bits (see section Storage Layout).

`QImode`
: "Quarter-Integer" mode represents a single byte treated as an integer.

`HImode`
: "Half-Integer" mode represents a two-byte integer.

`PSImode`
: "Partial Single Integer" mode represents an integer which occupies four bytes but which doesn't really use all four. On some machines, this is the right mode to use for pointers.

`SImode`
: "Single Integer" mode represents a four-byte integer.

`PDImode`
: "Partial Double Integer" mode represents an integer which occupies eight bytes but which doesn't really use all eight. On some machines, this is the right mode to use for certain pointers.

`DImode`
: "Double Integer" mode represents an eight-byte integer.

`TImode`
: "Tetra Integer" (?) mode represents a sixteen-byte integer.

`SFmode`
: "Single Floating" mode represents a single-precision (four byte) floating point number.

`DFmode`
: "Double Floating" mode represents a double-precision (eight byte) floating point number.

`XFmode`
: "Extended Floating" mode represents a triple-precision (twelve byte) floating point number. This mode is used for IEEE extended floating point. On some systems not all bits within these bytes will actually be used.

`TFmode`
: "Tetra Floating" mode represents a quadruple-precision (sixteen byte) floating point number.

`CCmode`

"Condition Code" mode represents the value of a condition code, which is a machine-specific set of bits used to represent the result of a comparison operation. Other machine-specific modes may also be used for the condition code. These modes are not used on machines that use `cc0` (see see section Condition Code Status).

`BLKmode`

"Block" mode represents values that are aggregates to which none of the other modes apply. In RTL, only memory references can have this mode, and only if they appear in string-move or vector instructions. On machines which have no such instructions, `BLKmode` will not appear in RTL.

`VOIDmode`

Void mode means the absence of a mode or an unspecified mode. For example, RTL expressions of code `const_int` have mode `VOIDmode` because they can be taken to have whatever mode the context requires. In debugging dumps of RTL, `VOIDmode` is expressed by the absence of any mode.

`SCmode, DCmode, XCmode, TCmode`

These modes stand for a complex number represented as a pair of floating point values. The floating point values are in `SFmode`, `DFmode`, `XFmode`, and `TFmode`, respectively.

`CQImode, CHImode, CSImode, CDImode, CTImode, COImode`

These modes stand for a complex number represented as a pair of integer values. The integer values are in `QImode`, `HImode`, `SImode`, `DImode`, `TImode`, and `OImode`, respectively.

The machine description defines `Pmode` as a C macro which expands into the machine mode used for addresses. Normally this is the mode whose size is `BITS_PER_WORD`, `SImode` on 32-bit machines.

The only modes which a machine description *must* support are `QImode`, and the modes corresponding to `BITS_PER_WORD`, `FLOAT_TYPE_SIZE` and `DOUBLE_TYPE_SIZE`. The compiler will attempt to use `DImode` for 8-byte structures and unions, but this can be prevented by overriding the definition of `MAX_FIXED_MODE_SIZE`. Alternatively, you can have the compiler use `TImode` for 16-byte structures and unions. Likewise, you can arrange for the C type `short int` to avoid using `HImode`.

Very few explicit references to machine modes remain in the compiler and these few references will soon be removed. Instead, the machine modes are divided into mode classes. These are represented by the enumeration type `enum mode_class` defined in `` `machmode.h' ``. The possible mode classes are:

`MODE_INT`

Integer modes. By default these are `QImode`, `HImode`, `SImode`, `DImode`, and `TImode`.

`MODE_PARTIAL_INT`

The "partial integer" modes, `PSImode` and `PDImode`.

`MODE_FLOAT`

floating point modes. By default these are `SFmode`, `DFmode`, `XFmode` and `TFmode`.

`MODE_COMPLEX_INT`

Complex integer modes. (These are not currently implemented).

`MODE_COMPLEX_FLOAT`

Complex floating point modes. By default these are `SCmode`, `DCmode`, `XCmode`, and `TCmode`.

`MODE_FUNCTION`

Algol or Pascal function variables including a static chain. (These are not currently implemented).

`MODE_CC`

Modes representing condition code values. These are `CCmode` plus any modes listed in the `EXTRA_CC_MODES` macro. See section Defining Jump Instruction Patterns, also see section Condition Code Status.

`MODE_RANDOM`

This is a catchall mode class for modes which don't fit into the above classes. Currently `VOIDmode` and `BLKmode` are in `MODE_RANDOM`.

Here are some C macros that relate to machine modes:

`GET_MODE (x)`

Returns the machine mode of the RTX x.

`PUT_MODE (x, newmode)`

Alters the machine mode of the RTX x to be newmode.

`NUM_MACHINE_MODES`

Stands for the number of machine modes available on the target machine. This is one greater than the largest numeric value of any machine mode.

`GET_MODE_NAME (m)`

Returns the name of mode m as a string.

`GET_MODE_CLASS (m)`

Returns the mode class of mode m.

`GET_MODE_WIDER_MODE (m)`

Returns the next wider natural mode. For example, the expression `GET_MODE_WIDER_MODE (QImode)` returns `HImode`.

`GET_MODE_SIZE (m)`

Returns the size in bytes of a datum of mode m.

`GET_MODE_BITSIZE (m)`

Returns the size in bits of a datum of mode m.

`GET_MODE_MASK (m)`

Returns a bitmask containing 1 for all bits in a word that fit within mode m. This macro can only be used for modes whose bitsize is less than or equal to `HOST_BITS_PER_INT`.

`GET_MODE_ALIGNMENT (m))`

Return the required alignment, in bits, for an object of mode m.

`GET_MODE_UNIT_SIZE (m)`

Returns the size in bytes of the subunits of a datum of mode m. This is the same as `GET_MODE_SIZE` except in the case of complex modes. For them, the unit size is the size of the real or imaginary part.

`GET_MODE_NUNITS (m)`

Returns the number of units contained in a mode, i.e., `GET_MODE_SIZE` divided by `GET_MODE_UNIT_SIZE`.

`GET_CLASS_NARROWEST_MODE (c)`

Returns the narrowest mode in mode class c.

The global variables `byte_mode` and `word_mode` contain modes whose classes are `MODE_INT` and whose bitsizes are either `BITS_PER_UNIT` or `BITS_PER_WORD`, respectively. On 32-bit machines, these are `QImode` and `SImode`, respectively.

Constant Expression Types

The simplest RTL expressions are those that represent constant values.

`(const_int i)`

This type of expression represents the integer value i. i is customarily accessed with the macro `INTVAL` as in `INTVAL (exp)`, which is equivalent to `XWINT (exp, 0)`.

There is only one expression object for the integer value zero; it is the value of the variable `const0_rtx`. Likewise, the only expression for integer value one is found in `const1_rtx`, the only expression for integer value two is found in `const2_rtx`, and the only expression for integer value negative one is found in `constm1_rtx`. Any attempt to create an expression of code `const_int` and value zero, one, two or negative one will return `const0_rtx`, `const1_rtx`, `const2_rtx` or `constm1_rtx` as appropriate.

Similarly, there is only one object for the integer whose value is `STORE_FLAG_VALUE`. It is found in `const_true_rtx`. If `STORE_FLAG_VALUE` is one, `const_true_rtx` and `const1_rtx` will point to the same object. If `STORE_FLAG_VALUE` is -1, `const_true_rtx` and `constm1_rtx` will point to the same object.

`(const_double:m addr i0 i1 ...)`

Represents either a floating-point constant of mode m or an integer constant too large to fit into `HOST_BITS_PER_WIDE_INT` bits but small enough to fit within twice that number of bits (GNU CC does not provide a mechanism to represent even larger constants). In the latter case, m will be `VOIDmode`.

addr is used to contain the `mem` expression that corresponds to the location in memory that at which the constant can be found. If it has not been allocated a memory location, but is on the chain of all `const_double` expressions in this compilation (maintained using an undisplayed field), addr contains `const0_rtx`. If it is not on the chain, addr contains `cc0_rtx`. addr is customarily accessed with the macro `CONST_DOUBLE_MEM` and the chain field via `CONST_DOUBLE_CHAIN`.

If m is `VOIDmode`, the bits of the value are stored in i0 and i1. i0 is customarily accessed with the macro `CONST_DOUBLE_LOW` and i1 with `CONST_DOUBLE_HIGH`.

If the constant is floating point (regardless of its precision), then the number of integers used to

store the value depends on the size of `REAL_VALUE_TYPE` (see section Cross Compilation a Floating Point). The integers represent a floating point number, but not precisely in the target machine's or host machine's floating point format. To convert them to the precise bit pattern used by the target machine, use the macro `REAL_VALUE_TO_TARGET_DOUBLE` and friends (see section Output of Data).

The macro `CONST0_RTX (mode)` refers to an expression with value 0 in mode mode. If mode mode is of mode class `MODE_INT`, it returns `const0_rtx`. Otherwise, it returns a `CONST_DOUBLE` expression in mode mode. Similarly, the macro `CONST1_RTX (mode)` refers to an expression with value 1 in mode mode and similarly for `CONST2_RTX`.

`(const_string str)`

Represents a constant string with value str. Currently this is used only for insn attributes (see section Instruction Attributes) since constant strings in C are placed in memory.

`(symbol_ref:mode symbol)`

Represents the value of an assembler label for data. symbol is a string that describes the name of the assembler label. If it starts with a `*', the label is the rest of symbol not including the `*'. Otherwise, the label is symbol, usually prefixed with `_'.

The `symbol_ref` contains a mode, which is usually `Pmode`. Usually that is the only mode for which a symbol is directly valid.

`(label_ref label)`

Represents the value of an assembler label for code. It contains one operand, an expression, which must be a `code_label` that appears in the instruction sequence to identify the place where the label should go.

The reason for using a distinct expression type for code label references is so that jump optimization can distinguish them.

`(const:m exp)`

Represents a constant that is the result of an assembly-time arithmetic computation. The operand, exp, is an expression that contains only constants (`const_int`, `symbol_ref` and `label_ref` expressions) combined with `plus` and `minus`. However, not all combinations are valid, since the assembler cannot do arbitrary arithmetic on relocatable symbols.

m should be `Pmode`.

`(high:m exp)`

Represents the high-order bits of exp, usually a `symbol_ref`. The number of bits is machine-dependent and is normally the number of bits specified in an instruction that initializes the high order bits of a register. It is used with `lo_sum` to represent the typical two-instruction sequence used in RISC machines to reference a global memory location.

m should be `Pmode`.

Registers and Memory

Here are the RTL expression types for describing access to machine registers and to main memory.

`(reg:m n)`

For small values of the integer n (those that are less than `FIRST_PSEUDO_REGISTER`), this stands for a reference to machine register number n: a hard register. For larger values of n, it stands for a temporary value or pseudo register. The compiler's strategy is to generate code assuming an unlimited number of such pseudo registers, and later convert them into hard registers or into memory references.

m is the machine mode of the reference. It is necessary because machines can generally refer to each register in more than one mode. For example, a register may contain a full word but there may be instructions to refer to it as a half word or as a single byte, as well as instructions to refer to it as a floating point number of various precisions.

Even for a register that the machine can access in only one mode, the mode must always be specified.

The symbol `FIRST_PSEUDO_REGISTER` is defined by the machine description, since the number of hard registers on the machine is an invariant characteristic of the machine. Note, however, that not all of the machine registers must be general registers. All the machine registers that can be used for storage of data are given hard register numbers, even those that can be used only in certain instructions or can hold only certain types of data.

A hard register may be accessed in various modes throughout one function, but each pseudo register is given a natural mode and is accessed only in that mode. When it is necessary to describe an access to a pseudo register using a nonnatural mode, a `subreg` expression is used.

A `reg` expression with a machine mode that specifies more than one word of data may actually stand for several consecutive registers. If in addition the register number specifies a hardware register, then it actually represents several consecutive hardware registers starting with the specified one.

Each pseudo register number used in a function's RTL code is represented by a unique `reg` expression.

Some pseudo register numbers, those within the range of `FIRST_VIRTUAL_REGISTER` to `LAST_VIRTUAL_REGISTER` only appear during the RTL generation phase and are eliminated before the optimization phases. These represent locations in the stack frame that cannot be determined until RTL generation for the function has been completed. The following virtual register numbers are defined:

`VIRTUAL_INCOMING_ARGS_REGNUM`

This points to the first word of the incoming arguments passed on the stack. Normally these arguments are placed there by the caller, but the callee may have pushed some arguments that were previously passed in registers.

When RTL generation is complete, this virtual register is replaced by the sum of the register

given by `ARG_POINTER_REGNUM` and the value of `FIRST_PARM_OFFSET`.

`VIRTUAL_STACK_VARS_REGNUM`

If `FRAME_GROWS_DOWNWARD` is defined, this points to immediately above the first variable on the stack. Otherwise, it points to the first variable on the stack.

`VIRTUAL_STACK_VARS_REGNUM` is replaced with the sum of the register given by `FRAME_POINTER_REGNUM` and the value `STARTING_FRAME_OFFSET`.

`VIRTUAL_STACK_DYNAMIC_REGNUM`

This points to the location of dynamically allocated memory on the stack immediately after the stack pointer has been adjusted by the amount of memory desired.

This virtual register is replaced by the sum of the register given by `STACK_POINTER_REGNUM` and the value `STACK_DYNAMIC_OFFSET`.

`VIRTUAL_OUTGOING_ARGS_REGNUM`

This points to the location in the stack at which outgoing arguments should be written when the stack is pre-pushed (arguments pushed using push insns should always use `STACK_POINTER_REGNUM`).

This virtual register is replaced by the sum of the register given by `STACK_POINTER_REGNUM` and the value `STACK_POINTER_OFFSET`.

- (subreg:m reg wordnum) `subreg` expressions are used to refer to a register in a machine mode other than its natural one, or to refer to one register of a multi-word `reg` that actually refers to several registers.

Each pseudo-register has a natural mode. If it is necessary to operate on it in a different mode--for example, to perform a fullword move instruction on a pseudo-register that contains a single byte--the pseudo-register must be enclosed in a `subreg`. In such a case, wordnum is zero.

Usually m is at least as narrow as the mode of reg, in which case it is restricting consideration to only the bits of reg that are in m.

Sometimes m is wider than the mode of reg. These `subreg` expressions are often called paradoxical. They are used in cases where we want to refer to an object in a wider mode but do not care what value the additional bits have. The reload pass ensures that paradoxical references are only made to hard registers.

The other use of `subreg` is to extract the individual registers of a multi-register value. Machine modes such as `DImode` and `TImode` can indicate values longer than a word, values which usually require two or more consecutive registers. To access one of the registers, use a `subreg` with mode `SImode` and a wordnum that says which register.

Storing in a non-paradoxical `subreg` has undefined results for bits belonging to the same word as the `subreg`. This laxity makes it easier to generate efficient code for such instructions. To represent an instruction that preserves all the bits outside of those in the `subreg`, use `strict_low_part` around the `subreg`.

The compilation parameter `WORDS_BIG_ENDIAN`, if set to 1, says that word number zero is the most

significant part; otherwise, it is the least significant part.

Between the combiner pass and the reload pass, it is possible to have a paradoxical `subreg` which contains a `mem` instead of a `reg` as its first operand. After the reload pass, it is also possible to have a non-paradoxical `subreg` which contains a `mem`; this usually occurs when the `mem` is a stack slot which replaced a pseudo register.

Note that it is not valid to access a `DFmode` value in `SFmode` using a `subreg`. On some machines the most significant part of a `DFmode` value does not have the same format as a single-precision floating value.

It is also not valid to access a single word of a multi-word value in a hard register when less registers can hold the value than would be expected from its size. For example, some 32-bit machines have floating-point registers that can hold an entire `DFmode` value. If register 10 were such a register `(subreg:SI (reg:DF 10) 1)` would be invalid because there is no way to convert that reference to a single machine register. The reload pass prevents `subreg` expressions such as these from being formed.

The first operand of a `subreg` expression is customarily accessed with the `SUBREG_REG` macro and the second operand is customarily accessed with the `SUBREG_WORD` macro.

- (scratch:m) This represents a scratch register that will be required for the execution of a single instruction and not used subsequently. It is converted into a `reg` by either the local register allocator or the reload pass.

`scratch` is usually present inside a `clobber` operation (see section Side Effect Expressions).

- (cc0) This refers to the machine's condition code register. It has no operands and may not have a machine mode. There are two ways to use it:
  - To stand for a complete set of condition code flags. This is best on most machines, where each comparison sets the entire series of flags.

    With this technique, `(cc0)` may be validly used in only two contexts: as the destination of an assignment (in test and compare instructions) and in comparison operators comparing against zero (`const_int` with value zero; that is to say, `const0_rtx`).
  - To stand for a single flag that is the result of a single condition. This is useful on machines that have only a single flag bit, and in which comparison instructions must specify the condition to test.

    With this technique, `(cc0)` may be validly used in only two contexts: as the destination of an assignment (in test and compare instructions) where the source is a comparison operator, and as the first operand of `if_then_else` (in a conditional branch).

There is only one expression object of code `cc0`; it is the value of the variable `cc0_rtx`. Any attempt to create an expression of code `cc0` will return `cc0_rtx`.

Instructions can set the condition code implicitly. On many machines, nearly all instructions set the condition code based on the value that they compute or store. It is not necessary to record these actions explicitly in the RTL because the machine description includes a prescription for recognizing the instructions that do so (by means of the macro `NOTICE_UPDATE_CC`). See section Condition Code

Status. Only instructions whose sole purpose is to set the condition code, and instructions that use the condition code, need mention `(cc0)`.

On some machines, the condition code register is given a register number and a `reg` is used instead of `(cc0)`. This is usually the preferable approach if only a small subset of instructions modify the condition code. Other machines store condition codes in general registers; in such cases a pseudo register should be used.

Some machines, such as the Sparc and RS/6000, have two sets of arithmetic instructions, one that sets and one that does not set the condition code. This is best handled by normally generating the instruction that does not set the condition code, and making a pattern that both performs the arithmetic and sets the condition code register (which would not be `(cc0)` in this case). For examples, search for `addcc' and `andcc' in `sparc.md'.

- (pc) This represents the machine's program counter. It has no operands and may not have a machine mode. `(pc)` may be validly used only in certain specific contexts in jump instructions.

There is only one expression object of code `pc`; it is the value of the variable `pc_rtx`. Any attempt to create an expression of code `pc` will return `pc_rtx`.

All instructions that do not jump alter the program counter implicitly by incrementing it, but there is no need to mention this in the RTL.

- (mem:m addr) This RTX represents a reference to main memory at an address represented by the expression addr. m specifies how large a unit of memory is accessed.

RTL Expressions for Arithmetic

Unless otherwise specified, all the operands of arithmetic expressions must be valid for mode m. An operand is valid for mode m if it has mode m, or if it is a `const_int` or `const_double` and m is a mode of class `MODE_INT`.

For commutative binary operations, constants should be placed in the second operand.

`(plus:m x y)`

Represents the sum of the values represented by x and y carried out in machine mode m.

`(lo_sum:m x y)`

Like `plus`, except that it represents that sum of x and the low-order bits of y. The number of low order bits is machine-dependent but is normally the number of bits in a `Pmode` item minus the number of bits set by the `high` code (see section Constant Expression Types).

m should be `Pmode`.

`(minus:m x y)`

Like `plus` but represents subtraction.

`(compare:m x y)`

Represents the result of subtracting y from x for purposes of comparison. The result is computed without overflow, as if with infinite precision.

Of course, machines can't really subtract with infinite precision. However, they can pretend to do so when only the sign of the result will be used, which is the case when the result is stored in the condition code. And that is the only way this kind of expression may validly be used: as a value to be stored in the condition codes.

The mode m is not related to the modes of x and y, but instead is the mode of the condition code value. If `(cc0)` is used, it is `VOIDmode`. Otherwise it is some mode in class `MODE_CC`, often `CCmode`. See section Condition Code Status.

Normally, x and y must have the same mode. Otherwise, `compare` is valid only if the mode of x is in class `MODE_INT` and y is a `const_int` or `const_double` with mode `VOIDmode`. The mode of x determines what mode the comparison is to be done in; thus it must not be `VOIDmode`.

If one of the operands is a constant, it should be placed in the second operand and the comparison code adjusted as appropriate.

A `compare` specifying two `VOIDmode` constants is not valid since there is no way to know in what mode the comparison is to be performed; the comparison must either be folded during the compilation or the first operand must be loaded into a register while its mode is still known.

`(neg:m x)`

Represents the negation (subtraction from zero) of the value represented by x, carried out in mode m.

`(mult:m x y)`

Represents the signed product of the values represented by x and y carried out in machine mode m.

Some machines support a multiplication that generates a product wider than the operands. Write the pattern for this as

```
(mult:m (sign_extend:m x) (sign_extend:m y))
```

where m is wider than the modes of x and y, which need not be the same.

Write patterns for unsigned widening multiplication similarly using `zero_extend`.

`(div:m x y)`

Represents the quotient in signed division of x by y, carried out in machine mode m. If m is a floating point mode, it represents the exact quotient; otherwise, the integerized quotient.

Some machines have division instructions in which the operands and quotient widths are not all the same; you should represent such instructions using `truncate` and `sign_extend` as in,

```
(truncate:m1 (div:m2 x (sign_extend:m2 y)))
```

`(udiv:m x y)`

Like `div` but represents unsigned division.

`(mod:m x y)`

`(umod:m x y)`

Like `div` and `udiv` but represent the remainder instead of the quotient.

`(smin:m x y)`

`(smax:m x y)`

Represents the smaller (for `smin`) or larger (for `smax`) of x and y, interpreted as signed integers in mode m.

`(umin:m x y)`

`(umax:m x y)`

Like `smin` and `smax`, but the values are interpreted as unsigned integers.

`(not:m x)`

Represents the bitwise complement of the value represented by x, carried out in mode m, which must be a fixed-point machine mode.

`(and:m x y)`

Represents the bitwise logical-and of the values represented by x and y, carried out in machine mode m, which must be a fixed-point machine mode.

`(ior:m x y)`

Represents the bitwise inclusive-or of the values represented by x and y, carried out in machine mode m, which must be a fixed-point mode.

`(xor:m x y)`

Represents the bitwise exclusive-or of the values represented by x and y, carried out in machine mode m, which must be a fixed-point mode.

`(ashift:m x c)`

Represents the result of arithmetically shifting x left by c places. x have mode m, a fixed-point machine mode. c be a fixed-point mode or be a constant with mode `VOIDmode`; which mode is determined by the mode called for in the machine description entry for the left-shift instruction. For example, on the Vax, the mode of c is `QImode` regardless of m.

`(lshiftrt:m x c)`

`(ashiftrt:m x c)`

Like `ashift` but for right shift. Unlike the case for left shift, these two operations are distinct.

`(rotate:m x c)`

`(rotatert:m x c)`

Similar but represent left and right rotate. If c is a constant, use `rotate`.

`(abs:m x)`

Represents the absolute value of x, computed in mode m.

`(sqrt:m x)`

Represents the square root of x, computed in mode m. Most often m will be a floating point mode.

`(ffs:m x)`

Represents one plus the index of the least significant 1-bit in x, represented as an integer of mode m. (The value is zero if x is zero.) The mode of x need not be m; depending on the target machine,

various mode combinations may be valid.

Comparison Operations

Comparison operators test a relation on two operands and are considered to represent a machine-dependent nonzero value described by, but not necessarily equal to, `STORE_FLAG_VALUE` (see section Miscellaneous Parameters) if the relation holds, or zero if it does not. The mode of the comparison operation is independent of the mode of the data being compared. If the comparison operation is being tested (e.g., the first operand of an `if_then_else`), the mode must be `VOIDmode`. If the comparison operation is producing data to be stored in some variable, the mode must be in class `MODE_INT`. All comparison operations producing data must use the same mode, which is machine-specific.

There are two ways that comparison operations may be used. The comparison operators may be used to compare the condition codes `(cc0)` against zero, as in `(eq (cc0) (const_int 0))`. Such a construct actually refers to the result of the preceding instruction in which the condition codes were set. The instructing setting the condition code must be adjacent to the instruction using the condition code; only `note` insns may separate them.

Alternatively, a comparison operation may directly compare two data objects. The mode of the comparison is determined by the operands; they must both be valid for a common machine mode. A comparison with both operands constant would be invalid as the machine mode could not be deduced from it, but such a comparison should never exist in RTL due to constant folding.

In the example above, if `(cc0)` were last set to `(compare x y)`, the comparison operation is identical to `(eq x y)`. Usually only one style of comparisons is supported on a particular machine, but the combine pass will try to merge the operations to produce the `eq` shown in case it exists in the context of the particular insn involved.

Inequality comparisons come in two flavors, signed and unsigned. Thus, there are distinct expression codes `gt` and `gtu` for signed and unsigned greater-than. These can produce different results for the same pair of integer values: for example, 1 is signed greater-than -1 but not unsigned greater-than, because -1 when regarded as unsigned is actually `0xffffffff` which is greater than 1.

The signed comparisons are also used for floating point values. Floating point comparisons are distinguished by the machine modes of the operands.

`(eq:m x y)`

1 if the values represented by x and y are equal, otherwise 0.

`(ne:m x y)`

1 if the values represented by x and y are not equal, otherwise 0.

`(gt:m x y)`

1 if the x is greater than y. If they are fixed-point, the comparison is done in a signed sense.

`(gtu:m x y)`

Like `gt` but does unsigned comparison, on fixed-point numbers only.

`(lt:m x y)`
`(ltu:m x y)`

Like `gt` and `gtu` but test for "less than".

`(ge:m x y)`
`(geu:m x y)`

Like `gt` and `gtu` but test for "greater than or equal".

`(le:m x y)`
`(leu:m x y)`

Like `gt` and `gtu` but test for "less than or equal".

`(if_then_else cond then else)`

This is not a comparison operation but is listed here because it is always used in conjunction with a comparison operation. To be precise, cond is a comparison expression. This expression represents a choice, according to cond, between the value represented by then and the one represented by else.

On most machines, `if_then_else` expressions are valid only to express conditional jumps.

`(cond [test1 value1 test2 value2 ...] default)`

Similar to `if_then_else`, but more general. Each of test1, test2, ... is performed in turn. The result of this expression is the value corresponding to the first non-zero test, or default if none of the tests are non-zero expressions.

This is currently not valid for instruction patterns and is supported only for insn attributes. See section Instruction Attributes.

Bit Fields

Special expression codes exist to represent bitfield instructions. These types of expressions are lvalues in RTL; they may appear on the left side of an assignment, indicating insertion of a value into the specified bit field.

`(sign_extract:m loc size pos)`

This represents a reference to a sign-extended bit field contained or starting in loc (a memory or register reference). The bit field is size bits wide and starts at bit pos. The compilation option `BITS_BIG_ENDIAN` says which end of the memory unit pos counts from.

If loc is in memory, its mode must be a single-byte integer mode. If loc is in a register, the mode to use is specified by the operand of the `insv` or `extv` pattern (see section Standard Pattern Names For Generation) and is usually a full-word integer mode.

The mode of pos is machine-specific and is also specified in the `insv` or `extv` pattern.

The mode m is the same as the mode that would be used for loc if it were a register.

`(zero_extract:m loc size pos)`

Like `sign_extract` but refers to an unsigned or zero-extended bit field. The same sequenc bits are extracted, but they are filled to an entire word with zeros instead of by sign-extension.

Conversions

All conversions between machine modes must be represented by explicit conversion operations. For example, an expression which is the sum of a byte and a full word cannot be written as `(plus:SI (reg:QI 34) (reg:SI 80))` because the `plus` operation requires two operands of the same machine mode. Therefore, the byte-sized operand is enclosed in a conversion operation, as in

```
(plus:SI (sign_extend:SI (reg:QI 34)) (reg:SI 80))
```

The conversion operation is not a mere placeholder, because there may be more than one way of converting from a given starting mode to the desired final mode. The conversion operation code says how to do it.

For all conversion operations, x must not be `VOIDmode` because the mode in which to do the conversion would not be known. The conversion must either be done at compile-time or x must be placed into a register.

`(sign_extend:m x)`

Represents the result of sign-extending the value x to machine mode m. m must be a fixed-point mode and x a fixed-point value of a mode narrower than m.

`(zero_extend:m x)`

Represents the result of zero-extending the value x to machine mode m. m must be a fixed-point mode and x a fixed-point value of a mode narrower than m.

`(float_extend:m x)`

Represents the result of extending the value x to machine mode m. m must be a floating point mode and x a floating point value of a mode narrower than m.

`(truncate:m x)`

Represents the result of truncating the value x to machine mode m. m must be a fixed-point mode and x a fixed-point value of a mode wider than m.

`(float_truncate:m x)`

Represents the result of truncating the value x to machine mode m. m must be a floating point mode and x a floating point value of a mode wider than m.

`(float:m x)`

Represents the result of converting fixed point value x, regarded as signed, to floating point mode m.

`(unsigned_float:m x)`

Represents the result of converting fixed point value x, regarded as unsigned, to floating point mode m.

`(fix:m x)`

When m is a fixed point mode, represents the result of converting floating point value x to mo m, regarded as signed. How rounding is done is not specified, so this operation may be used validly in compiling C code only for integer-valued operands.

`(unsigned_fix:m x)`

Represents the result of converting floating point value x to fixed point mode m, regarded as unsigned. How rounding is done is not specified.

`(fix:m x)`

When m is a floating point mode, represents the result of converting floating point value x (valid for mode m) to an integer, still represented in floating point mode m, by rounding towards zero.

Declarations

Declaration expression codes do not represent arithmetic operations but rather state assertions about their operands.

`(strict_low_part (subreg:m (reg:n r) 0))`

This expression code is used in only one context: as the destination operand of a `set` expression. In addition, the operand of this expression must be a non-paradoxical `subreg` expression.

The presence of `strict_low_part` says that the part of the register which is meaningful in mode n, but is not part of mode m, is not to be altered. Normally, an assignment to such a subreg is allowed to have undefined effects on the rest of the register when m is less than a word.

Side Effect Expressions

The expression codes described so far represent values, not actions. But machine instructions never produce values; they are meaningful only for their side effects on the state of the machine. Special expression codes are used to represent side effects.

The body of an instruction is always one of these side effect codes; the codes described above, which represent values, appear only as the operands of these.

`(set lval x)`

Represents the action of storing the value of x into the place represented by lval. lval must be an expression representing a place that can be stored in: `reg` (or `subreg` or `strict_low_part`), `mem`, `pc` or `cc0`.

If lval is a `reg`, `subreg` or `mem`, it has a machine mode; then x must be valid for that mode.

If lval is a `reg` whose machine mode is less than the full width of the register, then it means that the part of the register specified by the machine mode is given the specified value and the rest of the register receives an undefined value. Likewise, if lval is a `subreg` whose machine mode is narrower than the mode of the register, the rest of the register can be changed in an undefined way.

If lval is a `strict_low_part` of a `subreg`, then the part of the register specified by the machine mode of the `subreg` is given the value x and the rest of the register is not changed.

If lval is `(cc0)`, it has no machine mode, and x may be either a `compare` expression or a value that may have any mode. The latter case represents a "test" instruction. The expression `(set (cc0) (reg:m n))` is equivalent to `(set (cc0) (compare (reg:m n) (const_int 0)))`. Use the former expression to save space during the compilation.

If lval is `(pc)`, we have a jump instruction, and the possibilities for x are very limited. It may be a `label_ref` expression (unconditional jump). It may be an `if_then_else` (conditional jump), in which case either the second or the third operand must be `(pc)` (for the case which does not jump) and the other of the two must be a `label_ref` (for the case which does jump). x may also be a `mem` or `(plus:SI (pc) y)`, where y may be a `reg` or a `mem`; these unusual patterns are used to represent jumps through branch tables.

If lval is neither `(cc0)` nor `(pc)`, the mode of lval must not be `VOIDmode` and the mode of x must be valid for the mode of lval.

lval is customarily accessed with the `SET_DEST` macro and x with the `SET_SRC` macro.

`(return)`

As the sole expression in a pattern, represents a return from the current function, on machines where this can be done with one instruction, such as Vaxes. On machines where a multi-instruction "epilogue" must be executed in order to return from the function, returning is done by jumping to a label which precedes the epilogue, and the `return` expression code is never used.

Inside an `if_then_else` expression, represents the value to be placed in `pc` to return to the caller.

Note that an insn pattern of `(return)` is logically equivalent to `(set (pc) (return))`, but the latter form is never used.

`(call function nargs)`

Represents a function call. function is a `mem` expression whose address is the address of the function to be called. nargs is an expression which can be used for two purposes: on some machines it represents the number of bytes of stack argument; on others, it represents the number of argument registers.

Each machine has a standard machine mode which function must have. The machine description defines macro `FUNCTION_MODE` to expand into the requisite mode name. The purpose of this mode is to specify what kind of addressing is allowed, on machines where the allowed kinds of addressing depend on the machine mode being addressed.

`(clobber x)`

Represents the storing or possible storing of an unpredictable, undescribed value into x, which must be a `reg`, `scratch` or `mem` expression.

One place this is used is in string instructions that store standard values into particular hard registers. It may not be worth the trouble to describe the values that are stored, but it is essential to inform the compiler that the registers will be altered, lest it attempt to keep data in them across the string instruction.

If x is `(mem:BLK (const_int 0))`, it means that all memory locations must be presumed

clobbered.

Note that the machine description classifies certain hard registers as "call-clobbered". All function call instructions are assumed by default to clobber these registers, so there is no need to use `clobber` expressions to indicate this fact. Also, each function call is assumed to have the potential to alter any memory location, unless the function is declared `const`.

If the last group of expressions in a `parallel` are each a `clobber` expression whose arguments are `reg` or `match_scratch` (see section RTL Template) expressions, the combiner phase can add the appropriate `clobber` expressions to an insn it has constructed when doing so will cause a pattern to be matched.

This feature can be used, for example, on a machine that whose multiply and add instructions don't use an MQ register but which has an add-accumulate instruction that does clobber the MQ register. Similarly, a combined instruction might require a temporary register while the constituent instructions might not.

When a `clobber` expression for a register appears inside a `parallel` with other side effects, the register allocator guarantees that the register is unoccupied both before and after that insn. However, the reload phase may allocate a register used for one of the inputs unless the `&' constraint is specified for the selected alternative (see section Constraint Modifier Characters). You can clobber either a specific hard register, a pseudo register, or a `scratch` expression; in the latter two cases, GNU CC will allocate a hard register that is available there for use as a temporary.

For instructions that require a temporary register, you should use `scratch` instead of a pseudo-register because this will allow the combiner phase to add the `clobber` when required. You do this by coding (`clobber` (`match_scratch` ...)). If you do clobber a pseudo register, use one which appears nowhere else--generate a new one each time. Otherwise, you may confuse CSE.

There is one other known use for clobbering a pseudo register in a `parallel`: when one of the input operands of the insn is also clobbered by the insn. In this case, using the same pseudo register in the clobber and elsewhere in the insn produces the expected results.

`(use x)`

Represents the use of the value of x. It indicates that the value in x at this point in the program is needed, even though it may not be apparent why this is so. Therefore, the compiler will not attempt to delete previous instructions whose only effect is to store a value in x. x must be a `reg` expression.

During the delayed branch scheduling phase, x may be an insn. This indicates that x previously was located at this place in the code and its data dependencies need to be taken into account. These `use` insns will be deleted before the delayed branch scheduling phase exits.

`(parallel [x0 x1 ...])`

Represents several side effects performed in parallel. The square brackets stand for a vector; the operand of `parallel` is a vector of expressions. x0, x1 and so on are individual side effect expressions--expressions of code `set`, `call`, `return`, `clobber` or `use`.

"In parallel" means that first all the values used in the individual side-effects are computed, and second all the actual side-effects are performed. For example,

```
(parallel [(set (reg:SI 1) (mem:SI (reg:SI 1)))
           (set (mem:SI (reg:SI 1)) (reg:SI 1))])
```

says unambiguously that the values of hard register 1 and the memory location addressed by it are interchanged. In both places where `(reg:SI 1)` appears as a memory address it refers to the value in register 1 *before* the execution of the insn.

It follows that it is *incorrect* to use `parallel` and expect the result of one `set` to be available for the next one. For example, people sometimes attempt to represent a jump-if-zero instruction this way:

```
(parallel [(set (cc0) (reg:SI 34))
           (set (pc) (if_then_else
                       (eq (cc0) (const_int 0))
                       (label_ref ...)
                       (pc)))])
```

But this is incorrect, because it says that the jump condition depends on the condition code value *before* this instruction, not on the new value that is set by this instruction.

Peephole optimization, which takes place together with final assembly code output, can produce insns whose patterns consist of a `parallel` whose elements are the operands needed to output the resulting assembler code--often `reg`, `mem` or constant expressions. This would not be well-formed RTL at any other stage in compilation, but it is ok then because no further optimization remains to be done. However, the definition of the macro `NOTICE_UPDATE_CC`, if any, must deal with such insns if you define any peephole optimizations.

`(sequence [insns ...])`

Represents a sequence of insns. Each of the insns that appears in the vector is suitable for appearing in the chain of insns, so it must be an `insn`, `jump_insn`, `call_insn`, `code_label`, `barrier` or `note`.

A `sequence` RTX is never placed in an actual insn during RTL generation. It represents the sequence of insns that result from a `define_expand` *before* those insns are passed to `emit_insn` to insert them in the chain of insns. When actually inserted, the individual sub-insns are separated out and the `sequence` is forgotten.

After delay-slot scheduling is completed, an insn and all the insns that reside in its delay slots are grouped together into a `sequence`. The insn requiring the delay slot is the first insn in the vector; subsequent insns are to be placed in the delay slot.

`INSN_ANNULLED_BRANCH_P` is set on an insn in a delay slot to indicate that a branch insn should be used that will conditionally annul the effect of the insns in the delay slots. In such a case, `INSN_FROM_TARGET_P` indicates that the insn is from the target of the branch and should be executed only if the branch is taken; otherwise the insn should be executed only if the branch is

not taken. See section Delay Slot Scheduling.

These expression codes appear in place of a side effect, as the body of an insn, though strictly speaking they do not always describe side effects as such:

`(asm_input s)`

Represents literal assembler code as described by the string s.

`(unspec [operands ...] index)`

`(unspec_volatile [operands ...] index)`

Represents a machine-specific operation on operands. index selects between multiple machine-specific operations. `unspec_volatile` is used for volatile operations and operations that may trap; `unspec` is used for other operations.

These codes may appear inside a `pattern` of an insn, inside a `parallel`, or inside an expression.

`(addr_vec:m [lr0 lr1 ...])`

Represents a table of jump addresses. The vector elements lr0, etc., are `label_ref` expressions. The mode m specifies how much space is given to each address; normally m would be `Pmode`.

`(addr_diff_vec:m base [lr0 lr1 ...])`

Represents a table of jump addresses expressed as offsets from base. The vector elements lr0, etc., are `label_ref` expressions and so is base. The mode m specifies how much space is given to each address-difference.

Embedded Side-Effects on Addresses

Four special side-effect expression codes appear as memory addresses.

`(pre_dec:m x)`

Represents the side effect of decrementing x by a standard amount and represents also the value that x has after being decremented. x must be a `reg` or `mem`, but most machines allow only a `reg`. m must be the machine mode for pointers on the machine in use. The amount x is decremented by is the length in bytes of the machine mode of the containing memory reference of which this expression serves as the address. Here is an example of its use:

```
(mem:DF (pre_dec:SI (reg:SI 39)))
```

This says to decrement pseudo register 39 by the length of a `DFmode` value and use the result to address a `DFmode` value.

`(pre_inc:m x)`

Similar, but specifies incrementing x instead of decrementing it.

`(post_dec:m x)`

Represents the same side effect as `pre_dec` but a different value. The value represented here is the value x has *before* being decremented.

`(post_inc:m x)`

Similar, but specifies incrementing x instead of decrementing it.

These embedded side effect expressions must be used with care. Instruction patterns may not use them. Until the `flow' pass of the compiler, they may occur only to represent pushes onto the stack. The `flow' pass finds cases where registers are incremented or decremented in one instruction and used as an address shortly before or after; these cases are then transformed to use pre- or post-increment or -decrement.

If a register used as the operand of these expressions is used in another address in an insn, the original value of the register is used. Uses of the register outside of an address are not permitted within the same insn as a use in an embedded side effect expression because such insns behave differently on different machines and hence must be treated as ambiguous and disallowed.

An instruction that can be represented with an embedded side effect could also be represented using `parallel` containing an additional `set` to describe how the address register is altered. This is not done because machines that allow these operations at all typically allow them wherever a memory address is called for. Describing them as additional parallel stores would require doubling the number of entries in the machine description.

Assembler Instructions as Expressions

The RTX code `asm_operands` represents a value produced by a user-specified assembler instruction. It is used to represent an `asm` statement with arguments. An `asm` statement with a single output operand, like this:

```
asm ("foo %1,%2,%0" : "=a" (outputvar) : "g" (x + y), "di" (*z));
```

is represented using a single `asm_operands` RTX which represents the value that is stored in `outputvar`:

```
(set rtx-for-outputvar
     (asm_operands "foo %1,%2,%0" "a" 0
                   [rtx-for-addition-result rtx-for-*z]
                   [(asm_input:m1 "g")
                    (asm_input:m2 "di")]))
```

Here the operands of the `asm_operands` RTX are the assembler template string, the output-operand's constraint, the index-number of the output operand among the output operands specified, a vector of input operand RTX's, and a vector of input-operand modes and constraints. The mode m1 is the mode of the sum `x+y`; m2 is that of `*z`.

When an `asm` statement has multiple output values, its insn has several such `set` RTX's inside of a `parallel`. Each `set` contains a `asm_operands`; all of these share the same assembler template and vectors, but each contains the constraint for the respective output operand. They are also distinguished by the output-operand index number, which is 0, 1, ... for successive output operands.

Insns

The RTL representation of the code for a function is a doubly-linked chain of objects called insns. Insns are expressions with special codes that are used for no other purpose. Some insns are actual instructions; others represent dispatch tables for `switch` statements; others represent labels to jump to or various sorts of declarative information.

In addition to its own specific data, each insn must have a unique id-number that distinguishes it from all other insns in the current function (after delayed branch scheduling, copies of an insn with the same id-number may be present in multiple places in a function, but these copies will always be identical and will only appear inside a `sequence`), and chain pointers to the preceding and following insns. These three fields occupy the same position in every insn, independent of the expression code of the insn. They could be accessed with `XEXP` and `XINT`, but instead three special macros are always used:

`INSN_UID (i)`

Accesses the unique id of insn i.

`PREV_INSN (i)`

Accesses the chain pointer to the insn preceding i. If i is the first insn, this is a null pointer.

`NEXT_INSN (i)`

Accesses the chain pointer to the insn following i. If i is the last insn, this is a null pointer.

The first insn in the chain is obtained by calling `get_insns`; the last insn is the result of calling `get_last_insn`. Within the chain delimited by these insns, the `NEXT_INSN` and `PREV_INSN` pointers must always correspond: if insn is not the first insn,

```
NEXT_INSN (PREV_INSN (insn)) == insn
```

is always true and if insn is not the last insn,

```
PREV_INSN (NEXT_INSN (insn)) == insn
```

is always true.

After delay slot scheduling, some of the insns in the chain might be `sequence` expressions, which contain a vector of insns. The value of `NEXT_INSN` in all but the last of these insns is the next insn in the vector; the value of `NEXT_INSN` of the last insn in the vector is the same as the value of `NEXT_INSN` for the `sequence` in which it is contained. Similar rules apply for `PREV_INSN`.

This means that the above invariants are not necessarily true for insns inside `sequence` expressions. Specifically, if insn is the first insn in a `sequence`, `NEXT_INSN (PREV_INSN (insn))` is the insn containing the `sequence` expression, as is the value of `PREV_INSN (NEXT_INSN (insn))` is insn is the last insn in the `sequence` expression. You can use these expressions to find the containing `sequence` expression.

Every insn has one of the following six expression codes:

`insn`

The expression code `insn` is used for instructions that do not jump and do not do function calls. `sequence` expressions are always contained in insns with code `insn` even if one of those insns should jump or do function calls.

Insns with code `insn` have four additional fields beyond the three mandatory ones listed above. These four are described in a table below.

`jump_insn`

The expression code `jump_insn` is used for instructions that may jump (or, more generally, may contain `label_ref` expressions). If there is an instruction to return from the current function, it is recorded as a `jump_insn`.

`jump_insn` insns have the same extra fields as `insn` insns, accessed in the same way and in addition contain a field `JUMP_LABEL` which is defined once jump optimization has completed.

For simple conditional and unconditional jumps, this field contains the `code_label` to which this insn will (possibly conditionally) branch. In a more complex jump, `JUMP_LABEL` records one of the labels that the insn refers to; the only way to find the others is to scan the entire body of the insn.

Return insns count as jumps, but since they do not refer to any labels, they have zero in the `JUMP_LABEL` field.

`call_insn`

The expression code `call_insn` is used for instructions that may do function calls. It is important to distinguish these instructions because they imply that certain registers and memory locations may be altered unpredictably.

`call_insn` insns have the same extra fields as `insn` insns, accessed in the same way and in addition contain a field `CALL_INSN_FUNCTION_USAGE`, which contains a list (chain of `expr_list` expressions) containing `use` and `clobber` expressions that denote hard registers used or clobbered by the called function. A register specified in a `clobber` in this list is modified *after* the execution of the `call_insn`, while a register in a `clobber` in the body of the `call_insn` is clobbered before the insn completes execution. `clobber` expressions in this list augment registers specified in `CALL_USED_REGISTERS` (see section Basic Characteristics of Registers).

`code_label`

A `code_label` insn represents a label that a jump insn can jump to. It contains two special fields of data in addition to the three standard ones. `CODE_LABEL_NUMBER` is used to hold the label number, a number that identifies this label uniquely among all the labels in the compilation (not just in the current function). Ultimately, the label is represented in the assembler output as an assembler label, usually of the form `Ln' where n is the label number.

When a `code_label` appears in an RTL expression, it normally appears within a `label_ref` which represents the address of the label, as a number.

The field `LABEL_NUSES` is only defined once the jump optimization phase is completed and contains the number of times this label is referenced in the current function.

`barrier`

Barriers are placed in the instruction stream when control cannot flow past them. They are placed after unconditional jump instructions to indicate that the jumps are unconditional and after calls to `volatile` functions, which do not return (e.g., `exit`). They contain no information beyond the three standard fields.

`note`

`note` insns are used to represent additional debugging and declarative information. They contain two nonstandard fields, an integer which is accessed with the macro `NOTE_LINE_NUMBER` and a string accessed with `NOTE_SOURCE_FILE`.

If `NOTE_LINE_NUMBER` is positive, the note represents the position of a source line and `NOTE_SOURCE_FILE` is the source file name that the line came from. These notes control generation of line number data in the assembler output.

Otherwise, `NOTE_LINE_NUMBER` is not really a line number but a code with one of the following values (and `NOTE_SOURCE_FILE` must contain a null pointer):

`NOTE_INSN_DELETED`

Such a note is completely ignorable. Some passes of the compiler delete insns by altering them into notes of this kind.

`NOTE_INSN_BLOCK_BEG`

`NOTE_INSN_BLOCK_END`

These types of notes indicate the position of the beginning and end of a level of scoping of variable names. They control the output of debugging information.

`NOTE_INSN_LOOP_BEG`

`NOTE_INSN_LOOP_END`

These types of notes indicate the position of the beginning and end of a `while` or `for` loop. They enable the loop optimizer to find loops quickly.

`NOTE_INSN_LOOP_CONT`

Appears at the place in a loop that `continue` statements jump to.

`NOTE_INSN_LOOP_VTOP`

This note indicates the place in a loop where the exit test begins for those loops in which the exit test has been duplicated. This position becomes another virtual start of the loop when considering loop invariants.

`NOTE_INSN_FUNCTION_END`

Appears near the end of the function body, just before the label that `return` statements jump to (on machine where a single instruction does not suffice for returning). This note may be deleted by jump optimization.

`NOTE_INSN_SETJMP`

Appears following each call to `setjmp` or a related function.

These codes are printed symbolically when they appear in debugging dumps.

The machine mode of an insn is normally `VOIDmode`, but some phases use the mode for various purposes; for example, the reload pass sets it to `HImode` if the insn needs reloading but not register elimination and `QImode` if both are required. The common subexpression elimination pass sets the mode of an insn to `QImode` when it is the first insn in a block that has already been processed.

Here is a table of the extra fields of `insn`, `jump_insn` and `call_insn` insns:

`PATTERN (i)`

An expression for the side effect performed by this insn. This must be one of the following codes: `set`, `call`, `use`, `clobber`, `return`, `asm_input`, `asm_output`, `addr_vec`, `addr_diff_vec`, `trap_if`, `unspec`, `unspec_volatile`, `parallel`, or `sequence`. If it is a `parallel`, each element of the `parallel` must be one these codes, except that `parallel` expressions cannot be nested and `addr_vec` and `addr_diff_vec` are not permitted inside a `parallel` expression.

`INSN_CODE (i)`

An integer that says which pattern in the machine description matches this insn, or -1 if the matching has not yet been attempted.

Such matching is never attempted and this field remains -1 on an insn whose pattern consists of a single `use`, `clobber`, `asm_input`, `addr_vec` or `addr_diff_vec` expression.

Matching is also never attempted on insns that result from an `asm` statement. These contain at least one `asm_operands` expression. The function `asm_noperands` returns a non-negative value for such insns.

In the debugging output, this field is printed as a number followed by a symbolic representation that locates the pattern in the `` `md' `` file as some small positive or negative offset from a named pattern.

`LOG_LINKS (i)`

A list (chain of `insn_list` expressions) giving information about dependencies between instructions within a basic block. Neither a jump nor a label may come between the related insns.

`REG_NOTES (i)`

A list (chain of `expr_list` and `insn_list` expressions) giving miscellaneous information about the insn. It is often information pertaining to the registers used in this insn.

The `LOG_LINKS` field of an insn is a chain of `insn_list` expressions. Each of these has two operands: the first is an insn, and the second is another `insn_list` expression (the next one in the chain). The last `insn_list` in the chain has a null pointer as second operand. The significant thing about the chain is which insns appear in it (as first operands of `insn_list` expressions). Their order is not significant.

This list is originally set up by the flow analysis pass; it is a null pointer until then. Flow only adds links for those data dependencies which can be used for instruction combination. For each insn, the flow analysis pass adds a link to insns which store into registers values that are used for the first time in this insn. The instruction scheduling pass adds extra links so that every dependence will be represented. Links represent data dependencies, antidependencies and output dependencies; the machine mode of the link distinguishes these three types: antidependencies have mode `REG_DEP_ANTI`, output dependencies

have mode `REG_DEP_OUTPUT`, and data dependencies have mode `VOIDmode`.

The `REG_NOTES` field of an insn is a chain similar to the `LOG_LINKS` field but it includes `expr_list` expressions in addition to `insn_list` expressions. There are several kinds of register notes, which are distinguished by the machine mode, which in a register note is really understood as being an `enum reg_note`. The first operand op of the note is data whose meaning depends on the kind of note.

The macro `REG_NOTE_KIND (x)` returns the kind of register note. Its counterpart, the macro `PUT_REG_NOTE_KIND (x, newkind)` sets the register note type of x to be newkind.

Register notes are of three classes: They may say something about an input to an insn, they may say something about an output of an insn, or they may create a linkage between two insns. There are also a set of values that are only used in `LOG_LINKS`.

These register notes annotate inputs to an insn:

`REG_DEAD`

The value in op dies in this insn; that is to say, altering the value immediately after this insn would not affect the future behavior of the program.

This does not necessarily mean that the register op has no useful value after this insn since it may also be an output of the insn. In such a case, however, a `REG_DEAD` note would be redundant and is usually not present until after the reload pass, but no code relies on this fact.

`REG_INC`

The register op is incremented (or decremented; at this level there is no distinction) by an embedded side effect inside this insn. This means it appears in a `post_inc`, `pre_inc`, `post_dec` or `pre_dec` expression.

`REG_NONNEG`

The register op is known to have a nonnegative value when this insn is reached. This is used so that decrement and branch until zero instructions, such as the m68k dbra, can be matched.

The `REG_NONNEG` note is added to insns only if the machine description has a `decrement_and_branch_until_zero' pattern.

`REG_NO_CONFLICT`

This insn does not cause a conflict between op and the item being set by this insn even though it might appear that it does. In other words, if the destination register and op could otherwise be assigned the same register, this insn does not prevent that assignment.

Insns with this note are usually part of a block that begins with a `clobber` insn specifying a multi-word pseudo register (which will be the output of the block), a group of insns that each set one word of the value and have the `REG_NO_CONFLICT` note attached, and a final insn that copies the output to itself with an attached `REG_EQUAL` note giving the expression being computed. This block is encapsulated with `REG_LIBCALL` and `REG_RETVAL` notes on the first and last insns, respectively.

`REG_LABEL`

This insn uses op, a `code_label`, but is not a `jump_insn`. The presence of this note allows

jump optimization to be aware that op is, in fact, being used.

The following notes describe attributes of outputs of an insn:

`REG_EQUIV`

`REG_EQUAL`

This note is only valid on an insn that sets only one register and indicates that that register will be equal to op at run time; the scope of this equivalence differs between the two types of notes. The value which the insn explicitly copies into the register may look different from op, but they will be equal at run time. If the output of the single `set` is a `strict_low_part` expression, the note refers to the register that is contained in `SUBREG_REG` of the `subreg` expression. For `REG_EQUIV`, the register is equivalent to op throughout the entire function, and could validly be replaced in all its occurrences by op. ("Validly" here refers to the data flow of the program; simple replacement may make some insns invalid.) For example, when a constant is loaded into a register that is never assigned any other value, this kind of note is used.

When a parameter is copied into a pseudo-register at entry to a function, a note of this kind records that the register is equivalent to the stack slot where the parameter was passed. Although in this case the register may be set by other insns, it is still valid to replace the register by the stack slot throughout the function.

In the case of `REG_EQUAL`, the register that is set by this insn will be equal to op at run time at the end of this insn but not necessarily elsewhere in the function. In this case, op is typically an arithmetic expression. For example, when a sequence of insns such as a library call is used to perform an arithmetic operation, this kind of note is attached to the insn that produces or copies the final value.

These two notes are used in different ways by the compiler passes. `REG_EQUAL` is used by passes prior to register allocation (such as common subexpression elimination and loop optimization) to tell them how to think of that value. `REG_EQUIV` notes are used by register allocation to indicate that there is an available substitute expression (either a constant or a `mem` expression for the location of a parameter on the stack) that may be used in place of a register if insufficient registers are available.

Except for stack homes for parameters, which are indicated by a `REG_EQUIV` note and are not useful to the early optimization passes and pseudo registers that are equivalent to a memory location throughout there entire life, which is not detected until later in the compilation, all equivalences are initially indicated by an attached `REG_EQUAL` note. In the early stages of register allocation, a `REG_EQUAL` note is changed into a `REG_EQUIV` note if op is a constant and the insn represents the only set of its destination register.

Thus, compiler passes prior to register allocation need only check for `REG_EQUAL` notes and passes subsequent to register allocation need only check for `REG_EQUIV` notes.

`REG_UNUSED`

The register op being set by this insn will not be used in a subsequent insn. This differs from a `REG_DEAD` note, which indicates that the value in an input will not be used subsequently. These two notes are independent; both may be present for the same register.

`REG_WAS_0`

The single output of this insn contained zero before this insn. op is the insn that set it to zero. You can rely on this note if it is present and op has not been deleted or turned into a `note`; its absence implies nothing.

These notes describe linkages between insns. They occur in pairs: one insn has one of a pair of notes that points to a second insn, which has the inverse note pointing back to the first insn.

`REG_RETVAL`

This insn copies the value of a multi-insn sequence (for example, a library call), and op is the first insn of the sequence (for a library call, the first insn that was generated to set up the arguments for the library call).

Loop optimization uses this note to treat such a sequence as a single operation for code motion purposes and flow analysis uses this note to delete such sequences whose results are dead.

A `REG_EQUAL` note will also usually be attached to this insn to provide the expression being computed by the sequence.

`REG_LIBCALL`

This is the inverse of `REG_RETVAL`: it is placed on the first insn of a multi-insn sequence, and it points to the last one.

`REG_CC_SETTER`

`REG_CC_USER`

On machines that use `cc0`, the insns which set and use `cc0` set and use `cc0` are adjacent. However, when branch delay slot filling is done, this may no longer be true. In this case a `REG_CC_USER` note will be placed on the insn setting `cc0` to point to the insn using `cc0` and a `REG_CC_SETTER` note will be placed on the insn using `cc0` to point to the insn setting `cc0`.

These values are only used in the `LOG_LINKS` field, and indicate the type of dependency that each link represents. Links which indicate a data dependence (a read after write dependence) do not use any code, they simply have mode `VOIDmode`, and are printed without any descriptive text.

`REG_DEP_ANTI`

This indicates an anti dependence (a write after read dependence).

`REG_DEP_OUTPUT`

This indicates an output dependence (a write after write dependence).

For convenience, the machine mode in an `insn_list` or `expr_list` is printed using these symbolic codes in debugging dumps.

The only difference between the expression codes `insn_list` and `expr_list` is that the first operand of an `insn_list` is assumed to be an insn and is printed in debugging dumps as the insn's unique id; the first operand of an `expr_list` is printed in the ordinary way as an expression.

RTL Representation of Function-Call Insns

Insns that call subroutines have the RTL expression code `call_insn`. These insns must satisfy special rules, and their bodies must use a special RTL expression code, `call`.

A `call` expression has two operands, as follows:

```
(call (mem:fm addr) nbytes)
```

Here nbytes is an operand that represents the number of bytes of argument data being passed to the subroutine, fm is a machine mode (which must equal as the definition of the `FUNCTION_MODE` macro in the machine description) and addr represents the address of the subroutine.

For a subroutine that returns no value, the `call` expression as shown above is the entire body of the insn, except that the insn might also contain `use` or `clobber` expressions.

For a subroutine that returns a value whose mode is not `BLKmode`, the value is returned in a hard register. If this register's number is r, then the body of the call insn looks like this:

```
(set (reg:m r)
     (call (mem:fm addr) nbytes))
```

This RTL expression makes it clear (to the optimizer passes) that the appropriate register receives a useful value in this insn.

When a subroutine returns a `BLKmode` value, it is handled by passing to the subroutine the address of a place to store the value. So the call insn itself does not "return" any value, and it has the same RTL form as a call that returns nothing.

On some machines, the call instruction itself clobbers some register, for example to contain the return address. `call_insn` insns on these machines should have a body which is a `parallel` that contains both the `call` expression and `clobber` expressions that indicate which registers are destroyed. Similarly, if the call instruction requires some register other than the stack pointer that is not explicitly mentioned it its RTL, a `use` subexpression should mention that register.

Functions that are called are assumed to modify all registers listed in the configuration macro `CALL_USED_REGISTERS` (see section Basic Characteristics of Registers) and, with the exception of `const` functions and library calls, to modify all of memory.

Insns containing just `use` expressions directly precede the `call_insn` insn to indicate which registers contain inputs to the function. Similarly, if registers other than those in `CALL_USED_REGISTERS` are clobbered by the called function, insns containing a single `clobber` follow immediately after the call to indicate which registers.

Structure Sharing Assumptions

The compiler assumes that certain kinds of RTL expressions are unique; there do not exist two distinct objects representing the same value. In other cases, it makes an opposite assumption: that no RTL expression object of a certain kind appears in more than one place in the containing structure.

These assumptions refer to a single function; except for the RTL objects that describe global variables and external functions, and a few standard objects such as small integer constants, no RTL objects are common to two functions.

- Each pseudo-register has only a single `reg` object to represent it, and therefore only a single machine mode.
- For any symbolic label, there is only one `symbol_ref` object referring to it.
- There is only one `const_int` expression with value 0, only one with value 1, and only one with value -1. Some other integer values are also stored uniquely.
- There is only one `pc` expression.
- There is only one `cc0` expression.
- There is only one `const_double` expression with value 0 for each floating point mode. Likewise for values 1 and 2.
- No `label_ref` or `scratch` appears in more than one place in the RTL structure; in other words, it is safe to do a tree-walk of all the insns in the function and assume that each time a `label_ref` or `scratch` is seen it is distinct from all others that are seen.
- Only one `mem` object is normally created for each static variable or stack slot, so these objects are frequently shared in all the places they appear. However, separate but equal objects for these variables are occasionally made.
- When a single `asm` statement has multiple output operands, a distinct `asm_operands` expression is made for each output operand. However, these all share the vector which contains the sequence of input operands. This sharing is used later on to test whether two `asm_operands` expressions come from the same statement, so all optimizations must carefully preserve the sharing if they copy the vector at all.
- No RTL object appears in more than one place in the RTL structure except as described above. Many passes of the compiler rely on this by assuming that they can modify RTL objects in place without unwanted side-effects on other insns.
- During initial RTL generation, shared structure is freely introduced. After all the RTL for a function has been generated, all shared structure is copied by `unshare_all_rtl` in `` `emit-rtl.c' ``, after which the above rules are guaranteed to be followed.
- During the combiner pass, shared structure within an insn can exist temporarily. However, the shared structure is copied before the combiner is finished with the insn. This is done by calling `copy_rtx_if_shared`, which is a subroutine of `unshare_all_rtl`.

Reading RTL

To read an RTL object from a file, call `read_rtx`. It takes one argument, a stdio stream, and returns a single RTL object.

Reading RTL from a file is very slow. This is not currently a problem since reading RTL occurs only as part of building the compiler.

People frequently have the idea of using RTL stored as text in a file as an interface between a language front end and the bulk of GNU CC. This idea is not feasible.

GNU CC was designed to use RTL internally only. Correct RTL for a given program is very dependent on the particular target machine. And the RTL does not contain all the information about the program.

The proper way to interface GNU CC to a new language front end is with the "tree" data structure. There is no manual for this data structure, but it is described in the files `` `tree.h' `` and `` `tree.def' ``.

Go to the previous, next section.

Go to the previous, next section.

Machine Descriptions

A machine description has two parts: a file of instruction patterns (`.md' file) and a C header file of macro definitions.

The `.md' file for a target machine contains a pattern for each instruction that the target machine supports (or at least each instruction that is worth telling the compiler about). It may also contain comments. A semicolon causes the rest of the line to be a comment, unless the semicolon is inside a quoted string.

See the next chapter for information on the C header file.

Everything about Instruction Patterns

Each instruction pattern contains an incomplete RTL expression, with pieces to be filled in later, operand constraints that restrict how the pieces can be filled in, and an output pattern or C code to generate the assembler output, all wrapped up in a `define_insn` expression.

A `define_insn` is an RTL expression containing four or five operands:

1. An optional name. The presence of a name indicate that this instruction pattern can perform a certain standard job for the RTL-generation pass of the compiler. This pass knows certain names and will use the instruction patterns with those names, if the names are defined in the machine description.

   The absence of a name is indicated by writing an empty string where the name should go. Nameless instruction patterns are never used for generating RTL code, but they may permit several simpler insns to be combined later on.

   Names that are not thus known and used in RTL-generation have no effect; they are equivalent to no name at all.
2. The RTL template (see section RTL Template) is a vector of incomplete RTL expressions which show what the instruction should look like. It is incomplete because it may contain `match_operand`, `match_operator`, and `match_dup` expressions that stand for operands of the instruction.

   If the vector has only one element, that element is the template for the instruction pattern. If the vector has multiple elements, then the instruction pattern is a `parallel` expression containing the elements described.
3. A condition. This is a string which contains a C expression that is the final test to decide whether an insn body matches this pattern.

   For a named pattern, the condition (if present) may not depend on the data in the insn being matched, but only the target-machine-type flags. The compiler needs to test these conditions during initialization in order to learn exactly which named instructions are available in a particular run.

   For nameless patterns, the condition is applied only when matching an individual insn, and only after the insn has matched the pattern's recognition template. The insn's operands may be found in the vector `operands`.

4. The output template: a string that says how to output matching insns as assembler code. `%' in this string specifies where to substitute the value of an operand. See section Output Templates and Operand Substitution.

   When simple substitution isn't general enough, you can specify a piece of C code to compute the output. See section C Statements for Assembler Output.

5. Optionally, a vector containing the values of attributes for insns matching this pattern. See section Instruction Attributes.

Example of `define_insn`

Here is an actual example of an instruction pattern, for the 68000/68020.

```
(define_insn "tstsi"
  [(set (cc0)
        (match_operand:SI 0 "general_operand" "rm"))]
  ""
  "*
{ if (TARGET_68020 || ! ADDRESS_REG_P (operands[0]))
    return \"tstl %0\";
  return \"cmpl #0,%0\"; }")
```

This is an instruction that sets the condition codes based on the value of a general operand. It has no condition, so any insn whose RTL description has the form shown may be handled according to this pattern. The name `tstsi' means "test a `SImode` value" and tells the RTL generation pass that, when it is necessary to test such a value, an insn to do so can be constructed using this pattern.

The output control string is a piece of C code which chooses which output template to return based on the kind of operand and the specific type of CPU for which code is being generated.

`"rm"' is an operand constraint. Its meaning is explained below.

RTL Template

The RTL template is used to define which insns match the particular pattern and how to find their operands. For named patterns, the RTL template also says how to construct an insn from specified operands.

Construction involves substituting specified operands into a copy of the template. Matching involves determining the values that serve as the operands in the insn being matched. Both of these activities are controlled by special expression types that direct matching and substitution of the operands.

`(match_operand:m n predicate constraint)`

This expression is a placeholder for operand number n of the insn. When constructing an insn, operand number n will be substituted at this point. When matching an insn, whatever appears at this position in the insn will be taken as operand number n; but it must satisfy predicate or this instruction pattern will not match at all.

Operand numbers must be chosen consecutively counting from zero in each instruction pattern. Th may be only one `match_operand` expression in the pattern for each operand number. Usually operands are numbered in the order of appearance in `match_operand` expressions.

predicate is a string that is the name of a C function that accepts two arguments, an expression and a machine mode. During matching, the function will be called with the putative operand as the expression and m as the mode argument (if m is not specified, `VOIDmode` will be used, which normally causes predicate to accept any mode). If it returns zero, this instruction pattern fails to match. predicate may be an empty string; then it means no test is to be done on the operand, so anything which occurs in this position is valid.

Most of the time, predicate will reject modes other than m---but not always. For example, the predicate `address_operand` uses m as the mode of memory ref that the address should be valid for. Many predicates accept `const_int` nodes even though their mode is `VOIDmode`.

constraint controls reloading and the choice of the best register class to use for a value, as explained later (see section Operand Constraints).

People are often unclear on the difference between the constraint and the predicate. The predicate helps decide whether a given insn matches the pattern. The constraint plays no role in this decision; instead, it controls various decisions in the case of an insn which does match.

On CISC machines, the most common predicate is `"general_operand"`. This function checks that the putative operand is either a constant, a register or a memory reference, and that it is valid for mode m.

For an operand that must be a register, predicate should be `"register_operand"`. Using `"general_operand"` would be valid, since the reload pass would copy any non-register operands through registers, but this would make GNU CC do extra work, it would prevent invariant operands (such as constant) from being removed from loops, and it would prevent the register allocator from doing the best possible job. On RISC machines, it is usually most efficient to allow predicate to accept only objects that the constraints allow.

For an operand that must be a constant, you must be sure to either use `"immediate_operand"` for predicate, or make the instruction pattern's extra condition require a constant, or both. You cannot expect the constraints to do this work! If the constraints allow only constants, but the predicate allows something else, the compiler will crash when that case arises.

`(match_scratch:m n constraint)`

This expression is also a placeholder for operand number n and indicates that operand must be a `scratch` or `reg` expression.

When matching patterns, this is equivalent to

```
(match_operand:m n "scratch_operand" pred)
```

but, when generating RTL, it produces a (`scratch`:m) expression.

If the last few expressions in a `parallel` are `clobber` expressions whose operands are either a hard register or `match_scratch`, the combiner can add or delete them when necessary. See section Side Effect Expressions.

`(match_dup n)`

This expression is also a placeholder for operand number n. It is used when the operand needs to appear more than once in the insn.

In construction, `match_dup` acts just like `match_operand`: the operand is substituted into the insn being constructed. But in matching, `match_dup` behaves differently. It assumes that operand number n has already been determined by a `match_operand` appearing earlier in the recognition template, and it matches only an identical-looking expression.

`(match_operator:m n predicate [operands...])`

This pattern is a kind of placeholder for a variable RTL expression code.

When constructing an insn, it stands for an RTL expression whose expression code is taken from that of operand n, and whose operands are constructed from the patterns operands.

When matching an expression, it matches an expression if the function predicate returns nonzero on that expression *and* the patterns operands match the operands of the expression.

Suppose that the function `commutative_operator` is defined as follows, to match any expression whose operator is one of the commutative arithmetic operators of RTL and whose mode is mode:

```
int
commutative_operator (x, mode)
     rtx x;
     enum machine_mode mode;
{
  enum rtx_code code = GET_CODE (x);
  if (GET_MODE (x) != mode)
    return 0;
  return (GET_RTX_CLASS (code) == 'c'
          || code == EQ || code == NE);
}
```

Then the following pattern will match any RTL expression consisting of a commutative operator applied to two general operands:

```
(match_operator:SI 3 "commutative_operator"
  [(match_operand:SI 1 "general_operand" "g")
   (match_operand:SI 2 "general_operand" "g")])
```

Here the vector `[operands...]` contains two patterns because the expressions to be matched all contain two operands.

When this pattern does match, the two operands of the commutative operator are recorded as operands 1 and 2 of the insn. (This is done by the two instances of `match_operand`.) Operand 3 of the insn will be the entire commutative expression: use `GET_CODE (operands[3])` to see which commutative operator was used.

The machine mode m of `match_operator` works like that of `match_operand`: it is passed as the second argument to the predicate function, and that function is solely responsible for deciding whether

the expression to be matched "has" that mode.

When constructing an insn, argument 3 of the gen-function will specify the operation (i.e. the expression code) for the expression to be made. It should be an RTL expression, whose expression code is copied into a new expression whose operands are arguments 1 and 2 of the gen-function. The subexpressions of argument 3 are not used; only its expression code matters.

When `match_operator` is used in a pattern for matching an insn, it usually best if the operand number of the `match_operator` is higher than that of the actual operands of the insn. This improves register allocation because the register allocator often looks at operands 1 and 2 of insns to see if it can do register tying.

There is no way to specify constraints in `match_operator`. The operand of the insn which corresponds to the `match_operator` never has any constraints because it is never reloaded as a whole. However, if parts of its operands are matched by `match_operand` patterns, those parts may have constraints of their own.

`(match_op_dup:m n[operands...])`

Like `match_dup`, except that it applies to operators instead of operands. When constructing an insn, operand number n will be substituted at this point. But in matching, `match_op_dup` behaves differently. It assumes that operand number n has already been determined by a `match_operator` appearing earlier in the recognition template, and it matches only an identical-looking expression.

`(match_parallel n predicate [subpat...])`

This pattern is a placeholder for an insn that consists of a `parallel` expression with a variable number of elements. This expression should only appear at the top level of an insn pattern.

When constructing an insn, operand number n will be substituted at this point. When matching an insn, it matches if the body of the insn is a `parallel` expression with at least as many elements as the vector of subpat expressions in the `match_parallel`, if each subpat matches the corresponding element of the `parallel`, *and* the function predicate returns nonzero on the `parallel` that is the body of the insn. It is the responsibility of the predicate to validate elements of the `parallel` beyond those listed in the `match_parallel`.

A typical use of `match_parallel` is to match load and store multiple expressions, which can contain a variable number of elements in a `parallel`. For example,

```
(define_insn ""
  [(match_parallel 0 "load_multiple_operation"
     [(set (match_operand:SI 1 "gpc_reg_operand" "=r")
           (match_operand:SI 2 "memory_operand" "m"))
      (use (reg:SI 179))
      (clobber (reg:SI 179))])]
  ""
  "loadm 0,0,%1,%2")
```

This example comes from `a29k.md'. The function `load_multiple_operations` is defined in `a29k.c' and checks that subsequent elements in the `parallel` are the same as the `set` in the pattern, except that they are referencing subsequent registers and memory locations.

An insn that matches this pattern might look like:

```
(parallel
 [(set (reg:SI 20) (mem:SI (reg:SI 100)))
  (use (reg:SI 179))
  (clobber (reg:SI 179))
  (set (reg:SI 21)
       (mem:SI (plus:SI (reg:SI 100)
                        (const_int 4))))
  (set (reg:SI 22)
       (mem:SI (plus:SI (reg:SI 100)
                        (const_int 8))))])
```

`(match_par_dup n [subpat...])`

Like `match_op_dup`, but for `match_parallel` instead of `match_operator`.

`(address (match_operand:m n "address_operand" ""))`

This complex of expressions is a placeholder for an operand number n in a "load address" instruction: an operand which specifies a memory location in the usual way, but for which the actual operand value used is the address of the location, not the contents of the location.

`address` expressions never appear in RTL code, only in machine descriptions. And they are used only in machine descriptions that do not use the operand constraint feature. When operand constraints are in use, the letter `p' in the constraint serves this purpose.

m is the machine mode of the *memory location being addressed*, not the machine mode of the address itself. That mode is always the same on a given target machine (it is `Pmode`, which normally is `SImode`), so there is no point in mentioning it; thus, no machine mode is written in the `address` expression. If some day support is added for machines in which addresses of different kinds of objects appear differently or are used differently (such as the PDP-10), different formats would perhaps need different machine modes and these modes might be written in the `address` expression.

Output Templates and Operand Substitution

The output template is a string which specifies how to output the assembler code for an instruction pattern. Most of the template is a fixed string which is output literally. The character `%' is used to specify where to substitute an operand; it can also be used to identify places where different variants of the assembler require different syntax.

In the simplest case, a `%' followed by a digit n says to output operand n at that point in the string.

`%' followed by a letter and a digit says to output an operand in an alternate fashion. Four letters have standard, built-in meanings described below. The machine description macro `PRINT_OPERAND` can define additional letters with nonstandard meanings.

`%cdigit' can be used to substitute an operand that is a constant value without the syntax that normally indicates an immediate operand.

`%ndigit' is like `%cdigit' except that the value of the constant is negated before printing.

`%adigit' can be used to substitute an operand as if it were a memory reference, with the actual operand treated as the address. This may be useful when outputting a "load address" instruction, because often the assembler syntax for such an instruction requires you to write the operand as if it were a memory reference.

`%ldigit' is used to substitute a `label_ref` into a jump instruction.

`%=' outputs a number which is unique to each instruction in the entire compilation. This is useful for making local labels to be referred to more than once in a single template that generates multiple assembler instructions.

`%' followed by a punctuation character specifies a substitution that does not use an operand. Only one case is standard: `%%' outputs a `%' into the assembler code. Other nonstandard cases can be defined in the `PRINT_OPERAND` macro. You must also define which punctuation characters are valid with the `PRINT_OPERAND_PUNCT_VALID_P` macro.

The template may generate multiple assembler instructions. Write the text for the instructions, with `\;' between them.

When the RTL contains two operands which are required by constraint to match each other, the output template must refer only to the lower-numbered operand. Matching operands are not always identical, and the rest of the compiler arranges to put the proper RTL expression for printing into the lower-numbered operand.

One use of nonstandard letters or punctuation following `%' is to distinguish between different assembler languages for the same machine; for example, Motorola syntax versus MIT syntax for the 68000. Motorola syntax requires periods in most opcode names, while MIT syntax does not. For example, the opcode `movel' in MIT syntax is `move.l' in Motorola syntax. The same file of patterns is used for both kinds of output syntax, but the character sequence `%.' is used in each place where Motorola syntax wants a period. The `PRINT_OPERAND` macro for Motorola syntax defines the sequence to output a period; the macro for MIT syntax defines it to do nothing.

As a special case, a template consisting of the single character # instructs the compiler to first split the insn, and then output the resulting instructions separately. This helps eliminate redundancy in the output templates. If you have a `define_insn` that needs to emit multiple assembler instructions, and there is an matching `define_split` already defined, then you can simply use # as the output template instead of writing an output template that emits the multiple assembler instructions.

If `ASSEMBLER_DIALECT` is defined, you can use `{option0|option1|option2}' constructs in the templates. These describe multiple variants of assembler language syntax. See section Output of Assembler Instructions.

C Statements for Assembler Output

Often a single fixed template string cannot produce correct and efficient assembler code for all the cases that are recognized by a single instruction pattern. For example, the opcodes may depend on the kinds of operands; or some unfortunate combinations of operands may require extra machine instructions.

If the output control string starts with a `@', then it is actually a series of templates, each on a separate line. (Blank lines and leading spaces and tabs are ignored.) The templates correspond to the pattern's constraint

alternatives (see section Multiple Alternative Constraints). For example, if a target machine has a two-address add instruction `addr' to add into a register and another `addm' to add a register to memory, you might write this pattern:

```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "general_operand" "=r,m")
        (plus:SI (match_operand:SI 1 "general_operand" "0,0")
                 (match_operand:SI 2 "general_operand" "g,r")))]
  ""
  "@
   addr %2,%0
   addm %2,%0")
```

If the output control string starts with a `*', then it is not an output template but rather a piece of C program that should compute a template. It should execute a `return` statement to return the template-string you want. Most such templates use C string literals, which require doublequote characters to delimit them. To include these doublequote characters in the string, prefix each one with `\'.

The operands may be found in the array `operands`, whose C data type is `rtx []`.

It is very common to select different ways of generating assembler code based on whether an immediate operand is within a certain range. Be careful when doing this, because the result of `INTVAL` is an integer on the host machine. If the host machine has more bits in an `int` than the target machine has in the mode in which the constant will be used, then some of the bits you get from `INTVAL` will be superfluous. For proper results, you must carefully disregard the values of those bits.

It is possible to output an assembler instruction and then go on to output or compute more of them, using the subroutine `output_asm_insn`. This receives two arguments: a template-string and a vector of operands. The vector may be `operands`, or it may be another array of `rtx` that you declare locally and initialize yourself.

When an insn pattern has multiple alternatives in its constraints, often the appearance of the assembler code is determined mostly by which alternative was matched. When this is so, the C code can test the variable `which_alternative`, which is the ordinal number of the alternative that was actually satisfied (0 for the first, 1 for the second alternative, etc.).

For example, suppose there are two opcodes for storing zero, `clrreg' for registers and `clrmem' for memory locations. Here is how a pattern could use `which_alternative` to choose between them:

```
(define_insn ""
  [(set (match_operand:SI 0 "general_operand" "=r,m")
        (const_int 0))]
  ""
  "*
  return (which_alternative == 0
          ? \"clrreg %0\" : \"clrmem %0\");
  ")
```

The example above, where the assembler code to generate was *solely* determined by the alternative, could

also have been specified as follows, having the output control string start with a `@':

```
(define_insn ""
  [(set (match_operand:SI 0 "general_operand" "=r,m")
        (const_int 0))]
  ""
  "@
   clrreg %0
   clrmem %0")
```

Operand Constraints

Each `match_operand` in an instruction pattern can specify a constraint for the type of operands allowed. Constraints can say whether an operand may be in a register, and which kinds of register; whether the operand can be a memory reference, and which kinds of address; whether the operand may be an immediate constant, and which possible values it may have. Constraints can also require two operands to match.

Simple Constraints

The simplest kind of constraint is a string full of letters, each of which describes one kind of operand that is permitted. Here are the letters that are allowed:

`m'

A memory operand is allowed, with any kind of address that the machine supports in general.

`o'

A memory operand is allowed, but only if the address is offsettable. This means that adding a small integer (actually, the width in bytes of the operand, as determined by its machine mode) may be added to the address and the result is also a valid memory address.

For example, an address which is constant is offsettable; so is an address that is the sum of a register and a constant (as long as a slightly larger constant is also within the range of address-offsets supported by the machine); but an autoincrement or autodecrement address is not offsettable. More complicated indirect/indexed addresses may or may not be offsettable depending on the other addressing modes that the machine supports.

Note that in an output operand which can be matched by another operand, the constraint letter `o' is valid only when accompanied by both `<' (if the target machine has predecrement addressing) and `>' (if the target machine has preincrement addressing).

`V'

A memory operand that is not offsettable. In other words, anything that would fit the `m' constraint but not the `o' constraint.

`<'

A memory operand with autodecrement addressing (either predecrement or postdecrement) is allowed.

`>'

A memory operand with autoincrement addressing (either preincrement or postincrement) is allowed.

`r'

A register operand is allowed provided that it is in a general register.

`d', `a', `f', ...

Other letters can be defined in machine-dependent fashion to stand for particular classes of registers. `d', `a' and `f' are defined on the 68000/68020 to stand for data, address and floating point registers.

`i'

An immediate integer operand (one with constant value) is allowed. This includes symbolic constants whose values will be known only at assembly time.

`n'

An immediate integer operand with a known numeric value is allowed. Many systems cannot support assembly-time constants for operands less than a word wide. Constraints for these operands should use `n' rather than `i'.

`I', `J', `K', ... `P'

Other letters in the range `I' through `P' may be defined in a machine-dependent fashion to permit immediate integer operands with explicit integer values in specified ranges. For example, on the 68000, `I' is defined to stand for the range of values 1 to 8. This is the range permitted as a shift count in the shift instructions.

`E'

An immediate floating operand (expression code `const_double`) is allowed, but only if the target floating point format is the same as that of the host machine (on which the compiler is running).

`F'

An immediate floating operand (expression code `const_double`) is allowed.

`G', `H'

`G' and `H' may be defined in a machine-dependent fashion to permit immediate floating operands in particular ranges of values.

`s'

An immediate integer operand whose value is not an explicit integer is allowed.

This might appear strange; if an insn allows a constant operand with a value not known at compile time, it certainly must allow any known value. So why use `s' instead of `i'? Sometimes it allows better code to be generated.

For example, on the 68000 in a fullword instruction it is possible to use an immediate operand; but if the immediate value is between -128 and 127, better code results from loading the value into a register and using the register. This is because the load into the register can be done with a `moveq' instruction. We arrange for this to happen by defining the letter `K' to mean "any integer outside the range -128 to 127", and then specifying `Ks' in the operand constraints.

`g'

Any register, memory or immediate integer operand is allowed, except for registers that are not general registers.

`X'

Any operand whatsoever is allowed, even if it does not satisfy `general_operand`. This is normally used in the constraint of a `match_scratch` when certain alternatives will not actually require a

scratch register.

`0', `1', `2', ... `9'

An operand that matches the specified operand number is allowed. If a digit is used together with letters within the same alternative, the digit should come last.

This is called a matching constraint and what it really means is that the assembler has only a single operand that fills two roles considered separate in the RTL insn. For example, an add insn has two input operands and one output operand in the RTL, but on most CISC machines an add instruction really has only two operands, one of them an input-output operand:

```
addl #35,r12
```

Matching constraints are used in these circumstances. More precisely, the two operands that match must include one input-only operand and one output-only operand. Moreover, the digit must be a smaller number than the number of the operand that uses it in the constraint.

For operands to match in a particular case usually means that they are identical-looking RTL expressions. But in a few special cases specific kinds of dissimilarity are allowed. For example, `*x` as an input operand will match `*x++` as an output operand. For proper results in such cases, the output template should always use the output-operand's number when printing the operand.

`p'

An operand that is a valid memory address is allowed. This is for "load address" and "push address" instructions.

`p' in the constraint must be accompanied by `address_operand` as the predicate in the `match_operand`. This predicate interprets the mode specified in the `match_operand` as the mode of the memory reference for which the address would be valid.

`Q', `R', `S', ... `U'

Letters in the range `Q' through `U' may be defined in a machine-dependent fashion to stand for arbitrary operand types. The machine description macro `EXTRA_CONSTRAINT` is passed the operand as its first argument and the constraint letter as its second operand.

A typical use for this would be to distinguish certain types of memory references that affect other insn operands.

Do not define these constraint letters to accept register references (`reg`); the reload pass does not expect this and would not handle it properly.

In order to have valid assembler code, each operand must satisfy its constraint. But a failure to do so does not prevent the pattern from applying to an insn. Instead, it directs the compiler to modify the code so that the constraint will be satisfied. Usually this is done by copying an operand into a register.

Contrast, therefore, the two instruction patterns that follow:

```
(define_insn ""
  [(set (match_operand:SI 0 "general_operand" "=r")
        (plus:SI (match_dup 0)
                 (match_operand:SI 1 "general_operand" "r")))]
```


```
  ""
  "...")
```

which has two operands, one of which must appear in two places, and

```
(define_insn ""
  [(set (match_operand:SI 0 "general_operand" "=r")
        (plus:SI (match_operand:SI 1 "general_operand" "0")
                 (match_operand:SI 2 "general_operand" "r")))]
  ""
  "...")
```

which has three operands, two of which are required by a constraint to be identical. If we are considering an insn of the form

```
(insn n prev next
  (set (reg:SI 3)
       (plus:SI (reg:SI 6) (reg:SI 109)))
  ...)
```

the first pattern would not apply at all, because this insn does not contain two identical subexpressions in the right place. The pattern would say, "That does not look like an add instruction; try other patterns." The second pattern would say, "Yes, that's an add instruction, but there is something wrong with it." It would direct the reload pass of the compiler to generate additional insns to make the constraint true. The results might look like this:

```
(insn n2 prev n
  (set (reg:SI 3) (reg:SI 6))
  ...)

(insn n n2 next
  (set (reg:SI 3)
       (plus:SI (reg:SI 3) (reg:SI 109)))
  ...)
```

It is up to you to make sure that each operand, in each pattern, has constraints that can handle any RTL expression that could be present for that operand. (When multiple alternatives are in use, each pattern must, for each possible combination of operand expressions, have at least one alternative which can handle that combination of operands.) The constraints don't need to *allow* any possible operand--when this is the case, they do not constrain--but they must at least point the way to reloading any possible operand so that it will fit.

- If the constraint accepts whatever operands the predicate permits, there is no problem: reloading is never necessary for this operand.

  For example, an operand whose constraints permit everything except registers is safe provided its predicate rejects registers.

  An operand whose predicate accepts only constant values is safe provided its constraints include the

letter `i'. If any possible constant value is accepted, then nothing less than `i' will do; if the predica more selective, then the constraints may also be more selective.

- Any operand expression can be reloaded by copying it into a register. So if an operand's constraints allow some kind of register, it is certain to be safe. It need not permit all classes of registers; the compiler knows how to copy a register into another register of the proper class in order to make an instruction valid.
- A nonoffsettable memory reference can be reloaded by copying the address into a register. So if the constraint uses the letter `o', all memory references are taken care of.
- A constant operand can be reloaded by allocating space in memory to hold it as preinitialized data. Then the memory reference can be used in place of the constant. So if the constraint uses the letters `o' or `m', constant operands are not a problem.
- If the constraint permits a constant and a pseudo register used in an insn was not allocated to a hard register and is equivalent to a constant, the register will be replaced with the constant. If the predicate does not permit a constant and the insn is re-recognized for some reason, the compiler will crash. Thus the predicate must always recognize any objects allowed by the constraint.

If the operand's predicate can recognize registers, but the constraint does not permit them, it can make the compiler crash. When this operand happens to be a register, the reload pass will be stymied, because it does not know how to copy a register temporarily into memory.

Multiple Alternative Constraints

Sometimes a single instruction has multiple alternative sets of possible operands. For example, on the 68000, a logical-or instruction can combine register or an immediate value into memory, or it can combine any kind of operand into a register; but it cannot combine one memory location into another.

These constraints are represented as multiple alternatives. An alternative can be described by a series of letters for each operand. The overall constraint for an operand is made from the letters for this operand from the first alternative, a comma, the letters for this operand from the second alternative, a comma, and so on until the last alternative. Here is how it is done for fullword logical-or on the 68000:

```
(define_insn "iorsi3"
  [(set (match_operand:SI 0 "general_operand" "=m,d")
        (ior:SI (match_operand:SI 1 "general_operand" "%0,0")
                (match_operand:SI 2 "general_operand" "dKs,dmKs")))]
  ...)
```

The first alternative has `m' (memory) for operand 0, `0' for operand 1 (meaning it must match operand 0), and `dKs' for operand 2. The second alternative has `d' (data register) for operand 0, `0' for operand 1, and `dmKs' for operand 2. The `=' and `%' in the constraints apply to all the alternatives; their meaning is explained in the next section (see section Register Class Preferences).

If all the operands fit any one alternative, the instruction is valid. Otherwise, for each alternative, the compiler counts how many instructions must be added to copy the operands so that that alternative applies. The alternative requiring the least copying is chosen. If two alternatives need the same amount of copying, the one that comes first is chosen. These choices can be altered with the `?' and `!' characters:

?

Disparage slightly the alternative that the `?' appears in, as a choice when no alternative applies exactly. The compiler regards this alternative as one unit more costly for each `?' that appears in it

!

Disparage severely the alternative that the `!' appears in. This alternative can still be used if it fits without reloading, but if reloading is needed, some other alternative will be used.

When an insn pattern has multiple alternatives in its constraints, often the appearance of the assembler code is determined mostly by which alternative was matched. When this is so, the C code for writing the assembler code can use the variable `which_alternative`, which is the ordinal number of the alternative that was actually satisfied (0 for the first, 1 for the second alternative, etc.). See section C Statements for Assembler Output.

Register Class Preferences

The operand constraints have another function: they enable the compiler to decide which kind of hardware register a pseudo register is best allocated to. The compiler examines the constraints that apply to the insns that use the pseudo register, looking for the machine-dependent letters such as `d' and `a' that specify classes of registers. The pseudo register is put in whichever class gets the most "votes". The constraint letters `g' and `r' also vote: they vote in favor of a general register. The machine description says which registers are considered general.

Of course, on some machines all registers are equivalent, and no register classes are defined. Then none of this complexity is relevant.

Constraint Modifier Characters

Here are constraint modifier characters.

`='

Means that this operand is write-only for this instruction: the previous value is discarded and replaced by output data.

`+'

Means that this operand is both read and written by the instruction.

When the compiler fixes up the operands to satisfy the constraints, it needs to know which operands are inputs to the instruction and which are outputs from it. `=' identifies an output; `+' identifies an operand that is both input and output; all other operands are assumed to be input only.

`&'

Means (in a particular alternative) that this operand is written before the instruction is finished using the input operands. Therefore, this operand may not lie in a register that is used as an input operand or as part of any memory address.

`&' applies only to the alternative in which it is written. In constraints with multiple alternatives, sometimes one alternative requires `&' while others do not. See, for example, the `movdf' insn of the 68000.

`&' does not obviate the need to write `='.

`%'

Declares the instruction to be commutative for this operand and the following operand. This means that the compiler may interchange the two operands if that is the cheapest way to make all operands fit the constraints. This is often used in patterns for addition instructions that really have only two operands: the result must go in one of the arguments. Here for example, is how the 68000 halfword-add instruction is defined:

```
(define_insn "addhi3"
  [(set (match_operand:HI 0 "general_operand" "=m,r")
     (plus:HI (match_operand:HI 1 "general_operand" "%0,0")
              (match_operand:HI 2 "general_operand" "di,g")))]
  ...)
```

`#'

Says that all following characters, up to the next comma, are to be ignored as a constraint. They are significant only for choosing register preferences.

`*'

Says that the following character should be ignored when choosing register preferences. `*' has no effect on the meaning of the constraint as a constraint, and no effect on reloading.

Here is an example: the 68000 has an instruction to sign-extend a halfword in a data register, and can also sign-extend a value by copying it into an address register. While either kind of register is acceptable, the constraints on an address-register destination are less strict, so it is best if register allocation makes an address register its goal. Therefore, `*' is used so that the `d' constraint letter (for data register) is ignored when computing register preferences.

```
(define_insn "extendhisi2"
  [(set (match_operand:SI 0 "general_operand" "=*d,a")
        (sign_extend:SI
         (match_operand:HI 1 "general_operand" "0,g")))]
  ...)
```

Constraints for Particular Machines

Whenever possible, you should use the general-purpose constraint letters in `asm` arguments, since they will convey meaning more readily to people reading your code. Failing that, use the constraint letters that usually have very similar meanings across architectures. The most commonly used constraints are `m' and `r' (for memory and general-purpose registers respectively; see section Simple Constraints), and `I', usually the letter indicating the most common immediate-constant format.

For each machine architecture, the `config/machine.h' file defines additional constraints. These constraints are used by the compiler itself for instruction generation, as well as for `asm` statements; therefore, some of the constraints are not particularly interesting for `asm`. The constraints are defined through these macros:

`REG_CLASS_FROM_LETTER`

Register class constraints (usually lower case).

`CONST_OK_FOR_LETTER_P`

Immediate constant constraints, for non-floating point constants of word size or smaller precision (usually upper case).

`CONST_DOUBLE_OK_FOR_LETTER_P`

Immediate constant constraints, for all floating point constants and for constants of greater than word size precision (usually upper case).

`EXTRA_CONSTRAINT`

Special cases of registers or memory. This macro is not required, and is only defined for some machines.

Inspecting these macro definitions in the compiler source for your machine is the best way to be certain you have the right constraints. However, here is a summary of the machine-dependent constraints available on some particular machines.

ARM family---`arm.h'

`f`

Floating-point register

`F`

One of the floating-point constants 0.0, 0.5, 1.0, 2.0, 3.0, 4.0, 5.0 or 10.0

`G`

Floating-point constant that would satisfy the constraint `F' if it were negated

`I`

Integer that is valid as an immediate operand in a data processing instruction. That is, an integer in the range 0 to 255 rotated by a multiple of 2

`J`

Integer in the range -4095 to 4095

`K`

Integer that satisfies constraint `I' when inverted (ones complement)

`L`

Integer that satisfies constraint `I' when negated (twos complement)

`M`

Integer in the range 0 to 32

`Q`

A memory reference where the exact address is in a single register (``m'' is preferable for `asm` statements)

`R`

An item in the constant pool

`S`

A symbol in the text segment of the current file

- AMD 29000 family---`a29k.h'

`l`

Local register 0

`b`

Byte Pointer (`BP') register

`q`

`Q' register

`h`

Special purpose register

`A`

First accumulator register

`a`

Other accumulator register

`f`

Floating point register

`I`

Constant greater than 0, less than 0x100

`J`

Constant greater than 0, less than 0x10000

`K`

Constant whose high 24 bits are on (1)

`L`

16 bit constant whose high 8 bits are on (1)

`M`

32 bit constant whose high 16 bits are on (1)

`N`

32 bit negative constant that fits in 8 bits

`O`

The constant 0x80000000 or, on the 29050, any 32 bit constant whose low 16 bits are 0.

`P`

16 bit negative constant that fits in 8 bits

`G`

`H`

A floating point constant (in `asm` statements, use the machine independent `E' or `F' instead)

- IBM RS6000---`rs6000.h'`

`b`

Address base register

`f`

Floating point register

`h`

`MQ', `CTR', or `LINK' register

`q`
: `MQ' register

`c`
: `CTR' register

`l`
: `LINK' register

`x`
: `CR' register (condition register) number 0

`y`
: `CR' register (condition register)

`I`
: Signed 16 bit constant

`J`
: Constant whose low 16 bits are 0

`K`
: Constant whose high 16 bits are 0

`L`
: Constant suitable as a mask operand

`M`
: Constant larger than 31

`N`
: Exact power of 2

`O`
: Zero

`P`
: Constant whose negation is a signed 16 bit constant

`G`
: Floating point constant that can be loaded into a register with one instruction per word

`Q`
: Memory operand that is an offset from a register (`m' is preferable for `asm` statements)

- Intel 386---`` `i386.h' ``

`q`
: `a', `b`, `c`, or `d` register

`A`
: `a', or `d` register (for 64-bit ints)

`f`
: Floating point register

`t`
: First (top of stack) floating point register

`u`
: Second floating point register

`a`
: `a' register

`b`
: `b' register

`c`
: `c' register

`d`
: `d' register

`D`
: `di' register

`S`
: `si' register

`I`
: Constant in range 0 to 31 (for 32 bit shifts)

`J`
: Constant in range 0 to 63 (for 64 bit shifts)

`K`
: `0xff'

`L`
: `0xffff'

`M`
: 0, 1, 2, or 3 (shifts for `lea` instruction)

`G`
: Standard 80387 floating point constant

- Intel 960---`` `i960.h' ``

`f`
: Floating point register (`fp0` to `fp3`)

`l`
: Local register (`r0` to `r15`)

`b`
: Global register (`g0` to `g15`)

`d`
: Any local or global register

`I`

Integers from 0 to 31

`J`

0

`K`

Integers from -31 to 0

`G`

Floating point 0

`H`

Floating point 1

- MIPS---`mips.h'`

`d`

General-purpose integer register

`f`

Floating-point register (if available)

`h`

`Hi' register

`l`

`Lo' register

`x`

`Hi' or `Lo' register

`y`

General-purpose integer register

`z`

Floating-point status register

`I`

Signed 16 bit constant (for arithmetic instructions)

`J`

Zero

`K`

Zero-extended 16-bit constant (for logic instructions)

`L`

Constant with low 16 bits zero (can be loaded with `lui`)

`M`

32 bit constant which requires two instructions to load (a constant which is not `I', `K', or `L')

`N`

Negative 16 bit constant

`O`

Exact power of two

`P`

Positive 16 bit constant

`G`

Floating point zero

`Q`

Memory reference that can be loaded with more than one instruction (`m' is preferable for `asm` statements)

`R`

Memory reference that can be loaded with one instruction (`m' is preferable for `asm` statements)

`S`

Memory reference in external OSF/rose PIC format (`m' is preferable for `asm` statements)

- Motorola 680x0---`m68k.h'

`a`

Address register

`d`

Data register

`f`

68881 floating-point register, if available

`x`

Sun FPA (floating-point) register, if available

`y`

First 16 Sun FPA registers, if available

`I`

Integer in the range 1 to 8

`J`

16 bit signed number

`K`

Signed number whose magnitude is greater than 0x80

`L`

Integer in the range -8 to -1

`G`

Floating point constant that is not a 68881 constant

`H`

Floating point constant that can be used by Sun FPA

- SPARC---`sparc.h'

`f`

Floating-point register

`I`

Signed 13 bit constant

`J`

Zero

`K`

32 bit constant with the low 12 bits clear (a constant that can be loaded with the `sethi` instruction)

`G`

Floating-point zero

`H`

Signed 13 bit constant, sign-extended to 32 or 64 bits

`Q`

Memory reference that can be loaded with one instruction (`m' is more appropriate for `asm` statements)

`S`

Constant, or memory address

`T`

Memory address aligned to an 8-byte boundary

`U`

Even register

Not Using Constraints

Some machines are so clean that operand constraints are not required. For example, on the Vax, an operand valid in one context is valid in any other context. On such a machine, every operand constraint would be `g', excepting only operands of "load address" instructions which are written as if they referred to a memory location's contents but actual refer to its address. They would have constraint `p'.

For such machines, instead of writing `g' and `p' for all the constraints, you can choose to write a description with empty constraints. Then you write `""' for the constraint in every `match_operand`. Address operands are identified by writing an `address` expression around the `match_operand`, not by their constraints.

When the machine description has just empty constraints, certain parts of compilation are skipped, making the compiler faster. However, few machines actually do not need constraints; all machine descriptions now in existence use constraints.

Standard Pattern Names For Generation

Here is a table of the instruction names that are meaningful in the RTL generation pass of the compiler. Giving one of these names to an instruction pattern tells the RTL generation pass that it can use the pattern in to accomplish a certain task.

`movm'

Here m stands for a two-letter machine mode name, in lower case. This instruction pattern moves data with that machine mode from operand 1 to operand 0. For example, `movsi' moves full-word data.

If operand 0 is a `subreg` with mode m of a register whose own mode is wider than m, the effect of this instruction is to store the specified value in the part of the register that corresponds to mode m. The effect on the rest of the register is undefined.

This class of patterns is special in several ways. First of all, each of these names *must* be defined, because there is no other way to copy a datum from one place to another.

Second, these patterns are not used solely in the RTL generation pass. Even the reload pass can generate move insns to copy values from stack slots into temporary registers. When it does so, one of the operands is a hard register and the other is an operand that can need to be reloaded into a register.

Therefore, when given such a pair of operands, the pattern must generate RTL which needs no reloading and needs no temporary registers--no registers other than the operands. For example, if you support the pattern with a `define_expand`, then in such a case the `define_expand` mustn't call `force_reg` or any other such function which might generate new pseudo registers.

This requirement exists even for subword modes on a RISC machine where fetching those modes from memory normally requires several insns and some temporary registers. Look in `` `spur.md' `` to see how the requirement can be satisfied.

During reload a memory reference with an invalid address may be passed as an operand. Such an address will be replaced with a valid address later in the reload pass. In this case, nothing may be done with the address except to use it as it stands. If it is copied, it will not be replaced with a valid address. No attempt should be made to make such an address into a valid address and no routine (such as `change_address`) that will do so may be called. Note that `general_operand` will fail when applied to such an address.

The global variable `reload_in_progress` (which must be explicitly declared if required) can be used to determine whether such special handling is required.

The variety of operands that have reloads depends on the rest of the machine description, but typically on a RISC machine these can only be pseudo registers that did not get hard registers, while on other machines explicit memory references will get optional reloads.

If a scratch register is required to move an object to or from memory, it can be allocated using `gen_reg_rtx` prior to reload. But this is impossible during and after reload. If there are cases needing scratch registers after reload, you must define `SECONDARY_INPUT_RELOAD_CLASS` and perhaps also `SECONDARY_OUTPUT_RELOAD_CLASS` to detect them, and provide patterns `reload_inm' or `reload_outm' to handle them. See section Register Classes.

The constraints on a `movem' must permit moving any hard register to any other hard register provided that `HARD_REGNO_MODE_OK` permits mode m in both registers and `REGISTER_MOVE_COST` applied to their classes returns a value of 2.

It is obligatory to support floating point `movem' instructions into and out of any registers that can hold fixed point values, because unions and structures (which have modes `SImode` or `DImode`) can be in those registers and they may have floating point members.

There may also be a need to support fixed point `movem' instructions in and out of floating point registers. Unfortunately, I have forgotten why this was so, and I don't know whether it is still true. If `HARD_REGNO_MODE_OK` rejects fixed point values in floating point registers, then the constraints of

the fixed point `movem' instructions must be designed to avoid ever trying to reload into a floating point register.

`reload_inm'

`reload_outm'

Like `movm', but used when a scratch register is required to move between operand 0 and operand 1. Operand 2 describes the scratch register. See the discussion of the `SECONDARY_RELOAD_CLASS` macro in see section Register Classes.

`movstrictm'

Like `movm' except that if operand 0 is a `subreg` with mode m of a register whose natural mode is wider, the `movstrictm' instruction is guaranteed not to alter any of the register except the part which belongs to mode m.

`load_multiple'

Load several consecutive memory locations into consecutive registers. Operand 0 is the first of the consecutive registers, operand 1 is the first memory location, and operand 2 is a constant: the number of consecutive registers.

Define this only if the target machine really has such an instruction; do not define this if the most efficient way of loading consecutive registers from memory is to do them one at a time.

On some machines, there are restrictions as to which consecutive registers can be stored into memory, such as particular starting or ending register numbers or only a range of valid counts. For those machines, use a `define_expand` (see section Defining RTL Sequences for Code Generation) and make the pattern fail if the restrictions are not met.

Write the generated insn as a `parallel` with elements being a `set` of one register from the appropriate memory location (you may also need `use` or `clobber` elements). Use a `match_parallel` (see section RTL Template) to recognize the insn. See `a29k.md' and `rs6000.md' for examples of the use of this insn pattern.

`store_multiple'

Similar to `load_multiple', but store several consecutive registers into consecutive memory locations. Operand 0 is the first of the consecutive memory locations, operand 1 is the first register, and operand 2 is a constant: the number of consecutive registers.

`addm3'

Add operand 2 and operand 1, storing the result in operand 0. All operands must have mode m. This can be used even on two-address machines, by means of constraints requiring operands 1 and 0 to be the same location.

`subm3', `mulm3'

`divm3', `udivm3', `modm3', `umodm3'

`sminm3', `smaxm3', `uminm3', `umaxm3'

`andm3', `iorm3', `xorm3'

Similar, for other arithmetic operations.

`mulhisi3'

Multiply operands 1 and 2, which have mode `HImode`, and store a `SImode` product in operand 0.

`mulqihi3', `mulsidi3'

Similar widening-multiplication instructions of other widths.

`umulqihi3', `umulhisi3', `umulsidi3'

Similar widening-multiplication instructions that do unsigned multiplication.

`mulm3_highpart'

Perform a signed multiplication of operands 1 and 2, which have mode m, and store the most significant half of the product in operand 0. The least significant half of the product is discarded.

`umulm3_highpart'

Similar, but the multiplication is unsigned.

`divmodm4'

Signed division that produces both a quotient and a remainder. Operand 1 is divided by operand 2 to produce a quotient stored in operand 0 and a remainder stored in operand 3.

For machines with an instruction that produces both a quotient and a remainder, provide a pattern for `divmodm4' but do not provide patterns for `divm3' and `modm3'. This allows optimization in the relatively common case when both the quotient and remainder are computed.

If an instruction that just produces a quotient or just a remainder exists and is more efficient than the instruction that produces both, write the output routine of `divmodm4' to call `find_reg_note` and look for a `REG_UNUSED` note on the quotient or remainder and generate the appropriate instruction.

`udivmodm4'

Similar, but does unsigned division.

`ashlm3'

Arithmetic-shift operand 1 left by a number of bits specified by operand 2, and store the result in operand 0. Here m is the mode of operand 0 and operand 1; operand 2's mode is specified by the instruction pattern, and the compiler will convert the operand to that mode before generating the instruction.

`ashrm3', `lshrm3', `rotlm3', `rotrm3'

Other shift and rotate instructions, analogous to the `ashlm3` instructions.

`negm2'

Negate operand 1 and store the result in operand 0.

`absm2'

Store the absolute value of operand 1 into operand 0.

`sqrtm2'

Store the square root of operand 1 into operand 0.

The `sqrt` built-in function of C always uses the mode which corresponds to the C data type `double`.

`ffsm2'

Store into operand 0 one plus the index of the least significant 1-bit of operand 1. If operand 1 is zero, store zero. m is the mode of operand 0; operand 1's mode is specified by the instruction pattern, and the compiler will convert the operand to that mode before generating the instruction.

The `ffs` built-in function of C always uses the mode which corresponds to the C data type `int`.

`one_cmplm2'

Store the bitwise-complement of operand 1 into operand 0.

`cmpm'

Compare operand 0 and operand 1, and set the condition codes. The RTL pattern should look like this:

```
(set (cc0) (compare (match_operand:m 0 ...)
                    (match_operand:m 1 ...)))
```

`tstm'

Compare operand 0 against zero, and set the condition codes. The RTL pattern should look like this:

```
(set (cc0) (match_operand:m 0 ...))
```

`tstm' patterns should not be defined for machines that do not use `(cc0)`. Doing so would confuse the optimizer since it would no longer be clear which `set` operations were comparisons. The `cmpm' patterns should be used instead.

`movstrm'

Block move instruction. The addresses of the destination and source strings are the first two operands, and both are in mode `Pmode`. The number of bytes to move is the third operand, in mode m.

The fourth operand is the known shared alignment of the source and destination, in the form of a `const_int` rtx. Thus, if the compiler knows that both source and destination are word-aligned, it may provide the value 4 for this operand.

These patterns need not give special consideration to the possibility that the source and destination strings might overlap.

`cmpstrm'

Block compare instruction, with five operands. Operand 0 is the output; it has mode m. The remaining four operands are like the operands of `movstrm'. The two memory blocks specified are compared byte by byte in lexicographic order. The effect of the instruction is to store a value in operand 0 whose sign indicates the result of the comparison.

Compute the length of a string, with three operands. Operand 0 is the result (of mode m), operand 1 is a `mem` referring to the first character of the string, operand 2 is the character to search for (normally zero), and operand 3 is a constant describing the known alignment of the beginning of the string.

`floatmn2'

Convert signed integer operand 1 (valid for fixed point mode m) to floating point mode n and store in operand 0 (which has mode n).

`floatunsmn2'

Convert unsigned integer operand 1 (valid for fixed point mode m) to floating point mode n and store in operand 0 (which has mode n).

`fixmn2'

Convert operand 1 (valid for floating point mode m) to fixed point mode n as a signed number and store in operand 0 (which has mode n). This instruction's result is defined only when the value of operand 1 is an integer.

`fixunsmn2'

Convert operand 1 (valid for floating point mode m) to fixed point mode n as an unsigned number and store in operand 0 (which has mode n). This instruction's result is defined only when the value of operand 1 is an integer.

`ftruncm2'

Convert operand 1 (valid for floating point mode m) to an integer value, still represented in floating point mode m, and store it in operand 0 (valid for floating point mode m).

`fix_truncmn2'

Like `fixmn2' but works for any floating point value of mode m by converting the value to an integer.

`fixuns_truncmn2'

Like `fixunsmn2' but works for any floating point value of mode m by converting the value to an integer.

`truncmn'

Truncate operand 1 (valid for mode m) to mode n and store in operand 0 (which has mode n). Both modes must be fixed point or both floating point.

`extendmn'

Sign-extend operand 1 (valid for mode m) to mode n and store in operand 0 (which has mode n). Both modes must be fixed point or both floating point.

`zero_extendmn'

Zero-extend operand 1 (valid for mode m) to mode n and store in operand 0 (which has mode n). Both modes must be fixed point.

`extv'

Extract a bit field from operand 1 (a register or memory operand), where operand 2 specifies the width in bits and operand 3 the starting bit, and store it in operand 0. Operand 0 must have mode `word_mode`. Operand 1 may have mode `byte_mode` or `word_mode`; often `word_mode` is allowed only for registers. Operands 2 and 3 must be valid for `word_mode`.

The RTL generation pass generates this instruction only with constants for operands 2 and 3.

The bit-field value is sign-extended to a full word integer before it is stored in operand 0.

`extzv'

Like `extv' except that the bit-field value is zero-extended.

`insv'

Store operand 3 (which must be valid for `word_mode`) into a bit field in operand 0, where operand 1 specifies the width in bits and operand 2 the starting bit. Operand 0 may have mode `byte_mode` or `word_mode`; often `word_mode` is allowed only for registers. Operands 1 and 2 must be valid for `word_mode`.

The RTL generation pass generates this instruction only with constants for operands 1 and 2.

`movmodecc'

Conditionally move operand 2 or operand 3 into operand 0 according to the comparison in operand 1. If the comparison is true, operand 2 is moved into operand 0, otherwise operand 3 is moved.

The mode of the operands being compared need not be the same as the operands being moved. So machines, sparc64 for example, have instructions that conditionally move an integer value based the floating point condition codes and vice versa.

If the machine does not have conditional move instructions, do not define these patterns.

`scond'

Store zero or nonzero in the operand according to the condition codes. Value stored is nonzero iff the condition cond is true. cond is the name of a comparison operation expression code, such as `eq`, `lt` or `leu`.

You specify the mode that the operand must have when you write the `match_operand` expression. The compiler automatically sees which mode you have used and supplies an operand of that mode.

The value stored for a true condition must have 1 as its low bit, or else must be negative. Otherwise the instruction is not suitable and you should omit it from the machine description. You describe to the compiler exactly which value is stored by defining the macro `STORE_FLAG_VALUE` (see section Miscellaneous Parameters). If a description cannot be found that can be used for all the `scond' patterns, you should omit those operations from the machine description.

These operations may fail, but should do so only in relatively uncommon cases; if they would fail for common cases involving integer comparisons, it is best to omit these patterns.

If these operations are omitted, the compiler will usually generate code that copies the constant one to the target and branches around an assignment of zero to the target. If this code is more efficient than the potential instructions used for the `scond' pattern followed by those required to convert the result into a 1 or a zero in `SImode`, you should omit the `scond' operations from the machine description.

`bcond'

Conditional branch instruction. Operand 0 is a `label_ref` that refers to the label to jump to. Jump if the condition codes meet condition cond.

Some machines do not follow the model assumed here where a comparison instruction is followed by a conditional branch instruction. In that case, the `cmpm' (and `tstm') patterns should simply store the operands away and generate all the required insns in a `define_expand` (see section Defining RTL Sequences for Code Generation) for the conditional branch operations. All calls to expand `bcond' patterns are immediately preceded by calls to expand either a `cmpm' pattern or a `tstm' pattern.

Machines that use a pseudo register for the condition code value, or where the mode used for the comparison depends on the condition being tested, should also use the above mechanism. See section Defining Jump Instruction Patterns

The above discussion also applies to the `movmodecc' and `scond' patterns.

`call'

Subroutine call instruction returning no value. Operand 0 is the function to call; operand 1 is the number of bytes of arguments pushed (in mode `SImode`, except it is normally a `const_int`); operand 2 is the number of registers used as operands.

On most machines, operand 2 is not actually stored into the RTL pattern. It is supplied for the sake of some RISC machines which need to put this information into the assembler code; they can put it in the

RTL instead of operand 1.

Operand 0 should be a `mem` RTX whose address is the address of the function. Note, however, that this address can be a `symbol_ref` expression even if it would not be a legitimate memory address on the target machine. If it is also not a valid argument for a call instruction, the pattern for this operation should be a `define_expand` (see section Defining RTL Sequences for Code Generation) that places the address into a register and uses that register in the call instruction.

`call_value'

Subroutine call instruction returning a value. Operand 0 is the hard register in which the value is returned. There are three more operands, the same as the three operands of the `call' instruction (but with numbers increased by one).

Subroutines that return `BLKmode` objects use the `call' insn.

`call_pop', `call_value_pop'

Similar to `call' and `call_value', except used if defined and if `RETURN_POPS_ARGS` is non-zero. They should emit a `parallel` that contains both the function call and a `set` to indicate the adjustment made to the frame pointer.

For machines where `RETURN_POPS_ARGS` can be non-zero, the use of these patterns increases the number of functions for which the frame pointer can be eliminated, if desired.

`untyped_call'

Subroutine call instruction returning a value of any type. Operand 0 is the function to call; operand 1 is a memory location where the result of calling the function is to be stored; operand 2 is a `parallel` expression where each element is a `set` expression that indicates the saving of a function return value into the result block.

This instruction pattern should be defined to support `__builtin_apply` on machines where special instructions are needed to call a subroutine with arbitrary arguments or to save the value returned. This instruction pattern is required on machines that have multiple registers that can hold a return value (i.e. `FUNCTION_VALUE_REGNO_P` is true for more than one register).

`return'

Subroutine return instruction. This instruction pattern name should be defined only if a single instruction can do all the work of returning from a function.

Like the `movm' patterns, this pattern is also used after the RTL generation phase. In this case it is to support machines where multiple instructions are usually needed to return from a function, but some class of functions only requires one instruction to implement a return. Normally, the applicable functions are those which do not need to save any registers or allocate stack space.

For such machines, the condition specified in this pattern should only be true when `reload_completed` is non-zero and the function's epilogue would only be a single instruction. For machines with register windows, the routine `leaf_function_p` may be used to determine if a register window push is required.

Machines that have conditional return instructions should define patterns such as

```
(define_insn ""
```


```
[(set (pc)
      (if_then_else (match_operator
                        0 "comparison_operator"
                        [(cc0) (const_int 0)])
                    (return)
                    (pc)))]
"condition"
"...")
```

where condition would normally be the same condition specified on the named `return' pattern.

`untyped_return'

Untyped subroutine return instruction. This instruction pattern should be defined to support `__builtin_return` on machines where special instructions are needed to return a value of any type.

Operand 0 is a memory location where the result of calling a function with `__builtin_apply` is stored; operand 1 is a `parallel` expression where each element is a `set` expression that indicates the restoring of a function return value from the result block.

`nop'

No-op instruction. This instruction pattern name should always be defined to output a no-op in assembler code. `(const_int 0)` will do as an RTL pattern.

`indirect_jump'

An instruction to jump to an address which is operand zero. This pattern name is mandatory on all machines.

`casesi'

Instruction to jump through a dispatch table, including bounds checking. This instruction takes five operands:

The index to dispatch on, which has mode `SImode`.

The lower bound for indices in the table, an integer constant.

The total range of indices in the table--the largest index minus the smallest one (both inclusive).

A label that precedes the table itself.

A label to jump to if the index has a value outside the bounds. (If the machine-description macro `CASE_DROPS_THROUGH` is defined, then an out-of-bounds index drops through to the code following the jump table instead of jumping to this label. In that case, this label is not actually used by the `casesi' instruction, but it is always provided as an operand.)

The table is a `addr_vec` or `addr_diff_vec` inside of a `jump_insn`. The number of elements in the table is one plus the difference between the upper bound and the lower bound.

`tablejump'

Instruction to jump to a variable address. This is a low-level capability which can be used to implement a dispatch table when there is no `casesi' pattern.

This pattern requires two operands: the address or offset, and a label which should immediately precede the jump table. If the macro `CASE_VECTOR_PC_RELATIVE` is defined then the first

operand is an offset which counts from the address of the table; otherwise, it is an absolute address to jump to. In either case, the first operand has mode `Pmode`.

The `tablejump' insn is always the last insn before the jump table it uses. Its assembler code normally has no need to use the second operand, but you should incorporate it in the RTL pattern so that the jump optimizer will not delete the table as unreachable code.

`save_stack_block'

`save_stack_function'

`save_stack_nonlocal'

`restore_stack_block'

`restore_stack_function'

`restore_stack_nonlocal'

Most machines save and restore the stack pointer by copying it to or from an object of mode `Pmode`. Do not define these patterns on such machines.

Some machines require special handling for stack pointer saves and restores. On those machines, define the patterns corresponding to the non-standard cases by using a `define_expand` (see section Defining RTL Sequences for Code Generation) that produces the required insns. The three types of saves and restores are:

`save_stack_block' saves the stack pointer at the start of a block that allocates a variable-sized object, and `restore_stack_block' restores the stack pointer when the block is exited.

`save_stack_function' and `restore_stack_function' do a similar job for the outermost block of a function and are used when the function allocates variable-sized objects or calls `alloca`. Only the epilogue uses the restored stack pointer, allowing a simpler save or restore sequence on some machines.

`save_stack_nonlocal' is used in functions that contain labels branched to by nested functions. It saves the stack pointer in such a way that the inner function can use `restore_stack_nonlocal' to restore the stack pointer. The compiler generates code to restore the frame and argument pointer registers, but some machines require saving and restoring additional data such as register window information or stack backchains. Place insns in these patterns to save and restore any such required data.

When saving the stack pointer, operand 0 is the save area and operand 1 is the stack pointer. The mode used to allocate the save area is the mode of operand 0. You must specify an integral mode, or `VOIDmode` if no save area is needed for a particular type of save (either because no save is needed or because a machine-specific save area can be used). Operand 0 is the stack pointer and operand 1 is the save area for restore operations. If `save_stack_block' is defined, operand 0 must not be `VOIDmode` since these saves can be arbitrarily nested.

A save area is a `mem` that is at a constant offset from `virtual_stack_vars_rtx` when the stack pointer is saved for use by nonlocal gotos and a `reg` in the other two cases.

`allocate_stack'

Subtract (or add if `STACK_GROWS_DOWNWARD` is undefined) operand 0 from the stack pointer to create space for dynamically allocated data.

Do not define this pattern if all that must be done is the subtraction. Some machines require other operations such as stack probes or maintaining the back chain. Define this pattern to emit those operations in addition to updating the stack pointer.

When the Order of Patterns Matters

Sometimes an insn can match more than one instruction pattern. Then the pattern that appears first in the machine description is the one used. Therefore, more specific patterns (patterns that will match fewer things) and faster instructions (those that will produce better code when they do match) should usually go first in the description.

In some cases the effect of ordering the patterns can be used to hide a pattern when it is not valid. For example, the 68000 has an instruction for converting a fullword to floating point and another for converting a byte to floating point. An instruction converting an integer to floating point could match either one. We put the pattern to convert the fullword first to make sure that one will be used rather than the other. (Otherwise a large integer might be generated as a single-byte immediate quantity, which would not work.) Instead of using this pattern ordering it would be possible to make the pattern for convert-a-byte smart enough to deal properly with any constant value.

Interdependence of Patterns

Every machine description must have a named pattern for each of the conditional branch names `bcond'. The recognition template must always have the form

```
(set (pc)
     (if_then_else (cond (cc0) (const_int 0))
                   (label_ref (match_operand 0 "" ""))
                   (pc)))
```

In addition, every machine description must have an anonymous pattern for each of the possible reverse-conditional branches. Their templates look like

```
(set (pc)
     (if_then_else (cond (cc0) (const_int 0))
                   (pc)
                   (label_ref (match_operand 0 "" ""))))
```

They are necessary because jump optimization can turn direct-conditional branches into reverse-conditional branches.

It is often convenient to use the `match_operator` construct to reduce the number of patterns that must be specified for branches. For example,

```
(define_insn ""
  [(set (pc)
        (if_then_else (match_operator 0 "comparison_operator"
```


```
                                    [(cc0) (const_int 0)])
                      (pc)
                      (label_ref (match_operand 1 "" ""))))]
  "condition"
  "...")
```

In some cases machines support instructions identical except for the machine mode of one or more operands. For example, there may be "sign-extend halfword" and "sign-extend byte" instructions whose patterns are

```
(set (match_operand:SI 0 ...)
     (extend:SI (match_operand:HI 1 ...)))

(set (match_operand:SI 0 ...)
     (extend:SI (match_operand:QI 1 ...)))
```

Constant integers do not specify a machine mode, so an instruction to extend a constant value could match either pattern. The pattern it actually will match is the one that appears first in the file. For correct results, this must be the one for the widest possible mode (`HImode`, here). If the pattern matches the `QImode` instruction, the results will be incorrect if the constant value does not actually fit that mode.

Such instructions to extend constants are rarely generated because they are optimized away, but they do occasionally happen in nonoptimized compilations.

If a constraint in a pattern allows a constant, the reload pass may replace a register with a constant permitted by the constraint in some cases. Similarly for memory references. Because of this substitution, you should not provide separate patterns for increment and decrement instructions. Instead, they should be generated from the same pattern that supports register-register add insns by examining the operands and generating the appropriate machine instruction.

Defining Jump Instruction Patterns

For most machines, GNU CC assumes that the machine has a condition code. A comparison insn sets the condition code, recording the results of both signed and unsigned comparison of the given operands. A separate branch insn tests the condition code and branches or not according its value. The branch insns come in distinct signed and unsigned flavors. Many common machines, such as the Vax, the 68000 and the 32000, work this way.

Some machines have distinct signed and unsigned compare instructions, and only one set of conditional branch instructions. The easiest way to handle these machines is to treat them just like the others until the final stage where assembly code is written. At this time, when outputting code for the compare instruction, peek ahead at the following branch using `next_cc0_user (insn)`. (The variable `insn` refers to the insn being output, in the output-writing code in an instruction pattern.) If the RTL says that is an unsigned branch, output an unsigned compare; otherwise output a signed compare. When the branch itself is output, you can treat signed and unsigned branches identically.

The reason you can do this is that GNU CC always generates a pair of consecutive RTL insns, possibly separated by `note` insns, one to set the condition code and one to test it, and keeps the pair inviolate until the end.

To go with this technique, you must define the machine-description macro `NOTICE_UPDATE_CC` to do `CC_STATUS_INIT`; in other words, no compare instruction is superfluous.

Some machines have compare-and-branch instructions and no condition code. A similar technique works for them. When it is time to "output" a compare instruction, record its operands in two static variables. When outputting the branch-on-condition-code instruction that follows, actually output a compare-and-branch instruction that uses the remembered operands.

It also works to define patterns for compare-and-branch instructions. In optimizing compilation, the pair of compare and branch instructions will be combined according to these patterns. But this does not happen if optimization is not requested. So you must use one of the solutions above in addition to any special patterns you define.

In many RISC machines, most instructions do not affect the condition code and there may not even be a separate condition code register. On these machines, the restriction that the definition and use of the condition code be adjacent insns is not necessary and can prevent important optimizations. For example, on the IBM RS/6000, there is a delay for taken branches unless the condition code register is set three instructions earlier than the conditional branch. The instruction scheduler cannot perform this optimization if it is not permitted to separate the definition and use of the condition code register.

On these machines, do not use `(cc0)`, but instead use a register to represent the condition code. If there is a specific condition code register in the machine, use a hard register. If the condition code or comparison result can be placed in any general register, or if there are multiple condition registers, use a pseudo register.

On some machines, the type of branch instruction generated may depend on the way the condition code was produced; for example, on the 68k and Sparc, setting the condition code directly from an add or subtract instruction does not clear the overflow bit the way that a test instruction does, so a different branch instruction must be used for some conditional branches. For machines that use `(cc0)`, the set and use of the condition code must be adjacent (separated only by `note` insns) allowing flags in `cc_status` to be used. (See section Condition Code Status.) Also, the comparison and branch insns can be located from each other by using the functions `prev_cc0_setter` and `next_cc0_user`.

However, this is not true on machines that do not use `(cc0)`. On those machines, no assumptions can be made about the adjacency of the compare and branch insns and the above methods cannot be used. Instead, we use the machine mode of the condition code register to record different formats of the condition code register.

Registers used to store the condition code value should have a mode that is in class `MODE_CC`. Normally, it will be `CCmode`. If additional modes are required (as for the add example mentioned above in the Sparc), define the macro `EXTRA_CC_MODES` to list the additional modes required (see section Condition Code Status). Also define `EXTRA_CC_NAMES` to list the names of those modes and `SELECT_CC_MODE` to choose a mode given an operand of a compare.

If it is known during RTL generation that a different mode will be required (for example, if the machine has separate compare instructions for signed and unsigned quantities, like most IBM processors), they can be specified at that time.

If the cases that require different modes would be made by instruction combination, the macro `SELECT_CC_MODE` determines which machine mode should be used for the comparison result. The patterns should be written using that mode. To support the case of the add on the Sparc discussed above, we have the

pattern

```
(define_insn ""
  [(set (reg:CC_NOOV 0)
        (compare:CC_NOOV
          (plus:SI (match_operand:SI 0 "register_operand" "%r")
                   (match_operand:SI 1 "arith_operand" "rI"))
          (const_int 0)))]
  ""
  "...")
```

The `SELECT_CC_MODE` macro on the Sparc returns `CC_NOOVmode` for comparisons whose argument is a `plus`.

Canonicalization of Instructions

There are often cases where multiple RTL expressions could represent an operation performed by a single machine instruction. This situation is most commonly encountered with logical, branch, and multiply-accumulate instructions. In such cases, the compiler attempts to convert these multiple RTL expressions into a single canonical form to reduce the number of insn patterns required.

In addition to algebraic simplifications, following canonicalizations are performed:

- For commutative and comparison operators, a constant is always made the second operand. If a machine only supports a constant as the second operand, only patterns that match a constant in the second operand need be supplied.

  For these operators, if only one operand is a `neg`, `not`, `mult`, `plus`, or `minus` expression, it will be the first operand.
- For the `compare` operator, a constant is always the second operand on machines where `cc0` is used (see section Defining Jump Instruction Patterns). On other machines, there are rare cases where the compiler might want to construct a `compare` with a constant as the first operand. However, these cases are not common enough for it to be worthwhile to provide a pattern matching a constant as the first operand unless the machine actually has such an instruction.

  An operand of `neg`, `not`, `mult`, `plus`, or `minus` is made the first operand under the same conditions as above.
- `(minus x (const_int n))` is converted to `(plus x (const_int -n))`.
- Within address computations (i.e., inside `mem`), a left shift is converted into the appropriate multiplication by a power of two.

  De`Morgan's Law is used to move bitwise negation inside a bitwise logical-and or logical-or operation. If this results in only one operand being a `not` expression, it will be the first one.

  A machine that has an instruction that performs a bitwise logical-and of one operand with the bitwise negation of the other should specify the pattern for that instruction as

  ```
  (define_insn ""
  ```


```
  [(set (match_operand:m 0 ...)
        (and:m (not:m (match_operand:m 1 ...))
               (match_operand:m 2 ...)))]
  "..."
  "...")
```

Similarly, a pattern for a "NAND" instruction should be written

```
(define_insn ""
  [(set (match_operand:m 0 ...)
        (ior:m (not:m (match_operand:m 1 ...))
               (not:m (match_operand:m 2 ...))))]
  "..."
  "...")
```

In both cases, it is not necessary to include patterns for the many logically equivalent RTL expressions.

- The only possible RTL expressions involving both bitwise exclusive-or and bitwise negation are `(xor:m x y)` and `(not:m (xor:m x y))`.
- The sum of three items, one of which is a constant, will only appear in the form

```
(plus:m (plus:m x y) constant)
```

- On machines that do not use `cc0`, `(compare x (const_int 0))` will be converted to x.
- Equality comparisons of a group of bits (usually a single bit) with zero will be written using `zero_extract` rather than the equivalent `and` or `sign_extract` operations.

Machine-Specific Peephole Optimizers

In addition to instruction patterns the `md' file may contain definitions of machine-specific peephole optimizations.

The combiner does not notice certain peephole optimizations when the data flow in the program does not suggest that it should try them. For example, sometimes two consecutive insns related in purpose can be combined even though the second one does not appear to use a register computed in the first one. A machine-specific peephole optimizer can detect such opportunities.

A definition looks like this:

```
(define_peephole
  [insn-pattern-1
   insn-pattern-2
   ...]
  "condition"
  "template"
  "optional insn-attributes")
```

The last string operand may be omitted if you are not using any machine-specific information in this ma description. If present, it must obey the same rules as in a `define_insn`.

In this skeleton, insn-pattern-1 and so on are patterns to match consecutive insns. The optimization applies to a sequence of insns when insn-pattern-1 matches the first one, insn-pattern-2 matches the next, and so on.

Each of the insns matched by a peephole must also match a `define_insn`. Peepholes are checked only at the last stage just before code generation, and only optionally. Therefore, any insn which would match a peephole but no `define_insn` will cause a crash in code generation in an unoptimized compilation, or at various optimization stages.

The operands of the insns are matched with `match_operands`, `match_operator`, and `match_dup`, as usual. What is not usual is that the operand numbers apply to all the insn patterns in the definition. So, you can check for identical operands in two insns by using `match_operand` in one insn and `match_dup` in the other.

The operand constraints used in `match_operand` patterns do not have any direct effect on the applicability of the peephole, but they will be validated afterward, so make sure your constraints are general enough to apply whenever the peephole matches. If the peephole matches but the constraints are not satisfied, the compiler will crash.

It is safe to omit constraints in all the operands of the peephole; or you can write constraints which serve as a double-check on the criteria previously tested.

Once a sequence of insns matches the patterns, the condition is checked. This is a C expression which makes the final decision whether to perform the optimization (we do so if the expression is nonzero). If condition is omitted (in other words, the string is empty) then the optimization is applied to every sequence of insns that matches the patterns.

The defined peephole optimizations are applied after register allocation is complete. Therefore, the peephole definition can check which operands have ended up in which kinds of registers, just by looking at the operands.

The way to refer to the operands in condition is to write `operands[i]` for operand number i (as matched by `(match_operand i ...)`). Use the variable `insn` to refer to the last of the insns being matched; use `prev_nonnote_insn` to find the preceding insns.

When optimizing computations with intermediate results, you can use condition to match only when the intermediate results are not used elsewhere. Use the C expression `dead_or_set_p (insn, op)`, where insn is the insn in which you expect the value to be used for the last time (from the value of `insn`, together with use of `prev_nonnote_insn`), and op is the intermediate value (from `operands[i]`).

Applying the optimization means replacing the sequence of insns with one new insn. The template controls ultimate output of assembler code for this combined insn. It works exactly like the template of a `define_insn`. Operand numbers in this template are the same ones used in matching the original sequence of insns.

The result of a defined peephole optimizer does not need to match any of the insn patterns in the machine description; it does not even have an opportunity to match them. The peephole optimizer definition itself serves as the insn pattern to control how the insn is output.

Defined peephole optimizers are run as assembler code is being output, so the insns they produce are never

combined or rearranged in any way.

Here is an example, taken from the 68000 machine description:

```
(define_peephole
  [(set (reg:SI 15) (plus:SI (reg:SI 15) (const_int 4)))
   (set (match_operand:DF 0 "register_operand" "=f")
        (match_operand:DF 1 "register_operand" "ad"))]
  "FP_REG_P (operands[0]) && ! FP_REG_P (operands[1])"
  "*
{
  rtx xoperands[2];
  xoperands[1] = gen_rtx (REG, SImode, REGNO (operands[1]) + 1);
#ifdef MOTOROLA
  output_asm_insn (\"move.l %1,(sp)\", xoperands);
  output_asm_insn (\"move.l %1,-(sp)\", operands);
  return \"fmove.d (sp)+,%0\";
#else
  output_asm_insn (\"movel %1,sp@\", xoperands);
  output_asm_insn (\"movel %1,sp@-\", operands);
  return \"fmoved sp@+,%0\";
#endif
}
")
```

The effect of this optimization is to change

```
jbsr _foobar
addql #4,sp
movel d1,sp@-
movel d0,sp@-
fmoved sp@+,fp0
```

into

```
jbsr _foobar
movel d1,sp@
movel d0,sp@-
fmoved sp@+,fp0
```

insn-pattern-1 and so on look *almost* like the second operand of `define_insn`. There is one important difference: the second operand of `define_insn` consists of one or more RTX's enclosed in square brackets. Usually, there is only one: then the same action can be written as an element of a `define_peephole`. But when there are multiple actions in a `define_insn`, they are implicitly enclosed in a `parallel`. Then you must explicitly write the `parallel`, and the square brackets within it, in the `define_peephole`. Thus, if an insn pattern looks like this,

```
(define_insn "divmodsi4"
```


```
  [(set (match_operand:SI 0 "general_operand" "=d")
        (div:SI (match_operand:SI 1 "general_operand" "0")
                (match_operand:SI 2 "general_operand" "dmsK")))
   (set (match_operand:SI 3 "general_operand" "=d")
        (mod:SI (match_dup 1) (match_dup 2)))]
  "TARGET_68020"
  "divsl%.l %2,%3:%0")
```

then the way to mention this insn in a peephole is as follows:

```
(define_peephole
  [...
   (parallel
    [(set (match_operand:SI 0 "general_operand" "=d")
          (div:SI (match_operand:SI 1 "general_operand" "0")
                  (match_operand:SI 2 "general_operand" "dmsK")))
     (set (match_operand:SI 3 "general_operand" "=d")
          (mod:SI (match_dup 1) (match_dup 2)))])
   ...]
  ...)
```

Defining RTL Sequences for Code Generation

On some target machines, some standard pattern names for RTL generation cannot be handled with single insn, but a sequence of RTL insns can represent them. For these target machines, you can write a `define_expand` to specify how to generate the sequence of RTL.

A `define_expand` is an RTL expression that looks almost like a `define_insn`; but, unlike the latter, a `define_expand` is used only for RTL generation and it can produce more than one RTL insn.

A `define_expand` RTX has four operands:

- The name. Each `define_expand` must have a name, since the only use for it is to refer to it by name.
- The RTL template. This is just like the RTL template for a `define_peephole` in that it is a vector of RTL expressions each being one insn.
- The condition, a string containing a C expression. This expression is used to express how the availability of this pattern depends on subclasses of target machine, selected by command-line options when GNU CC is run. This is just like the condition of a `define_insn` that has a standard name. Therefore, the condition (if present) may not depend on the data in the insn being matched, but only the target-machine-type flags. The compiler needs to test these conditions during initialization in order to learn exactly which named instructions are available in a particular run.
- The preparation statements, a string containing zero or more C statements which are to be executed before RTL code is generated from the RTL template.

  Usually these statements prepare temporary registers for use as internal operands in the RTL template, but they can also generate RTL insns directly by calling routines such as `emit_insn`, etc. Any such insns precede the ones that come from the RTL template.

Every RTL insn emitted by a `define_expand` must match some `define_insn` in the machine description. Otherwise, the compiler will crash when trying to generate code for the insn or trying to optimize it.

The RTL template, in addition to controlling generation of RTL insns, also describes the operands that need to be specified when this pattern is used. In particular, it gives a predicate for each operand.

A true operand, which needs to be specified in order to generate RTL from the pattern, should be described with a `match_operand` in its first occurrence in the RTL template. This enters information on the operand's predicate into the tables that record such things. GNU CC uses the information to preload the operand into a register if that is required for valid RTL code. If the operand is referred to more than once, subsequent references should use `match_dup`.

The RTL template may also refer to internal "operands" which are temporary registers or labels used only within the sequence made by the `define_expand`. Internal operands are substituted into the RTL template with `match_dup`, never with `match_operand`. The values of the internal operands are not passed in as arguments by the compiler when it requests use of this pattern. Instead, they are computed within the pattern, in the preparation statements. These statements compute the values and store them into the appropriate elements of `operands` so that `match_dup` can find them.

There are two special macros defined for use in the preparation statements: `DONE` and `FAIL`. Use them with a following semicolon, as a statement.

`DONE`

> Use the `DONE` macro to end RTL generation for the pattern. The only RTL insns resulting from the pattern on this occasion will be those already emitted by explicit calls to `emit_insn` within the preparation statements; the RTL template will not be generated.

`FAIL`

> Make the pattern fail on this occasion. When a pattern fails, it means that the pattern was not truly available. The calling routines in the compiler will try other strategies for code generation using other patterns.
>
> Failure is currently supported only for binary (addition, multiplication, shifting, etc.) and bitfield (`extv`, `extzv`, and `insv`) operations.

Here is an example, the definition of left-shift for the SPUR chip:

```
(define_expand "ashlsi3"
  [(set (match_operand:SI 0 "register_operand" "")
        (ashift:SI
          (match_operand:SI 1 "register_operand" "")
          (match_operand:SI 2 "nonmemory_operand" "")))]
  ""
  "


{
  if (GET_CODE (operands[2]) != CONST_INT
      || (unsigned) INTVAL (operands[2]) > 3)
```


```
    FAIL;
}")
```

This example uses `define_expand` so that it can generate an RTL insn for shifting when the shift-count is in the supported range of 0 to 3 but fail in other cases where machine insns aren't available. When it fails, the compiler tries another strategy using different patterns (such as, a library call).

If the compiler were able to handle nontrivial condition-strings in patterns with names, then it would be possible to use a `define_insn` in that case. Here is another case (zero-extension on the 68000) which makes more use of the power of `define_expand`:

```
(define_expand "zero_extendhisi2"
  [(set (match_operand:SI 0 "general_operand" "")
        (const_int 0))
   (set (strict_low_part
          (subreg:HI
            (match_dup 0)
            0))
        (match_operand:HI 1 "general_operand" ""))]
  ""
  "operands[1] = make_safe_from (operands[1], operands[0]);")
```

Here two RTL insns are generated, one to clear the entire output operand and the other to copy the input operand into its low half. This sequence is incorrect if the input operand refers to [the old value of] the output operand, so the preparation statement makes sure this isn't so. The function `make_safe_from` copies the `operands[1]` into a temporary register if it refers to `operands[0]`. It does this by emitting another RTL insn.

Finally, a third example shows the use of an internal operand. Zero-extension on the SPUR chip is done by `and`-ing the result against a halfword mask. But this mask cannot be represented by a `const_int` because the constant value is too large to be legitimate on this machine. So it must be copied into a register with `force_reg` and then the register used in the `and`.

```
(define_expand "zero_extendhisi2"
  [(set (match_operand:SI 0 "register_operand" "")
        (and:SI (subreg:SI
                  (match_operand:HI 1 "register_operand" "")
                  0)
                (match_dup 2)))]
  ""
  "operands[2]
     = force_reg (SImode, gen_rtx (CONST_INT,
                                   VOIDmode, 65535)); ")
```

Note: If the `define_expand` is used to serve a standard binary or unary arithmetic operation or a bitfield operation, then the last insn it generates must not be a `code_label`, `barrier` or `note`. It must be an `insn`, `jump_insn` or `call_insn`. If you don't need a real insn at the end, emit an insn to copy the result of the operation into itself. Such an insn will generate no code, but it can avoid problems in the compiler.

Defining How to Split Instructions

There are two cases where you should specify how to split a pattern into multiple insns. On machines that have instructions requiring delay slots (see section Delay Slot Scheduling) or that have instructions whose output is not available for multiple cycles (see section Specifying Function Units), the compiler phases that optimize these cases need to be able to move insns into one-instruction delay slots. However, some insns may generate more than one machine instruction. These insns cannot be placed into a delay slot.

Often you can rewrite the single insn as a list of individual insns, each corresponding to one machine instruction. The disadvantage of doing so is that it will cause the compilation to be slower and require more space. If the resulting insns are too complex, it may also suppress some optimizations. The compiler splits the insn if there is a reason to believe that it might improve instruction or delay slot scheduling.

The insn combiner phase also splits putative insns. If three insns are merged into one insn with a complex expression that cannot be matched by some `define_insn` pattern, the combiner phase attempts to split the complex pattern into two insns that are recognized. Usually it can break the complex pattern into two patterns by splitting out some subexpression. However, in some other cases, such as performing an addition of a large constant in two insns on a RISC machine, the way to split the addition into two insns is machine-dependent.

The `define_split` definition tells the compiler how to split a complex insn into several simpler insns. It looks like this:

```
(define_split
  [insn-pattern]
  "condition"
  [new-insn-pattern-1
   new-insn-pattern-2
   ...]
  "preparation statements")
```

insn-pattern is a pattern that needs to be split and condition is the final condition to be tested, as in a `define_insn`. When an insn matching insn-pattern and satisfying condition is found, it is replaced in the insn list with the insns given by new-insn-pattern-1, new-insn-pattern-2, etc.

The preparation statements are similar to those statements that are specified for `define_expand` (see section Defining RTL Sequences for Code Generation) and are executed before the new RTL is generated to prepare for the generated code or emit some insns whose pattern is not fixed. Unlike those in `define_expand`, however, these statements must not generate any new pseudo-registers. Once reload has completed, they also must not allocate any space in the stack frame.

Patterns are matched against insn-pattern in two different circumstances. If an insn needs to be split for delay slot scheduling or insn scheduling, the insn is already known to be valid, which means that it must have been matched by some `define_insn` and, if `reload_completed` is non-zero, is known to satisfy the constraints of that `define_insn`. In that case, the new insn patterns must also be insns that are matched by some `define_insn` and, if `reload_completed` is non-zero, must also satisfy the constraints of those definitions.

As an example of this usage of `define_split`, consider the following example from `a29k.md', which splits a `sign_extend` from `HImode` to `SImode` into a pair of shift insns:

```
(define_split
  [(set (match_operand:SI 0 "gen_reg_operand" "")
        (sign_extend:SI (match_operand:HI 1 "gen_reg_operand" "")))]
  ""
  [(set (match_dup 0)
        (ashift:SI (match_dup 1)
                   (const_int 16)))
   (set (match_dup 0)
        (ashiftrt:SI (match_dup 0)
                     (const_int 16)))]
  "
{ operands[1] = gen_lowpart (SImode, operands[1]); }")
```

When the combiner phase tries to split an insn pattern, it is always the case that the pattern is *not* matched by any `define_insn`. The combiner pass first tries to split a single `set` expression and then the same `set` expression inside a `parallel`, but followed by a `clobber` of a pseudo-reg to use as a scratch register. In these cases, the combiner expects exactly two new insn patterns to be generated. It will verify that these patterns match some `define_insn` definitions, so you need not do this test in the `define_split` (of course, there is no point in writing a `define_split` that will never produce insns that match).

Here is an example of this use of `define_split`, taken from `rs6000.md':

```
(define_split
  [(set (match_operand:SI 0 "gen_reg_operand" "")
        (plus:SI (match_operand:SI 1 "gen_reg_operand" "")
                 (match_operand:SI 2 "non_add_cint_operand" "")))]
  ""
  [(set (match_dup 0) (plus:SI (match_dup 1) (match_dup 3)))
   (set (match_dup 0) (plus:SI (match_dup 0) (match_dup 4)))]
"
{
  int low = INTVAL (operands[2]) & 0xffff;
  int high = (unsigned) INTVAL (operands[2]) >> 16;

  if (low & 0x8000)
    high++, low |= 0xffff0000;

  operands[3] = gen_rtx (CONST_INT, VOIDmode, high << 16);
  operands[4] = gen_rtx (CONST_INT, VOIDmode, low);
}")
```

Here the predicate `non_add_cint_operand` matches any `const_int` that is *not* a valid operand of a single add insn. The add with the smaller displacement is written so that it can be substituted into the address of a subsequent operation.

An example that uses a scratch register, from the same file, generates an equality comparison of a regist and a large constant:

```
(define_split
  [(set (match_operand:CC 0 "cc_reg_operand" "")
        (compare:CC (match_operand:SI 1 "gen_reg_operand" "")
                    (match_operand:SI 2 "non_short_cint_operand" "")))
   (clobber (match_operand:SI 3 "gen_reg_operand" ""))]
  "find_single_use (operands[0], insn, 0)
   && (GET_CODE (*find_single_use (operands[0], insn, 0)) == EQ
       || GET_CODE (*find_single_use (operands[0], insn, 0)) == NE)"
  [(set (match_dup 3) (xor:SI (match_dup 1) (match_dup 4)))
   (set (match_dup 0) (compare:CC (match_dup 3) (match_dup 5)))]
  "
{
  /* Get the constant we are comparing against, C, and see what it
     looks like sign-extended to 16 bits.  Then see what constant
     could be XOR'ed with C to get the sign-extended value.  */

  int c = INTVAL (operands[2]);
  int sextc = (c << 16) >> 16;
  int xorv = c ^ sextc;

  operands[4] = gen_rtx (CONST_INT, VOIDmode, xorv);
  operands[5] = gen_rtx (CONST_INT, VOIDmode, sextc);
}")
```

To avoid confusion, don't write a single `define_split` that accepts some insns that match some `define_insn` as well as some insns that don't. Instead, write two separate `define_split` definitions, one for the insns that are valid and one for the insns that are not valid.

Instruction Attributes

In addition to describing the instruction supported by the target machine, the `md' file also defines a group of attributes and a set of values for each. Every generated insn is assigned a value for each attribute. One possible attribute would be the effect that the insn has on the machine's condition code. This attribute can then be used by `NOTICE_UPDATE_CC` to track the condition codes.

Defining Attributes and their Values

The `define_attr` expression is used to define each attribute required by the target machine. It looks like:

```
(define_attr name list-of-values default)
```

name is a string specifying the name of the attribute being defined.

list-of-values is either a string that specifies a comma-separated list of values that can be assigned to the

attribute, or a null string to indicate that the attribute takes numeric values.

default is an attribute expression that gives the value of this attribute for insns that match patterns whose definition does not include an explicit value for this attribute. See section Example of Attribute Specifications, for more information on the handling of defaults. See section Constant Attributes, for information on attributes that do not depend on any particular insn.

For each defined attribute, a number of definitions are written to the `insn-attr.h' file. For cases where an explicit set of values is specified for an attribute, the following are defined:

- A `#define' is written for the symbol `HAVE_ATTR_name'.
- An enumeral class is defined for `attr_name' with elements of the form `upper-name_upper-value' where the attribute name and value are first converted to upper case.
- A function `get_attr_name' is defined that is passed an insn and returns the attribute value for that insn.

For example, if the following is present in the `md' file:

```
(define_attr "type" "branch,fp,load,store,arith" ...)
```

the following lines will be written to the file `insn-attr.h'.

```
#define HAVE_ATTR_type
enum attr_type {TYPE_BRANCH, TYPE_FP, TYPE_LOAD,
                TYPE_STORE, TYPE_ARITH};
extern enum attr_type get_attr_type ();
```

If the attribute takes numeric values, no `enum` type will be defined and the function to obtain the attribute's value will return `int`.

Attribute Expressions

RTL expressions used to define attributes use the codes described above plus a few specific to attribute definitions, to be discussed below. Attribute value expressions must have one of the following forms:

`(const_int i)`

The integer i specifies the value of a numeric attribute. i must be non-negative.

The value of a numeric attribute can be specified either with a `const_int` or as an integer represented as a string in `const_string`, `eq_attr` (see below), and `set_attr` (see section Assigning Attribute Values to Insns) expressions.

`(const_string value)`

The string value specifies a constant attribute value. If value is specified as `"*"', it means that the default value of the attribute is to be used for the insn containing this expression. `"*"' obviously cannot be used in the default expression of a `define_attr`.

If the attribute whose value is being specified is numeric, value must be a string containing a non-negative integer (normally `const_int` would be used in this case). Otherwise, it must contain one of the valid values for the attribute.

`(if_then_else test true-value false-value)`

test specifies an attribute test, whose format is defined below. The value of this expression is true-value if test is true, otherwise it is false-value.

`(cond [test1 value1 ...] default)`

The first operand of this expression is a vector containing an even number of expressions and consisting of pairs of test and value expressions. The value of the `cond` expression is that of the value corresponding to the first true test expression. If none of the test expressions are true, the value of the `cond` expression is that of the default expression.

test expressions can have one of the following forms:

`(const_int i)`

This test is true if i is non-zero and false otherwise.

`(not test)`

`(ior test1 test2)`

`(and test1 test2)`

These tests are true if the indicated logical function is true.

`(match_operand:m n pred constraints)`

This test is true if operand n of the insn whose attribute value is being determined has mode m (this part of the test is ignored if m is `VOIDmode`) and the function specified by the string pred returns a non-zero value when passed operand n and mode m (this part of the test is ignored if pred is the null string).

The constraints operand is ignored and should be the null string.

`(le arith1 arith2)`

`(leu arith1 arith2)`

`(lt arith1 arith2)`

`(ltu arith1 arith2)`

`(gt arith1 arith2)`

`(gtu arith1 arith2)`

`(ge arith1 arith2)`

`(geu arith1 arith2)`

`(ne arith1 arith2)`

`(eq arith1 arith2)`

These tests are true if the indicated comparison of the two arithmetic expressions is true. Arithmetic expressions are formed with `plus`, `minus`, `mult`, `div`, `mod`, `abs`, `neg`, `and`, `ior`, `xor`, `not`, `ashift`, `lshiftrt`, and `ashiftrt` expressions.

`const_int` and `symbol_ref` are always valid terms (see section Computing the Length of an Insn,for additional forms). `symbol_ref` is a string denoting a C expression that yields an `int` when evaluated by the `get_attr_...' routine. It should normally be a global variable.

`(eq_attr name value)`

name is a string specifying the name of an attribute.

value is a string that is either a valid value for attribute name, a comma-separated list of values, or followed by a value or list. If value does not begin with a `!', this test is true if the value of the name attribute of the current insn is in the list specified by value. If value begins with a `!', this test is true if the attribute's value is *not* in the specified list.

For example,

```
(eq_attr "type" "load,store")
```

is equivalent to

```
(ior (eq_attr "type" "load") (eq_attr "type" "store"))
```

If name specifies an attribute of `alternative', it refers to the value of the compiler variable `which_alternative` (see section C Statements for Assembler Output) and the values must be small integers. For example,

```
(eq_attr "alternative" "2,3")
```

is equivalent to

```
(ior (eq (symbol_ref "which_alternative") (const_int 2))
     (eq (symbol_ref "which_alternative") (const_int 3)))
```

Note that, for most attributes, an `eq_attr` test is simplified in cases where the value of the attribute being tested is known for all insns matching a particular pattern. This is by far the most common case.

`(attr_flag name)`

The value of an `attr_flag` expression is true if the flag specified by name is true for the `insn` currently being scheduled.

name is a string specifying one of a fixed set of flags to test. Test the flags `forward` and `backward` to determine the direction of a conditional branch. Test the flags `very_likely`, `likely`, `very_unlikely`, and `unlikely` to determine if a conditional branch is expected to be taken.

If the `very_likely` flag is true, then the `likely` flag is also true. Likewise for the `very_unlikely` and `unlikely` flags.

This example describes a conditional branch delay slot which can be nullified for forward branches that are taken (annul-true) or for backward branches which are not taken (annul-false).

```
(define_delay (eq_attr "type" "cbranch")
  [(eq_attr "in_branch_delay" "true")
   (and (eq_attr "in_branch_delay" "true")
        (attr_flag "forward"))
   (and (eq_attr "in_branch_delay" "true")
        (attr_flag "backward"))])
```

The `forward` and `backward` flags are false if the current `insn` being scheduled is not a conditional branch.

The `very_likely` and `likely` flags are true if the `insn` being scheduled is not a conditional branch. The The `very_unlikely` and `unlikely` flags are false if the `insn` being scheduled is not a conditional branch.

`attr_flag` is only used during delay slot scheduling and has no meaning to other passes of the compiler.

Assigning Attribute Values to Insns

The value assigned to an attribute of an insn is primarily determined by which pattern is matched by that insn (or which `define_peephole` generated it). Every `define_insn` and `define_peephole` can have an optional last argument to specify the values of attributes for matching insns. The value of any attribute not specified in a particular insn is set to the default value for that attribute, as specified in its `define_attr`. Extensive use of default values for attributes permits the specification of the values for only one or two attributes in the definition of most insn patterns, as seen in the example in the next section.

The optional last argument of `define_insn` and `define_peephole` is a vector of expressions, each of which defines the value for a single attribute. The most general way of assigning an attribute's value is to use a `set` expression whose first operand is an `attr` expression giving the name of the attribute being set. The second operand of the `set` is an attribute expression (see section Attribute Expressions) giving the value of the attribute.

When the attribute value depends on the `alternative' attribute (i.e., which is the applicable alternative in the constraint of the insn), the `set_attr_alternative` expression can be used. It allows the specification of a vector of attribute expressions, one for each alternative.

When the generality of arbitrary attribute expressions is not required, the simpler `set_attr` expression can be used, which allows specifying a string giving either a single attribute value or a list of attribute values, one for each alternative.

The form of each of the above specifications is shown below. In each case, name is a string specifying the attribute to be set.

`(set_attr name value-string)`

value-string is either a string giving the desired attribute value, or a string containing a comma-separated list giving the values for succeeding alternatives. The number of elements must match the number of alternatives in the constraint of the insn pattern.

Note that it may be useful to specify `*' for some alternative, in which case the attribute will assume its default value for insns matching that alternative.

`(set_attr_alternative name [value1 value2 ...])`

Depending on the alternative of the insn, the value will be one of the specified values. This is a shorthand for using a `cond` with tests on the `alternative' attribute.

`(set (attr name) value)`

The first operand of this `set` must be the special RTL expression `attr`, whose sole operand is a string giving the name of the attribute being set. value is the value of the attribute.

The following shows three different ways of representing the same attribute value specification:


```
(set_attr "type" "load,store,arith")

(set_attr_alternative "type"
                      [(const_string "load") (const_string "store")
                       (const_string "arith")])

(set (attr "type")
     (cond [(eq_attr "alternative" "1") (const_string "load")
            (eq_attr "alternative" "2") (const_string "store")]
           (const_string "arith")))
```

The `define_asm_attributes` expression provides a mechanism to specify the attributes assigned to insns produced from an `asm` statement. It has the form:

```
(define_asm_attributes [attr-sets])
```

where attr-sets is specified the same as for both the `define_insn` and the `define_peephole` expressions.

These values will typically be the "worst case" attribute values. For example, they might indicate that the condition code will be clobbered.

A specification for a `length` attribute is handled specially. The way to compute the length of an `asm` insn is to multiply the length specified in the expression `define_asm_attributes` by the number of machine instructions specified in the `asm` statement, determined by counting the number of semicolons and newlines in the string. Therefore, the value of the `length` attribute specified in a `define_asm_attributes` should be the maximum possible length of a single machine instruction.

Example of Attribute Specifications

The judicious use of defaulting is important in the efficient use of insn attributes. Typically, insns are divided into types and an attribute, customarily called `type`, is used to represent this value. This attribute is normally used only to define the default value for other attributes. An example will clarify this usage.

Assume we have a RISC machine with a condition code and in which only full-word operations are performed in registers. Let us assume that we can divide all insns into loads, stores, (integer) arithmetic operations, floating point operations, and branches.

Here we will concern ourselves with determining the effect of an insn on the condition code and will limit ourselves to the following possible effects: The condition code can be set unpredictably (clobbered), not be changed, be set to agree with the results of the operation, or only changed if the item previously set into the condition code has been modified.

Here is part of a sample `` `md' `` file for such a machine:

```
(define_attr "type" "load,store,arith,fp,branch" (const_string "arith"))

(define_attr "cc" "clobber,unchanged,set,change0"
```


```
             (cond [(eq_attr "type" "load")
                        (const_string "change0")
                    (eq_attr "type" "store,branch")
                        (const_string "unchanged")
                    (eq_attr "type" "arith")
                        (if_then_else (match_operand:SI 0 "" "")
                                      (const_string "set")
                                      (const_string "clobber"))]
                   (const_string "clobber")))

(define_insn ""
  [(set (match_operand:SI 0 "general_operand" "=r,r,m")
        (match_operand:SI 1 "general_operand" "r,m,r"))]
  ""
  "@
   move %0,%1
   load %0,%1
   store %0,%1"
  [(set_attr "type" "arith,load,store")])
```

Note that we assume in the above example that arithmetic operations performed on quantities smaller than a machine word clobber the condition code since they will set the condition code to a value corresponding to the full-word result.

Computing the Length of an Insn

For many machines, multiple types of branch instructions are provided, each for different length branch displacements. In most cases, the assembler will choose the correct instruction to use. However, when the assembler cannot do so, GCC can when a special attribute, the `length' attribute, is defined. This attribute must be defined to have numeric values by specifying a null string in its `define_attr`.

In the case of the `length' attribute, two additional forms of arithmetic terms are allowed in test expressions:

`(match_dup n)`

This refers to the address of operand n of the current insn, which must be a `label_ref`.

`(pc)`

This refers to the address of the *current* insn. It might have been more consistent with other usage to make this the address of the *next* insn but this would be confusing because the length of the current insn is to be computed.

For normal insns, the length will be determined by value of the `length' attribute. In the case of `addr_vec` and `addr_diff_vec` insn patterns, the length is computed as the number of vectors multiplied by the size of each vector.

Lengths are measured in addressable storage units (bytes).

The following macros can be used to refine the length computation:

`FIRST_INSN_ADDRESS`

When the `length` insn attribute is used, this macro specifies the value to be assigned to the addr of the first insn in a function. If not specified, 0 is used.

`ADJUST_INSN_LENGTH (insn, length)`

If defined, modifies the length assigned to instruction insn as a function of the context in which it is used. length is an lvalue that contains the initially computed length of the insn and should be updated with the correct length of the insn. If updating is required, insn must not be a varying-length insn.

This macro will normally not be required. A case in which it is required is the ROMP. On this machine, the size of an `addr_vec` insn must be increased by two to compensate for the fact that alignment may be required.

The routine that returns `get_attr_length` (the value of the `length` attribute) can be used by the output routine to determine the form of the branch instruction to be written, as the example below illustrates.

As an example of the specification of variable-length branches, consider the IBM 360. If we adopt the convention that a register will be set to the starting address of a function, we can jump to labels within 4k of the start using a four-byte instruction. Otherwise, we need a six-byte sequence to load the address from memory and then branch to it.

On such a machine, a pattern for a branch instruction might be specified as follows:

```
(define_insn "jump"
  [(set (pc)
        (label_ref (match_operand 0 "" "")))]
  ""
  "*
{
   return (get_attr_length (insn) == 4
           ? \"b %l0\" : \"l r15,=a(%l0); br r15\");
}"
  [(set (attr "length") (if_then_else (lt (match_dup 0) (const_int 4096))
                                      (const_int 4)
                                      (const_int 6)))])
```

Constant Attributes

A special form of `define_attr`, where the expression for the default value is a `const` expression, indicates an attribute that is constant for a given run of the compiler. Constant attributes may be used to specify which variety of processor is used. For example,

```
(define_attr "cpu" "m88100,m88110,m88000"
 (const
  (cond [(symbol_ref "TARGET_88100") (const_string "m88100")
         (symbol_ref "TARGET_88110") (const_string "m88110")]
        (const_string "m88000"))))

(define_attr "memory" "fast,slow"
```


```
(const
 (if_then_else (symbol_ref "TARGET_FAST_MEM")
               (const_string "fast")
               (const_string "slow"))))
```

The routine generated for constant attributes has no parameters as it does not depend on any particular insn. RTL expressions used to define the value of a constant attribute may use the `symbol_ref` form, but may not use either the `match_operand` form or `eq_attr` forms involving insn attributes.

Delay Slot Scheduling

The insn attribute mechanism can be used to specify the requirements for delay slots, if any, on a target machine. An instruction is said to require a delay slot if some instructions that are physically after the instruction are executed as if they were located before it. Classic examples are branch and call instructions, which often execute the following instruction before the branch or call is performed.

On some machines, conditional branch instructions can optionally annul instructions in the delay slot. This means that the instruction will not be executed for certain branch outcomes. Both instructions that annul if the branch is true and instructions that annul if the branch is false are supported. Delay slot scheduling differs from instruction scheduling in that determining whether an instruction needs a delay slot is dependent only on the type of instruction being generated, not on data flow between the instructions. See the next section for a discussion of data-dependent instruction scheduling.

The requirement of an insn needing one or more delay slots is indicated via the `define_delay` expression. It has the following form:

```
(define_delay test
              [delay-1 annul-true-1 annul-false-1
               delay-2 annul-true-2 annul-false-2
               ...])
```

test is an attribute test that indicates whether this `define_delay` applies to a particular insn. If so, the number of required delay slots is determined by the length of the vector specified as the second argument. An insn placed in delay slot n must satisfy attribute test delay-n. annul-true-n is an attribute test that specifies which insns may be annulled if the branch is true. Similarly, annul-false-n specifies which insns in the delay slot may be annulled if the branch is false. If annulling is not supported for that delay slot, `(nil)` should be coded.

For example, in the common case where branch and call insns require a single delay slot, which may contain any insn other than a branch or call, the following would be placed in the `` `md' `` file:

```
(define_delay (eq_attr "type" "branch,call")
              [(eq_attr "type" "!branch,call") (nil) (nil)])
```

Multiple `define_delay` expressions may be specified. In this case, each such expression specifies different delay slot requirements and there must be no insn for which tests in two `define_delay` expressions are both true.

For example, if we have a machine that requires one delay slot for branches but two for calls, no delay slot

can contain a branch or call insn, and any valid insn in the delay slot for the branch can be annulled if th branch is true, we might represent this as follows:

```
(define_delay (eq_attr "type" "branch")
   [(eq_attr "type" "!branch,call")
    (eq_attr "type" "!branch,call")
    (nil)])

(define_delay (eq_attr "type" "call")
              [(eq_attr "type" "!branch,call") (nil) (nil)
               (eq_attr "type" "!branch,call") (nil) (nil)])
```

Specifying Function Units

On most RISC machines, there are instructions whose results are not available for a specific number of cycles. Common cases are instructions that load data from memory. On many machines, a pipeline stall will result if the data is referenced too soon after the load instruction.

In addition, many newer microprocessors have multiple function units, usually one for integer and one for floating point, and often will incur pipeline stalls when a result that is needed is not yet ready.

The descriptions in this section allow the specification of how much time must elapse between the execution of an instruction and the time when its result is used. It also allows specification of when the execution of an instruction will delay execution of similar instructions due to function unit conflicts.

For the purposes of the specifications in this section, a machine is divided into function units, each of which execute a specific class of instructions in first-in-first-out order. Function units that accept one instruction each cycle and allow a result to be used in the succeeding instruction (usually via forwarding) need not be specified. Classic RISC microprocessors will normally have a single function unit, which we can call `memory'. The newer "superscalar" processors will often have function units for floating point operations, usually at least a floating point adder and multiplier.

Each usage of a function units by a class of insns is specified with a `define_function_unit` expression, which looks like this:

```
(define_function_unit name multiplicity simultaneity
                      test ready-delay issue-delay
                     [conflict-list])
```

name is a string giving the name of the function unit.

multiplicity is an integer specifying the number of identical units in the processor. If more than one unit is specified, they will be scheduled independently. Only truly independent units should be counted; a pipelined unit should be specified as a single unit. (The only common example of a machine that has multiple function units for a single instruction class that are truly independent and not pipelined are the two multiply and two increment units of the CDC 6600.)

simultaneity specifies the maximum number of insns that can be executing in each instance of the function unit simultaneously or zero if the unit is pipelined and has no limit.

All `define_function_unit` definitions referring to function unit name must have the same name a values for multiplicity and simultaneity.

test is an attribute test that selects the insns we are describing in this definition. Note that an insn may use more than one function unit and a function unit may be specified in more than one `define_function_unit`.

ready-delay is an integer that specifies the number of cycles after which the result of the instruction can be used without introducing any stalls.

issue-delay is an integer that specifies the number of cycles after the instruction matching the test expression begins using this unit until a subsequent instruction can begin. A cost of N indicates an N-1 cycle delay. A subsequent instruction may also be delayed if an earlier instruction has a longer ready-delay value. This blocking effect is computed using the simultaneity, ready-delay, issue-delay, and conflict-list terms. For a normal non-pipelined function unit, simultaneity is one, the unit is taken to block for the ready-delay cycles of the executing insn, and smaller values of issue-delay are ignored.

conflict-list is an optional list giving detailed conflict costs for this unit. If specified, it is a list of condition test expressions to be applied to insns chosen to execute in name following the particular insn matching test that is already executing in name. For each insn in the list, issue-delay specifies the conflict cost; for insns not in the list, the cost is zero. If not specified, conflict-list defaults to all instructions that use the function unit.

Typical uses of this vector are where a floating point function unit can pipeline either single- or double-precision operations, but not both, or where a memory unit can pipeline loads, but not stores, etc.

As an example, consider a classic RISC machine where the result of a load instruction is not available for two cycles (a single "delay" instruction is required) and where only one load instruction can be executed simultaneously. This would be specified as:

```
(define_function_unit "memory" 1 1 (eq_attr "type" "load") 2 0)
```

For the case of a floating point function unit that can pipeline either single or double precision, but not both, the following could be specified:

```
(define_function_unit
   "fp" 1 0 (eq_attr "type" "sp_fp") 4 4 [(eq_attr "type" "dp_fp")])
(define_function_unit
   "fp" 1 0 (eq_attr "type" "dp_fp") 4 4 [(eq_attr "type" "sp_fp")])
```

Note: The scheduler attempts to avoid function unit conflicts and uses all the specifications in the `define_function_unit` expression. It has recently come to our attention that these specifications may not allow modeling of some of the newer "superscalar" processors that have insns using multiple pipelined units. These insns will cause a potential conflict for the second unit used during their execution and there is no way of representing that conflict. We welcome any examples of how function unit conflicts work in such processors and suggestions for their representation.

Go to the previous, next section.

Go to the previous, next section.

Target Description Macros

In addition to the file `machine.md', a machine description includes a C header file conventionally given the name `machine.h'. This header file defines numerous macros that convey the information about the target machine that does not fit into the scheme of the `.md' file. The file `tm.h' should be a link to `machine.h'. The header file `config.h' includes `tm.h' and most compiler source files include `config.h'.

Controlling the Compilation Driver, `gcc'

You can control the compilation driver.

`SWITCH_TAKES_ARG (char)`

A C expression which determines whether the option `-char' takes arguments. The value should be the number of arguments that option takes--zero, for many options.

By default, this macro is defined to handle the standard options properly. You need not define it unless you wish to add additional options which take arguments.

`WORD_SWITCH_TAKES_ARG (name)`

A C expression which determines whether the option `-name' takes arguments. The value should be the number of arguments that option takes--zero, for many options. This macro rather than `SWITCH_TAKES_ARG` is used for multi-character option names.

By default, this macro is defined as `DEFAULT_WORD_SWITCH_TAKES_ARG`, which handles the standard options properly. You need not define `WORD_SWITCH_TAKES_ARG` unless you wish to add additional options which take arguments. Any redefinition should call `DEFAULT_WORD_SWITCH_TAKES_ARG` and then check for additional options.

`SWITCHES_NEED_SPACES`

A string-valued C expression which is nonempty if the linker needs a space between the `-L' or `-o' option and its argument.

If this macro is not defined, the default value is 0.

`CPP_SPEC`

A C string constant that tells the GNU CC driver program options to pass to CPP. It can also specify how to translate options you give to GNU CC into options for GNU CC to pass to the CPP.

Do not define this macro if it does not need to do anything.

`NO_BUILTIN_SIZE_TYPE`

If this macro is defined, the preprocessor will not define the builtin macro `__SIZE_TYPE__`. The macro `__SIZE_TYPE__` must then be defined by `CPP_SPEC` instead.

This should be defined if `SIZE_TYPE` depends on target dependent flags which are not accessible

to the preprocessor. Otherwise, it should not be defined.

`NO_BUILTIN_PTRDIFF_TYPE`

If this macro is defined, the preprocessor will not define the builtin macro `__PTRDIFF_TYPE__`. The macro `__PTRDIFF_TYPE__` must then be defined by `CPP_SPEC` instead.

This should be defined if `PTRDIFF_TYPE` depends on target dependent flags which are not accessible to the preprocessor. Otherwise, it should not be defined.

`SIGNED_CHAR_SPEC`

A C string constant that tells the GNU CC driver program options to pass to CPP. By default, this macro is defined to pass the option `-D__CHAR_UNSIGNED__' to CPP if `char` will be treated as `unsigned char` by `cc1`.

Do not define this macro unless you need to override the default definition.

`CC1_SPEC`

A C string constant that tells the GNU CC driver program options to pass to `cc1`. It can also specify how to translate options you give to GNU CC into options for GNU CC to pass to the `cc1`.

Do not define this macro if it does not need to do anything.

`CC1PLUS_SPEC`

A C string constant that tells the GNU CC driver program options to pass to `cc1plus`. It can also specify how to translate options you give to GNU CC into options for GNU CC to pass to the `cc1plus`.

Do not define this macro if it does not need to do anything.

`ASM_SPEC`

A C string constant that tells the GNU CC driver program options to pass to the assembler. It can also specify how to translate options you give to GNU CC into options for GNU CC to pass to the assembler. See the file `` `sun3.h' `` for an example of this.

Do not define this macro if it does not need to do anything.

`ASM_FINAL_SPEC`

A C string constant that tells the GNU CC driver program how to run any programs which cleanup after the normal assembler. Normally, this is not needed. See the file `` `mips.h' `` for an example of this.

Do not define this macro if it does not need to do anything.

`LINK_SPEC`

A C string constant that tells the GNU CC driver program options to pass to the linker. It can also specify how to translate options you give to GNU CC into options for GNU CC to pass to the linker.

Do not define this macro if it does not need to do anything.

`LIB_SPEC`

Another C string constant used much like `LINK_SPEC`. The difference between the two is that

`LIB_SPEC` is used at the end of the command given to the linker.

If this macro is not defined, a default is provided that loads the standard C library from the usual place. See `gcc.c'.

`LIBGCC_SPEC`

Another C string constant that tells the GNU CC driver program how and when to place a reference to `libgcc.a' into the linker command line. This constant is placed both before and after the value of `LIB_SPEC`.

If this macro is not defined, the GNU CC driver provides a default that passes the string `-lgcc' to the linker unless the `-shared' option is specified.

`STARTFILE_SPEC`

Another C string constant used much like `LINK_SPEC`. The difference between the two is that `STARTFILE_SPEC` is used at the very beginning of the command given to the linker.

If this macro is not defined, a default is provided that loads the standard C startup file from the usual place. See `gcc.c'.

`ENDFILE_SPEC`

Another C string constant used much like `LINK_SPEC`. The difference between the two is that `ENDFILE_SPEC` is used at the very end of the command given to the linker.

Do not define this macro if it does not need to do anything.

`LINK_LIBGCC_SPECIAL`

Define this macro meaning that `gcc` should find the library `libgcc.a' by hand, rather than passing the argument `-lgcc' to tell the linker to do the search; also, `gcc` should not generate `-L' options to pass to the linker (as it normally does).

`LINK_LIBGCC_SPECIAL_1`

Define this macro meaning that `gcc` should find the library `libgcc.a' by hand, rather than passing the argument `-lgcc' to tell the linker to do the search.

`RELATIVE_PREFIX_NOT_LINKDIR`

Define this macro to tell `gcc` that it should only translate a `-B' prefix into a `-L' linker option if the prefix indicates an absolute file name.

`STANDARD_EXEC_PREFIX`

Define this macro as a C string constant if you wish to override the standard choice of `/usr/local/lib/gcc-lib/' as the default prefix to try when searching for the executable files of the compiler.

`MD_EXEC_PREFIX`

If defined, this macro is an additional prefix to try after `STANDARD_EXEC_PREFIX`. `MD_EXEC_PREFIX` is not searched when the `-b' option is used, or the compiler is built as a cross compiler.

`STANDARD_STARTFILE_PREFIX`

Define this macro as a C string constant if you wish to override the standard choice of

`/usr/local/lib/' as the default prefix to try when searching for startup files such as `crt0.o'.

`MD_STARTFILE_PREFIX`

If defined, this macro supplies an additional prefix to try after the standard prefixes. `MD_EXEC_PREFIX` is not searched when the `-b' option is used, or when the compiler is built as a cross compiler.

`MD_STARTFILE_PREFIX_1`

If defined, this macro supplies yet another prefix to try after the standard prefixes. It is not searched when the `-b' option is used, or when the compiler is built as a cross compiler.

`INIT_ENVIRONMENT`

Define this macro as a C string constant if you with to set environment variables for programs called by the driver, such as the assembler and loader. The driver passes the value of this macro to `putenv` to initialize the necessary environment variables.

`LOCAL_INCLUDE_DIR`

Define this macro as a C string constant if you wish to override the standard choice of `/usr/local/include' as the default prefix to try when searching for local header files. `LOCAL_INCLUDE_DIR` comes before `SYSTEM_INCLUDE_DIR` in the search order.

Cross compilers do not use this macro and do not search either `/usr/local/include' or its replacement.

`SYSTEM_INCLUDE_DIR`

Define this macro as a C string constant if you wish to specify a system-specific directory to search for header files before the standard directory. `SYSTEM_INCLUDE_DIR` comes before `STANDARD_INCLUDE_DIR` in the search order.

Cross compilers do not use this macro and do not search the directory specified.

`STANDARD_INCLUDE_DIR`

Define this macro as a C string constant if you wish to override the standard choice of `/usr/include' as the default prefix to try when searching for header files.

Cross compilers do not use this macro and do not search either `/usr/include' or its replacement.

`INCLUDE_DEFAULTS`

Define this macro if you wish to override the entire default search path for include files. The default search path includes `GCC_INCLUDE_DIR`, `LOCAL_INCLUDE_DIR`, `SYSTEM_INCLUDE_DIR`, `GPLUSPLUS_INCLUDE_DIR`, and `STANDARD_INCLUDE_DIR`. In addition, `GPLUSPLUS_INCLUDE_DIR` and `GCC_INCLUDE_DIR` are defined automatically by `Makefile', and specify private search areas for GCC. The directory `GPLUSPLUS_INCLUDE_DIR` is used only for C++ programs.

The definition should be an initializer for an array of structures. Each array element should have two elements: the directory name (a string constant) and a flag for C++-only directories. Mark the end of the array with a null element. For example, here is the definition used for VMS:


```
#define INCLUDE_DEFAULTS \
{                                       \
  { "GNU_GXX_INCLUDE:", 1},             \
  { "GNU_CC_INCLUDE:", 0},              \
  { "SYS$SYSROOT:[SYSLIB.]", 0},        \
  { ".", 0},                            \
  { 0, 0}                               \
}
```

Here is the order of prefixes tried for exec files:

1. Any prefixes specified by the user with `-B'.
2. The environment variable `GCC_EXEC_PREFIX`, if any.
3. The directories specified by the environment variable `COMPILER_PATH`.
4. The macro `STANDARD_EXEC_PREFIX`.
5. `/usr/lib/gcc/'.
6. The macro `MD_EXEC_PREFIX`, if any.

Here is the order of prefixes tried for startfiles:

1. Any prefixes specified by the user with `-B'.
2. The environment variable `GCC_EXEC_PREFIX`, if any.
3. The directories specified by the environment variable `LIBRARY_PATH` (native only, cross compilers do not use this).
4. The macro `STANDARD_EXEC_PREFIX`.
5. `/usr/lib/gcc/'.
6. The macro `MD_EXEC_PREFIX`, if any.
7. The macro `MD_STARTFILE_PREFIX`, if any.
8. The macro `STANDARD_STARTFILE_PREFIX`.
9. `/lib/'.
10. `/usr/lib/'.

Run-time Target Specification

Here are run-time target specifications.

`CPP_PREDEFINES`

Define this to be a string constant containing `-D' options to define the predefined macros that identify this machine and system. These macros will be predefined unless the `-ansi' option is specified.

In addition, a parallel set of macros are predefined, whose names are made by appending `__' at the beginning and at the end. These `__' macros are permitted by the ANSI standard, so they are

predefined regardless of whether `-ansi' is specified.

For example, on the Sun, one can use the following value:

```
"-Dmc68000 -Dsun -Dunix"
```

The result is to define the macros `__mc68000__`, `__sun__` and `__unix__` unconditionally, and the macros `mc68000`, `sun` and `unix` provided `-ansi' is not specified.

`extern int target_flags;`

This declaration should be present.

`TARGET_...`

This series of macros is to allow compiler command arguments to enable or disable the use of optional features of the target machine. For example, one machine description serves both the 68000 and the 68020; a command argument tells the compiler whether it should use 68020-only instructions or not. This command argument works by means of a macro `TARGET_68020` that tests a bit in `target_flags`.

Define a macro `TARGET_featurename` for each such option. Its definition should test a bit in `target_flags`; for example:

```
#define TARGET_68020 (target_flags & 1)
```

One place where these macros are used is in the condition-expressions of instruction patterns. Note how `TARGET_68020` appears frequently in the 68000 machine description file, `m68k.md'. Another place they are used is in the definitions of the other macros in the `machine.h' file.

`TARGET_SWITCHES`

This macro defines names of command options to set and clear bits in `target_flags`. Its definition is an initializer with a subgrouping for each command option.

Each subgrouping contains a string constant, that defines the option name, and a number, which contains the bits to set in `target_flags`. A negative number says to clear bits instead; the negative of the number is which bits to clear. The actual option name is made by appending `-m' to the specified name.

One of the subgroupings should have a null string. The number in this grouping is the default value for `target_flags`. Any target options act starting with that value.

Here is an example which defines `-m68000' and `-m68020' with opposite meanings, and picks the latter as the default:

```
#define TARGET_SWITCHES \
  { { "68020", 1},      \
    { "68000", -1},     \
    { "", 1}}
```

`TARGET_OPTIONS`

This macro is similar to `TARGET_SWITCHES` but defines names of command options that ha values. Its definition is an initializer with a subgrouping for each command option.

Each subgrouping contains a string constant, that defines the fixed part of the option name, and the address of a variable. The variable, type `char *`, is set to the variable part of the given option if the fixed part matches. The actual option name is made by appending `-m' to the specified name.

Here is an example which defines `-mshort-data-number'. If the given option is `-mshort-data-512', the variable `m88k_short_data` will be set to the string `"512"`.

```
extern char *m88k_short_data;
#define TARGET_OPTIONS \
 { { "short-data-", &m88k_short_data } }
```

`TARGET_VERSION`

This macro is a C statement to print on `stderr` a string describing the particular machine description choice. Every machine description should define `TARGET_VERSION`. For example:

```
#ifdef MOTOROLA
#define TARGET_VERSION \
  fprintf (stderr, " (68k, Motorola syntax)");
#else
#define TARGET_VERSION \
  fprintf (stderr, " (68k, MIT syntax)");
#endif
```

`OVERRIDE_OPTIONS`

Sometimes certain combinations of command options do not make sense on a particular target machine. You can define a macro `OVERRIDE_OPTIONS` to take account of this. This macro, if defined, is executed once just after all the command options have been parsed.

Don't use this macro to turn on various extra optimizations for `-O'. That is what `OPTIMIZATION_OPTIONS` is for.

`OPTIMIZATION_OPTIONS (level)`

Some machines may desire to change what optimizations are performed for various optimization levels. This macro, if defined, is executed once just after the optimization level is determined and before the remainder of the command options have been parsed. Values set in this macro are used as the default values for the other command line options.

level is the optimization level specified; 2 if `-O2' is specified, 1 if `-O' is specified, and 0 if neither is specified.

You should not use this macro to change options that are not machine-specific. These should uniformly selected by the same optimization level on all supported machines. Use this macro to enable machbine-specific optimizations.

Do not examine `write_symbols` in this macro! The debugging options are not supposed to alter the generated code.

`CAN_DEBUG_WITHOUT_FP`

Define this macro if debugging can be performed even without a frame pointer. If this macro is defined, GNU CC will turn on the `-fomit-frame-pointer' option whenever `-O' is specified.

Storage Layout

Note that the definitions of the macros in this table which are sizes or alignments measured in bits do not need to be constant. They can be C expressions that refer to static variables, such as the `target_flags`. See section Run-time Target Specification.

`BITS_BIG_ENDIAN`

Define this macro to have the value 1 if the most significant bit in a byte has the lowest number; otherwise define it to have the value zero. This means that bit-field instructions count from the most significant bit. If the machine has no bit-field instructions, then this must still be defined, but it doesn't matter which value it is defined to. This macro need not be a constant.

This macro does not affect the way structure fields are packed into bytes or words; that is controlled by `BYTES_BIG_ENDIAN`.

`BYTES_BIG_ENDIAN`

Define this macro to have the value 1 if the most significant byte in a word has the lowest number. This macro need not be a constant.

`WORDS_BIG_ENDIAN`

Define this macro to have the value 1 if, in a multiword object, the most significant word has the lowest number. This applies to both memory locations and registers; GNU CC fundamentally assumes that the order of words in memory is the same as the order in registers. This macro need not be a constant.

`LIBGCC2_WORDS_BIG_ENDIAN`

Define this macro if WORDS_BIG_ENDIAN is not constant. This must be a constant value with the same meaning as WORDS_BIG_ENDIAN, which will be used only when compiling libgcc2.c. Typically the value will be set based on preprocessor defines.

`FLOAT_WORDS_BIG_ENDIAN`

Define this macro to have the value 1 if `DFmode`, `XFmode` or `TFmode` floating point numbers are stored in memory with the word containing the sign bit at the lowest address; otherwise define it to have the value 0. This macro need not be a constant.

You need not define this macro if the ordering is the same as for multi-word integers.

`BITS_PER_UNIT`

Define this macro to be the number of bits in an addressable storage unit (byte); normally 8.

`BITS_PER_WORD`

Number of bits in a word; normally 32.

`MAX_BITS_PER_WORD`

Maximum number of bits in a word. If this is undefined, the default is `BITS_PER_WORD`.

Otherwise, it is the constant value that is the largest value that `BITS_PER_WORD` can have at run-time.

`UNITS_PER_WORD`

Number of storage units in a word; normally 4.

`MIN_UNITS_PER_WORD`

Minimum number of units in a word. If this is undefined, the default is `UNITS_PER_WORD`. Otherwise, it is the constant value that is the smallest value that `UNITS_PER_WORD` can have at run-time.

`POINTER_SIZE`

Width of a pointer, in bits. You must specify a value no wider than the width of `Pmode`. If it is not equal to the width of `Pmode`, you must define `POINTERS_EXTEND_UNSIGNED`.

`POINTERS_EXTEND_UNSIGNED`

A C expression whose value is nonzero if pointers that need to be extended from being `POINTER_SIZE` bits wide to `Pmode` are sign-extended and zero if they are zero-extended.

You need not define this macro if the `POINTER_SIZE` is equal to the width of `Pmode`.

`PROMOTE_MODE (m, unsignedp, type)`

A macro to update m and unsignedp when an object whose type is type and which has the specified mode and signedness is to be stored in a register. This macro is only called when type is a scalar type.

On most RISC machines, which only have operations that operate on a full register, define this macro to set m to `word_mode` if m is an integer mode narrower than `BITS_PER_WORD`. In most cases, only integer modes should be widened because wider-precision floating-point operations are usually more expensive than their narrower counterparts.

For most machines, the macro definition does not change unsignedp. However, some machines, have instructions that preferentially handle either signed or unsigned quantities of certain modes. For example, on the DEC Alpha, 32-bit loads from memory and 32-bit add instructions sign-extend the result to 64 bits. On such machines, set unsignedp according to which kind of extension is more efficient.

Do not define this macro if it would never modify m.

`PROMOTE_FUNCTION_ARGS`

Define this macro if the promotion described by `PROMOTE_MODE` should also be done for outgoing function arguments.

`PROMOTE_FUNCTION_RETURN`

Define this macro if the promotion described by `PROMOTE_MODE` should also be done for the return value of functions.

If this macro is defined, `FUNCTION_VALUE` must perform the same promotions done by `PROMOTE_MODE`.

`PROMOTE_FOR_CALL_ONLY`

Define this macro if the promotion described by `PROMOTE_MODE` should *only* be performed f outgoing function arguments or function return values, as specified by `PROMOTE_FUNCTION_ARGS` and `PROMOTE_FUNCTION_RETURN`, respectively.

`PARM_BOUNDARY`

Normal alignment required for function parameters on the stack, in bits. All stack parameters receive at least this much alignment regardless of data type. On most machines, this is the same as the size of an integer.

`STACK_BOUNDARY`

Define this macro if you wish to preserve a certain alignment for the stack pointer. The definition is a C expression for the desired alignment (measured in bits).

If `PUSH_ROUNDING` is not defined, the stack will always be aligned to the specified boundary. If `PUSH_ROUNDING` is defined and specifies a less strict alignment than `STACK_BOUNDARY`, the stack may be momentarily unaligned while pushing arguments.

`FUNCTION_BOUNDARY`

Alignment required for a function entry point, in bits.

`BIGGEST_ALIGNMENT`

Biggest alignment that any data type can require on this machine, in bits.

`BIGGEST_FIELD_ALIGNMENT`

Biggest alignment that any structure field can require on this machine, in bits. If defined, this overrides `BIGGEST_ALIGNMENT` for structure fields only.

`MAX_OFILE_ALIGNMENT`

Biggest alignment supported by the object file format of this machine. Use this macro to limit the alignment which can be specified using the `__attribute__ ((aligned (n)))` construct. If not defined, the default value is `BIGGEST_ALIGNMENT`.

`DATA_ALIGNMENT (type, basic-align)`

If defined, a C expression to compute the alignment for a static variable. type is the data type, and basic-align is the alignment that the object would ordinarily have. The value of this macro is used instead of that alignment to align the object.

If this macro is not defined, then basic-align is used.

One use of this macro is to increase alignment of medium-size data to make it all fit in fewer cache lines. Another is to cause character arrays to be word-aligned so that `strcpy` calls that copy constants to character arrays can be done inline.

`CONSTANT_ALIGNMENT (constant, basic-align)`

If defined, a C expression to compute the alignment given to a constant that is being placed in memory. constant is the constant and basic-align is the alignment that the object would ordinarily have. The value of this macro is used instead of that alignment to align the object.

If this macro is not defined, then basic-align is used.

The typical use of this macro is to increase alignment for string constants to be word aligned so

that `strcpy` calls that copy constants can be done inline.

`EMPTY_FIELD_BOUNDARY`

Alignment in bits to be given to a structure bit field that follows an empty field such as `int : 0;`.

Note that `PCC_BITFIELD_TYPE_MATTERS` also affects the alignment that results from an empty field.

`STRUCTURE_SIZE_BOUNDARY`

Number of bits which any structure or union's size must be a multiple of. Each structure or union's size is rounded up to a multiple of this.

If you do not define this macro, the default is the same as `BITS_PER_UNIT`.

`STRICT_ALIGNMENT`

Define this macro to be the value 1 if instructions will fail to work if given data not on the nominal alignment. If instructions will merely go slower in that case, define this macro as 0.

`PCC_BITFIELD_TYPE_MATTERS`

Define this if you wish to imitate the way many other C compilers handle alignment of bitfields and the structures that contain them.

The behavior is that the type written for a bitfield (`int`, `short`, or other integer type) imposes an alignment for the entire structure, as if the structure really did contain an ordinary field of that type. In addition, the bitfield is placed within the structure so that it would fit within such a field, not crossing a boundary for it.

Thus, on most machines, a bitfield whose type is written as `int` would not cross a four-byte boundary, and would force four-byte alignment for the whole structure. (The alignment used may not be four bytes; it is controlled by the other alignment parameters.)

If the macro is defined, its definition should be a C expression; a nonzero value for the expression enables this behavior.

Note that if this macro is not defined, or its value is zero, some bitfields may cross more than one alignment boundary. The compiler can support such references if there are `insv', `extv', and `extzv' insns that can directly reference memory.

The other known way of making bitfields work is to define `STRUCTURE_SIZE_BOUNDARY` as large as `BIGGEST_ALIGNMENT`. Then every structure can be accessed with fullwords.

Unless the machine has bitfield instructions or you define `STRUCTURE_SIZE_BOUNDARY` that way, you must define `PCC_BITFIELD_TYPE_MATTERS` to have a nonzero value.

If your aim is to make GNU CC use the same conventions for laying out bitfields as are used by another compiler, here is how to investigate what the other compiler does. Compile and run this program:

```
struct foo1
{
```


```
  char x;
  char :0;
  char y;
};

struct foo2
{
  char x;
  int :0;
  char y;
};

main ()
{
  printf ("Size of foo1 is %d\n",
          sizeof (struct foo1));
  printf ("Size of foo2 is %d\n",
          sizeof (struct foo2));
  exit (0);
}
```

If this prints 2 and 5, then the compiler's behavior is what you would get from `PCC_BITFIELD_TYPE_MATTERS`.

`BITFIELD_NBYTES_LIMITED`

Like PCC_BITFIELD_TYPE_MATTERS except that its effect is limited to aligning a bitfield within the structure.

`ROUND_TYPE_SIZE (struct, size, align)`

Define this macro as an expression for the overall size of a structure (given by struct as a tree node) when the size computed from the fields is size and the alignment is align.

The default is to round size up to a multiple of align.

`ROUND_TYPE_ALIGN (struct, computed, specified)`

Define this macro as an expression for the alignment of a structure (given by struct as a tree node) if the alignment computed in the usual way is computed and the alignment explicitly specified was specified.

The default is to use specified if it is larger; otherwise, use the smaller of computed and `BIGGEST_ALIGNMENT`

`MAX_FIXED_MODE_SIZE`

An integer expression for the size in bits of the largest integer machine mode that should actually be used. All integer machine modes of this size or smaller can be used for structures and unions with the appropriate sizes. If this macro is undefined, `GET_MODE_BITSIZE (DImode)` is assumed.

`CHECK_FLOAT_VALUE (mode, value, overflow)`

A C statement to validate the value value (of type `double`) for mode mode. This means that y check whether value fits within the possible range of values for mode mode on this target mac The mode mode is always a mode of class `MODE_FLOAT`. overflow is nonzero if the value is already known to be out of range.

If value is not valid or if overflow is nonzero, you should set overflow to 1 and then assign some valid value to value. Allowing an invalid value to go through the compiler can produce incorrect assembler code which may even cause Unix assemblers to crash.

This macro need not be defined if there is no work for it to do.

`TARGET_FLOAT_FORMAT`

A code distinguishing the floating point format of the target machine. There are three defined values:

`IEEE_FLOAT_FORMAT`

This code indicates IEEE floating point. It is the default; there is no need to define this macro when the format is IEEE.

`VAX_FLOAT_FORMAT`

This code indicates the peculiar format used on the Vax.

`UNKNOWN_FLOAT_FORMAT`

This code indicates any other format.

The value of this macro is compared with `HOST_FLOAT_FORMAT` (see section The Configuration File) to determine whether the target machine has the same format as the host machine. If any other formats are actually in use on supported machines, new codes should be defined for them.

The ordering of the component words of floating point values stored in memory is controlled by `FLOAT_WORDS_BIG_ENDIAN` for the target machine and `HOST_FLOAT_WORDS_BIG_ENDIAN` for the host.

Layout of Source Language Data Types

These macros define the sizes and other characteristics of the standard basic data types used in programs being compiled. Unlike the macros in the previous section, these apply to specific features of C and related languages, rather than to fundamental aspects of storage layout.

`INT_TYPE_SIZE`

A C expression for the size in bits of the type `int` on the target machine. If you don't define this, the default is one word.

`MAX_INT_TYPE_SIZE`

Maximum number for the size in bits of the type `int` on the target machine. If this is undefined, the default is `INT_TYPE_SIZE`. Otherwise, it is the constant value that is the largest value that `INT_TYPE_SIZE` can have at run-time. This is used in `cpp`.

`SHORT_TYPE_SIZE`

A C expression for the size in bits of the type `short` on the target machine. If you don't defin this, the default is half a word. (If this would be less than one storage unit, it is rounded up to c unit.)

`LONG_TYPE_SIZE`

A C expression for the size in bits of the type `long` on the target machine. If you don't define this, the default is one word.

`MAX_LONG_TYPE_SIZE`

Maximum number for the size in bits of the type `long` on the target machine. If this is undefined, the default is `LONG_TYPE_SIZE`. Otherwise, it is the constant value that is the largest value that `LONG_TYPE_SIZE` can have at run-time. This is used in `cpp`.

`LONG_LONG_TYPE_SIZE`

A C expression for the size in bits of the type `long long` on the target machine. If you don't define this, the default is two words. If you want to support GNU Ada on your machine, the value of macro must be at least 64.

`CHAR_TYPE_SIZE`

A C expression for the size in bits of the type `char` on the target machine. If you don't define this, the default is one quarter of a word. (If this would be less than one storage unit, it is rounded up to one unit.)

`MAX_CHAR_TYPE_SIZE`

Maximum number for the size in bits of the type `char` on the target machine. If this is undefined, the default is `CHAR_TYPE_SIZE`. Otherwise, it is the constant value that is the largest value that `CHAR_TYPE_SIZE` can have at run-time. This is used in `cpp`.

`FLOAT_TYPE_SIZE`

A C expression for the size in bits of the type `float` on the target machine. If you don't define this, the default is one word.

`DOUBLE_TYPE_SIZE`

A C expression for the size in bits of the type `double` on the target machine. If you don't define this, the default is two words.

`LONG_DOUBLE_TYPE_SIZE`

A C expression for the size in bits of the type `long double` on the target machine. If you don't define this, the default is two words.

`DEFAULT_SIGNED_CHAR`

An expression whose value is 1 or 0, according to whether the type `char` should be signed or unsigned by default. The user can always override this default with the options `-fsigned-char' and `-funsigned-char'.

`DEFAULT_SHORT_ENUMS`

A C expression to determine whether to give an `enum` type only as many bytes as it takes to represent the range of possible values of that type. A nonzero value means to do that; a zero value means all `enum` types should be allocated like `int`.

If you don't define the macro, the default is 0.

`SIZE_TYPE`

A C expression for a string describing the name of the data type to use for size values. The typedef name `size_t` is defined using the contents of the string.

The string can contain more than one keyword. If so, separate them with spaces, and write first any length keyword, then `unsigned` if appropriate, and finally `int`. The string must exactly match one of the data type names defined in the function `init_decl_processing` in the file `` `c-decl.c' ``. You may not omit `int` or change the order--that would cause the compiler to crash on startup.

If you don't define this macro, the default is `"long unsigned int"`.

`PTRDIFF_TYPE`

A C expression for a string describing the name of the data type to use for the result of subtracting two pointers. The typedef name `ptrdiff_t` is defined using the contents of the string. See `SIZE_TYPE` above for more information.

If you don't define this macro, the default is `"long int"`.

`WCHAR_TYPE`

A C expression for a string describing the name of the data type to use for wide characters. The typedef name `wchar_t` is defined using the contents of the string. See `SIZE_TYPE` above for more information.

If you don't define this macro, the default is `"int"`.

`WCHAR_TYPE_SIZE`

A C expression for the size in bits of the data type for wide characters. This is used in `cpp`, which cannot make use of `WCHAR_TYPE`.

`MAX_WCHAR_TYPE_SIZE`

Maximum number for the size in bits of the data type for wide characters. If this is undefined, the default is `WCHAR_TYPE_SIZE`. Otherwise, it is the constant value that is the largest value that `WCHAR_TYPE_SIZE` can have at run-time. This is used in `cpp`.

`OBJC_INT_SELECTORS`

Define this macro if the type of Objective C selectors should be `int`.

If this macro is not defined, then selectors should have the type `struct objc_selector *`.

`OBJC_SELECTORS_WITHOUT_LABELS`

Define this macro if the compiler can group all the selectors together into a vector and use just one label at the beginning of the vector. Otherwise, the compiler must give each selector its own assembler label.

On certain machines, it is important to have a separate label for each selector because this enables the linker to eliminate duplicate selectors.

`TARGET_BELL`

A C constant expression for the integer value for escape sequence `\a'.

`TARGET_BS`

`TARGET_TAB`

`TARGET_NEWLINE`

C constant expressions for the integer values for escape sequences `\b', `\t' and `\n'.

`TARGET_VT`

`TARGET_FF`

`TARGET_CR`

C constant expressions for the integer values for escape sequences `\v', `\f' and `\r'.

Register Usage

This section explains how to describe what registers the target machine has, and how (in general) they can be used.

The description of which registers a specific instruction can use is done with register classes; see section Register Classes. For information on using registers to access a stack frame, see section Registers That Address the Stack Frame. For passing values in registers, see section Passing Arguments in Registers. For returning values in registers, see section How Scalar Function Values Are Returned.

Basic Characteristics of Registers

Registers have various characteristics.

`FIRST_PSEUDO_REGISTER`

Number of hardware registers known to the compiler. They receive numbers 0 through `FIRST_PSEUDO_REGISTER-1`; thus, the first pseudo register's number really is assigned the number `FIRST_PSEUDO_REGISTER`.

`FIXED_REGISTERS`

An initializer that says which registers are used for fixed purposes all throughout the compiled code and are therefore not available for general allocation. These would include the stack pointer, the frame pointer (except on machines where that can be used as a general register when no frame pointer is needed), the program counter on machines where that is considered one of the addressable registers, and any other numbered register with a standard use.

This information is expressed as a sequence of numbers, separated by commas and surrounded by braces. The nth number is 1 if register n is fixed, 0 otherwise.

The table initialized from this macro, and the table initialized by the following one, may be overridden at run time either automatically, by the actions of the macro `CONDITIONAL_REGISTER_USAGE`, or by the user with the command options `-ffixed-reg', `-fcall-used-reg' and `-fcall-saved-reg'.

`CALL_USED_REGISTERS`

Like `FIXED_REGISTERS` but has 1 for each register that is clobbered (in general) by function calls as well as for fixed registers. This macro therefore identifies the registers that are not

available for general allocation of values that must live across function calls.

If a register has 0 in `CALL_USED_REGISTERS`, the compiler automatically saves it on function entry and restores it on function exit, if the register is used within the function.

`CONDITIONAL_REGISTER_USAGE`

Zero or more C statements that may conditionally modify two variables `fixed_regs` and `call_used_regs` (both of type `char []`) after they have been initialized from the two preceding macros.

This is necessary in case the fixed or call-clobbered registers depend on target flags.

You need not define this macro if it has no work to do.

If the usage of an entire class of registers depends on the target flags, you may indicate this to GCC by using this macro to modify `fixed_regs` and `call_used_regs` to 1 for each of the registers in the classes which should not be used by GCC. Also define the macro `REG_CLASS_FROM_LETTER` to return `NO_REGS` if it is called with a letter for a class that shouldn't be used.

(However, if this class is not included in `GENERAL_REGS` and all of the insn patterns whose constraints permit this class are controlled by target switches, then GCC will automatically avoid using these registers when the target switches are opposed to them.)

`NON_SAVING_SETJMP`

If this macro is defined and has a nonzero value, it means that `setjmp` and related functions fail to save the registers, or that `longjmp` fails to restore them. To compensate, the compiler avoids putting variables in registers in functions that use `setjmp`.

`INCOMING_REGNO (out)`

Define this macro if the target machine has register windows. This C expression returns the register number as seen by the called function corresponding to the register number out as seen by the calling function. Return out if register number out is not an outbound register.

`OUTGOING_REGNO (in)`

Define this macro if the target machine has register windows. This C expression returns the register number as seen by the calling function corresponding to the register number in as seen by the called function. Return in if register number in is not an inbound register.

Order of Allocation of Registers

Registers are allocated in order.

`REG_ALLOC_ORDER`

If defined, an initializer for a vector of integers, containing the numbers of hard registers in the order in which GNU CC should prefer to use them (from most preferred to least).

If this macro is not defined, registers are used lowest numbered first (all else being equal).

One use of this macro is on machines where the highest numbered registers must always be saved

and the save-multiple-registers instruction supports only sequences of consecutive registers. O such machines, define `REG_ALLOC_ORDER` to be an initializer that lists the highest numbere allocatable register first.

`ORDER_REGS_FOR_LOCAL_ALLOC`

A C statement (sans semicolon) to choose the order in which to allocate hard registers for pseudo-registers local to a basic block.

Store the desired register order in the array `reg_alloc_order`. Element 0 should be the register to allocate first; element 1, the next register; and so on.

The macro body should not assume anything about the contents of `reg_alloc_order` before execution of the macro.

On most machines, it is not necessary to define this macro.

How Values Fit in Registers

This section discusses the macros that describe which kinds of values (specifically, which machine modes) each register can hold, and how many consecutive registers are needed for a given mode.

`HARD_REGNO_NREGS (regno, mode)`

A C expression for the number of consecutive hard registers, starting at register number regno, required to hold a value of mode mode.

On a machine where all registers are exactly one word, a suitable definition of this macro is

```
#define HARD_REGNO_NREGS(REGNO, MODE)            \
   ((GET_MODE_SIZE (MODE) + UNITS_PER_WORD - 1)  \
    / UNITS_PER_WORD))
```

`HARD_REGNO_MODE_OK (regno, mode)`

A C expression that is nonzero if it is permissible to store a value of mode mode in hard register number regno (or in several registers starting with that one). For a machine where all registers are equivalent, a suitable definition is

```
#define HARD_REGNO_MODE_OK(REGNO, MODE) 1
```

It is not necessary for this macro to check for the numbers of fixed registers, because the allocation mechanism considers them to be always occupied.

On some machines, double-precision values must be kept in even/odd register pairs. The way to implement that is to define this macro to reject odd register numbers for such modes.

The minimum requirement for a mode to be OK in a register is that the `movmode' instruction pattern support moves between the register and any other hard register for which the mode is OK; and that moving a value into the register and back out not alter it.

Since the same instruction used to move `SImode` will work for all narrower integer modes, it is

not necessary on any machine for `HARD_REGNO_MODE_OK` to distinguish between these mo provided you define patterns `movhi', etc., to take advantage of this. This is useful because of t interaction between `HARD_REGNO_MODE_OK` and `MODES_TIEABLE_P`; it is very desirable for all integer modes to be tieable.

Many machines have special registers for floating point arithmetic. Often people assume that floating point machine modes are allowed only in floating point registers. This is not true. Any registers that can hold integers can safely *hold* a floating point machine mode, whether or not floating arithmetic can be done on it in those registers. Integer move instructions can be used to move the values.

On some machines, though, the converse is true: fixed-point machine modes may not go in floating registers. This is true if the floating registers normalize any value stored in them, because storing a non-floating value there would garble it. In this case, `HARD_REGNO_MODE_OK` should reject fixed-point machine modes in floating registers. But if the floating registers do not automatically normalize, if you can store any bit pattern in one and retrieve it unchanged without a trap, then any machine mode may go in a floating register, so you can define this macro to say so.

The primary significance of special floating registers is rather that they are the registers acceptable in floating point arithmetic instructions. However, this is of no concern to `HARD_REGNO_MODE_OK`. You handle it by writing the proper constraints for those instructions.

On some machines, the floating registers are especially slow to access, so that it is better to store a value in a stack frame than in such a register if floating point arithmetic is not being done. As long as the floating registers are not in class `GENERAL_REGS`, they will not be used unless some pattern's constraint asks for one.

`MODES_TIEABLE_P (mode1, mode2)`

A C expression that is nonzero if it is desirable to choose register allocation so as to avoid move instructions between a value of mode mode1 and a value of mode mode2.

If `HARD_REGNO_MODE_OK (r, mode1)` and `HARD_REGNO_MODE_OK (r, mode2)` are ever different for any r, then `MODES_TIEABLE_P (mode1, mode2)` must be zero.

Handling Leaf Functions

On some machines, a leaf function (i.e., one which makes no calls) can run more efficiently if it does not make its own register window. Often this means it is required to receive its arguments in the registers where they are passed by the caller, instead of the registers where they would normally arrive.

The special treatment for leaf functions generally applies only when other conditions are met; for example, often they may use only those registers for its own variables and temporaries. We use the term "leaf function" to mean a function that is suitable for this special handling, so that functions with no calls are not necessarily "leaf functions".

GNU CC assigns register numbers before it knows whether the function is suitable for leaf function treatment. So it needs to renumber the registers in order to output a leaf function. The following macros accomplish this.

`LEAF_REGISTERS`

A C initializer for a vector, indexed by hard register number, which contains 1 for a register that is allowable in a candidate for leaf function treatment.

If leaf function treatment involves renumbering the registers, then the registers marked here should be the ones before renumbering--those that GNU CC would ordinarily allocate. The registers which will actually be used in the assembler code, after renumbering, should not be marked with 1 in this vector.

Define this macro only if the target machine offers a way to optimize the treatment of leaf functions.

`LEAF_REG_REMAP (regno)`

A C expression whose value is the register number to which regno should be renumbered, when a function is treated as a leaf function.

If regno is a register number which should not appear in a leaf function before renumbering, then the expression should yield -1, which will cause the compiler to abort.

Define this macro only if the target machine offers a way to optimize the treatment of leaf functions, and registers need to be renumbered to do this.

Normally, `FUNCTION_PROLOGUE` and `FUNCTION_EPILOGUE` must treat leaf functions specially. It can test the C variable `leaf_function` which is nonzero for leaf functions. (The variable `leaf_function` is defined only if `LEAF_REGISTERS` is defined.)

Registers That Form a Stack

There are special features to handle computers where some of the "registers" form a stack, as in the 80387 coprocessor for the 80386. Stack registers are normally written by pushing onto the stack, and are numbered relative to the top of the stack.

Currently, GNU CC can only handle one group of stack-like registers, and they must be consecutively numbered.

`STACK_REGS`

Define this if the machine has any stack-like registers.

`FIRST_STACK_REG`

The number of the first stack-like register. This one is the top of the stack.

`LAST_STACK_REG`

The number of the last stack-like register. This one is the bottom of the stack.

Obsolete Macros for Controlling Register Usage

These features do not work very well. They exist because they used to be required to generate correct code for the 80387 coprocessor of the 80386. They are no longer used by that machine description and may be removed in a later version of the compiler. Don't use them!

`OVERLAPPING_REGNO_P (regno)`

If defined, this is a C expression whose value is nonzero if hard register number regno is an overlapping register. This means a hard register which overlaps a hard register with a different number. (Such overlap is undesirable, but occasionally it allows a machine to be supported which otherwise could not be.) This macro must return nonzero for *all* the registers which overlap each other. GNU CC can use an overlapping register only in certain limited ways. It can be used for allocation within a basic block, and may be spilled for reloading; that is all.

If this macro is not defined, it means that none of the hard registers overlap each other. This is the usual situation.

`INSN_CLOBBERS_REGNO_P (insn, regno)`

If defined, this is a C expression whose value should be nonzero if the insn insn has the effect of mysteriously clobbering the contents of hard register number regno. By "mysterious" we mean that the insn's RTL expression doesn't describe such an effect.

If this macro is not defined, it means that no insn clobbers registers mysteriously. This is the usual situation; all else being equal, it is best for the RTL expression to show all the activity.

`PRESERVE_DEATH_INFO_REGNO_P (regno)`

If defined, this is a C expression whose value is nonzero if accurate `REG_DEAD` notes are needed for hard register number regno at the time of outputting the assembler code. When this is so, a few optimizations that take place after register allocation and could invalidate the death notes are not done when this register is involved.

You would arrange to preserve death info for a register when some of the code in the machine description which is executed to write the assembler code looks at the death notes. This is necessary only when the actual hardware feature which GNU CC thinks of as a register is not actually a register of the usual sort. (It might, for example, be a hardware stack.)

If this macro is not defined, it means that no death notes need to be preserved. This is the usual situation.

Register Classes

On many machines, the numbered registers are not all equivalent. For example, certain registers may not be allowed for indexed addressing; certain registers may not be allowed in some instructions. These machine restrictions are described to the compiler using register classes.

You define a number of register classes, giving each one a name and saying which of the registers belong to it. Then you can specify register classes that are allowed as operands to particular instruction patterns.

In general, each register will belong to several classes. In fact, one class must be named `ALL_REGS` and contain all the registers. Another class must be named `NO_REGS` and contain no registers. Often the union of two classes will be another class; however, this is not required.

One of the classes must be named `GENERAL_REGS`. There is nothing terribly special about the name, but the operand constraint letters `r' and `g' specify this class. If `GENERAL_REGS` is the same as

`ALL_REGS`, just define it as a macro which expands to `ALL_REGS`.

Order the classes so that if class x is contained in class y then x has a lower class number than y.

The way classes other than `GENERAL_REGS` are specified in operand constraints is through machine-dependent operand constraint letters. You can define such letters to correspond to various classes, then use them in operand constraints.

You should define a class for the union of two classes whenever some instruction allows both classes. For example, if an instruction allows either a floating point (coprocessor) register or a general register for a certain operand, you should define a class `FLOAT_OR_GENERAL_REGS` which includes both of them. Otherwise you will get suboptimal code.

You must also specify certain redundant information about the register classes: for each class, which classes contain it and which ones are contained in it; for each pair of classes, the largest class contained in their union.

When a value occupying several consecutive registers is expected in a certain class, all the registers used must belong to that class. Therefore, register classes cannot be used to enforce a requirement for a register pair to start with an even-numbered register. The way to specify this requirement is with `HARD_REGNO_MODE_OK`.

Register classes used for input-operands of bitwise-and or shift instructions have a special requirement: each such class must have, for each fixed-point machine mode, a subclass whose registers can transfer that mode to or from memory. For example, on some machines, the operations for single-byte values (`QImode`) are limited to certain registers. When this is so, each register class that is used in a bitwise-and or shift instruction must have a subclass consisting of registers from which single-byte values can be loaded or stored. This is so that `PREFERRED_RELOAD_CLASS` can always have a possible value to return.

`enum reg_class`

An enumeral type that must be defined with all the register class names as enumeral values. `NO_REGS` must be first. `ALL_REGS` must be the last register class, followed by one more enumeral value, `LIM_REG_CLASSES`, which is not a register class but rather tells how many classes there are.

Each register class has a number, which is the value of casting the class name to type `int`. The number serves as an index in many of the tables described below.

`N_REG_CLASSES`

The number of distinct register classes, defined as follows:

```
#define N_REG_CLASSES (int) LIM_REG_CLASSES
```

`REG_CLASS_NAMES`

An initializer containing the names of the register classes as C string constants. These names are used in writing some of the debugging dumps.

`REG_CLASS_CONTENTS`

An initializer containing the contents of the register classes, as integers which are bit masks. The

nth integer specifies the contents of class n. The way the integer mask is interpreted is that reg r is in the class if `mask & (1 << r)` is 1.

When the machine has more than 32 registers, an integer does not suffice. Then the integers are replaced by sub-initializers, braced groupings containing several integers. Each sub-initializer must be suitable as an initializer for the type `HARD_REG_SET` which is defined in `hard-reg-set.h`.

`REGNO_REG_CLASS (regno)`

A C expression whose value is a register class containing hard register regno. In general there is more than one such class; choose a class which is minimal, meaning that no smaller class also contains the register.

`BASE_REG_CLASS`

A macro whose definition is the name of the class to which a valid base register must belong. A base register is one used in an address which is the register value plus a displacement.

`INDEX_REG_CLASS`

A macro whose definition is the name of the class to which a valid index register must belong. An index register is one used in an address where its value is either multiplied by a scale factor or added to another register (as well as added to a displacement).

`REG_CLASS_FROM_LETTER (char)`

A C expression which defines the machine-dependent operand constraint letters for register classes. If char is such a letter, the value should be the register class corresponding to it. Otherwise, the value should be `NO_REGS`. The register letter `r', corresponding to class `GENERAL_REGS`, will not be passed to this macro; you do not need to handle it.

`REGNO_OK_FOR_BASE_P (num)`

A C expression which is nonzero if register number num is suitable for use as a base register in operand addresses. It may be either a suitable hard register or a pseudo register that has been allocated such a hard register.

`REGNO_OK_FOR_INDEX_P (num)`

A C expression which is nonzero if register number num is suitable for use as an index register in operand addresses. It may be either a suitable hard register or a pseudo register that has been allocated such a hard register.

The difference between an index register and a base register is that the index register may be scaled. If an address involves the sum of two registers, neither one of them scaled, then either one may be labeled the "base" and the other the "index"; but whichever labeling is used must fit the machine's constraints of which registers may serve in each capacity. The compiler will try both labelings, looking for one that is valid, and will reload one or both registers only if neither labeling works.

`PREFERRED_RELOAD_CLASS (x, class)`

A C expression that places additional restrictions on the register class to use when it is necessary to copy value x into a register in class class. The value is a register class; perhaps class, or perhaps another, smaller class. On many machines, the following definition is safe:


```
#define PREFERRED_RELOAD_CLASS(X,CLASS) CLASS
```

Sometimes returning a more restrictive class makes better code. For example, on the 68000, when x is an integer constant that is in range for a `moveq' instruction, the value of this macro is always `DATA_REGS` as long as class includes the data registers. Requiring a data register guarantees that a `moveq' will be used.

If x is a `const_double`, by returning `NO_REGS` you can force x into a memory constant. This is useful on certain machines where immediate floating values cannot be loaded into certain kinds of registers.

`PREFERRED_OUTPUT_RELOAD_CLASS (x, class)`

Like `PREFERRED_RELOAD_CLASS`, but for output reloads instead of input reloads. If you don't define this macro, the default is to use class, unchanged.

`LIMIT_RELOAD_CLASS (mode, class)`

A C expression that places additional restrictions on the register class to use when it is necessary to be able to hold a value of mode mode in a reload register for which class class would ordinarily be used.

Unlike `PREFERRED_RELOAD_CLASS`, this macro should be used when there are certain modes that simply can't go in certain reload classes.

The value is a register class; perhaps class, or perhaps another, smaller class.

Don't define this macro unless the target machine has limitations which require the macro to do something nontrivial.

`SECONDARY_RELOAD_CLASS (class, mode, x)`

`SECONDARY_INPUT_RELOAD_CLASS (class, mode, x)`

`SECONDARY_OUTPUT_RELOAD_CLASS (class, mode, x)`

Many machines have some registers that cannot be copied directly to or from memory or even from other types of registers. An example is the `MQ' register, which on most machines, can only be copied to or from general registers, but not memory. Some machines allow copying all registers to and from memory, but require a scratch register for stores to some memory locations (e.g., those with symbolic address on the RT, and those with certain symbolic address on the Sparc when compiling PIC). In some cases, both an intermediate and a scratch register are required.

You should define these macros to indicate to the reload phase that it may need to allocate at least one register for a reload in addition to the register to contain the data. Specifically, if copying x to a register class in mode requires an intermediate register, you should define `SECONDARY_INPUT_RELOAD_CLASS` to return the largest register class all of whose registers can be used as intermediate registers or scratch registers.

If copying a register class in mode to x requires an intermediate or scratch register, `SECONDARY_OUTPUT_RELOAD_CLASS` should be defined to return the largest register class required. If the requirements for input and output reloads are the same, the macro `SECONDARY_RELOAD_CLASS` should be used instead of defining both macros identically.

The values returned by these macros are often `GENERAL_REGS`. Return `NO_REGS` if no spare register is needed; i.e., if x can be directly copied to or from a register of class in mode without requiring a scratch register. Do not define this macro if it would always return `NO_REGS`.

If a scratch register is required (either with or without an intermediate register), you should define patterns for `reload_inm' or `reload_outm', as required (see section Standard Pattern Names For Generation. These patterns, which will normally be implemented with a `define_expand`, should be similar to the `movm' patterns, except that operand 2 is the scratch register.

Define constraints for the reload register and scratch register that contain a single register class. If the original reload register (whose class is class) can meet the constraint given in the pattern, the value returned by these macros is used for the class of the scratch register. Otherwise, two additional reload registers are required. Their classes are obtained from the constraints in the insn pattern.

x might be a pseudo-register or a `subreg` of a pseudo-register, which could either be in a hard register or in memory. Use `true_regnum` to find out; it will return -1 if the pseudo is in memory and the hard register number if it is in a register.

These macros should not be used in the case where a particular class of registers can only be copied to memory and not to another class of registers. In that case, secondary reload registers are not needed and would not be helpful. Instead, a stack location must be used to perform the copy and the `movm` pattern should use memory as a intermediate storage. This case often occurs between floating-point and general registers.

`SECONDARY_MEMORY_NEEDED (class1, class2, m)`

Certain machines have the property that some registers cannot be copied to some other registers without using memory. Define this macro on those machines to be a C expression that is non-zero if objects of mode m in registers of class1 can only be copied to registers of class class2 by storing a register of class1 into memory and loading that memory location into a register of class2.

Do not define this macro if its value would always be zero.

`SECONDARY_MEMORY_NEEDED_RTX (mode)`

Normally when `SECONDARY_MEMORY_NEEDED` is defined, the compiler allocates a stack slot for a memory location needed for register copies. If this macro is defined, the compiler instead uses the memory location defined by this macro.

Do not define this macro if you do not define `SECONDARY_MEMORY_NEEDED`.

`SECONDARY_MEMORY_NEEDED_MODE (mode)`

When the compiler needs a secondary memory location to copy between two registers of mode mode, it normally allocates sufficient memory to hold a quantity of `BITS_PER_WORD` bits and performs the store and load operations in a mode that many bits wide and whose class is the same as that of mode.

This is right thing to do on most machines because it ensures that all bits of the register are copied and prevents accesses to the registers in a narrower mode, which some machines prohibit for floating-point registers.

However, this default behavior is not correct on some machines, such as the DEC Alpha, that s short integers in floating-point registers differently than in integer registers. On those machines, the default widening will not work correctly and you must define this macro to suppress that widening in some cases. See the file `alpha.h' for details.

Do not define this macro if you do not define SECONDARY_MEMORY_NEEDED or if widening mode to a mode that is BITS_PER_WORD bits wide is correct for your machine.

SMALL_REGISTER_CLASSES

Normally the compiler avoids choosing registers that have been explicitly mentioned in the rtl as spill registers (these registers are normally those used to pass parameters and return values). However, some machines have so few registers of certain classes that there would not be enough registers to use as spill registers if this were done.

Define SMALL_REGISTER_CLASSES on these machines. When it is defined, the compiler allows registers explicitly used in the rtl to be used as spill registers but avoids extending the lifetime of these registers.

It is always safe to define this macro, but if you unnecessarily define it, you will reduce the amount of optimizations that can be performed in some cases. If you do not define this macro when it is required, the compiler will run out of spill registers and print a fatal error message. For most machines, you should not define this macro.

CLASS_LIKELY_SPILLED_P (class)

A C expression whose value is nonzero if pseudos that have been assigned to registers of class class would likely be spilled because registers of class are needed for spill registers.

The default value of this macro returns 1 if class has exactly one register and zero otherwise. On most machines, this default should be used. Only define this macro to some other expression if pseudo allocated by `local-alloc.c' end up in memory because their hard registers were needed for spill registers. If this macro returns nonzero for those classes, those pseudos will only be allocated by `global.c', which knows how to reallocate the pseudo to another register. If there would not be another register available for reallocation, you should not change the definition of this macro since the only effect of such a definition would be to slow down register allocation.

CLASS_MAX_NREGS (class, mode)

A C expression for the maximum number of consecutive registers of class class needed to hold a value of mode mode.

This is closely related to the macro HARD_REGNO_NREGS. In fact, the value of the macro CLASS_MAX_NREGS (class, mode) should be the maximum value of HARD_REGNO_NREGS (regno, mode) for all regno values in the class class.

This macro helps control the handling of multiple-word values in the reload pass.

CLASS_CANNOT_CHANGE_SIZE

If defined, a C expression for a class that contains registers which the compiler must always access in a mode that is the same size as the mode in which it loaded the register.

For the example, loading 32-bit integer or floating-point objects into floating-point registers on the

Alpha extends them to 64-bits. Therefore loading a 64-bit object and then storing it as a 32-bit object does not store the low-order 32-bits, as would be the case for a normal register. Therefo `alpha.h' defines this macro as `FLOAT_REGS`.

Three other special macros describe which operands fit which constraint letters.

`CONST_OK_FOR_LETTER_P (value, c)`

A C expression that defines the machine-dependent operand constraint letters that specify particular ranges of integer values. If c is one of those letters, the expression should check that value, an integer, is in the appropriate range and return 1 if so, 0 otherwise. If c is not one of those letters, the value should be 0 regardless of value.

`CONST_DOUBLE_OK_FOR_LETTER_P (value, c)`

A C expression that defines the machine-dependent operand constraint letters that specify particular ranges of `const_double` values.

If c is one of those letters, the expression should check that value, an RTX of code `const_double`, is in the appropriate range and return 1 if so, 0 otherwise. If c is not one of those letters, the value should be 0 regardless of value.

`const_double` is used for all floating-point constants and for `DImode` fixed-point constants. A given letter can accept either or both kinds of values. It can use `GET_MODE` to distinguish between these kinds.

`EXTRA_CONSTRAINT (value, c)`

A C expression that defines the optional machine-dependent constraint letters that can be used to segregate specific types of operands, usually memory references, for the target machine. Normally this macro will not be defined. If it is required for a particular target machine, it should return 1 if value corresponds to the operand type represented by the constraint letter c. If c is not defined as an extra constraint, the value returned should be 0 regardless of value.

For example, on the ROMP, load instructions cannot have their output in r0 if the memory reference contains a symbolic address. Constraint letter `Q' is defined as representing a memory address that does *not* contain a symbolic address. An alternative is specified with a `Q' constraint on the input and `r' on the output. The next alternative specifies `m' on the input and a register class that does not include r0 on the output.

Stack Layout and Calling Conventions

This describes the stack layout and calling conventions.

Basic Stack Layout

Here is the basic stack layout.

`STACK_GROWS_DOWNWARD`

Define this macro if pushing a word onto the stack moves the stack pointer to a smaller address.

When we say, "define this macro if ...," it means that the compiler checks this macro only with `#ifdef` so the precise definition used does not matter.

`FRAME_GROWS_DOWNWARD`

Define this macro if the addresses of local variable slots are at negative offsets from the frame pointer.

`ARGS_GROW_DOWNWARD`

Define this macro if successive arguments to a function occupy decreasing addresses on the stack.

`STARTING_FRAME_OFFSET`

Offset from the frame pointer to the first local variable slot to be allocated.

If `FRAME_GROWS_DOWNWARD`, find the next slot's offset by subtracting the first slot's length from `STARTING_FRAME_OFFSET`. Otherwise, it is found by adding the length of the first slot to the value `STARTING_FRAME_OFFSET`.

`STACK_POINTER_OFFSET`

Offset from the stack pointer register to the first location at which outgoing arguments are placed. If not specified, the default value of zero is used. This is the proper value for most machines.

If `ARGS_GROW_DOWNWARD`, this is the offset to the location above the first location at which outgoing arguments are placed.

`FIRST_PARM_OFFSET (fundecl)`

Offset from the argument pointer register to the first argument's address. On some machines it may depend on the data type of the function.

If `ARGS_GROW_DOWNWARD`, this is the offset to the location above the first argument's address.

`STACK_DYNAMIC_OFFSET (fundecl)`

Offset from the stack pointer register to an item dynamically allocated on the stack, e.g., by `alloca`.

The default value for this macro is `STACK_POINTER_OFFSET` plus the length of the outgoing arguments. The default is correct for most machines. See `` `function.c' `` for details.

`DYNAMIC_CHAIN_ADDRESS (frameaddr)`

A C expression whose value is RTL representing the address in a stack frame where the pointer to the caller's frame is stored. Assume that frameaddr is an RTL expression for the address of the stack frame itself.

If you don't define this macro, the default is to return the value of frameaddr---that is, the stack frame address is also the address of the stack word that points to the previous frame.

`SETUP_FRAME_ADDRESSES ()`

If defined, a C expression that produces the machine-specific code to setup the stack so that arbitrary frames can be accessed. For example, on the Sparc, we must flush all of the register windows to the stack before we can access arbitrary stack frames. This macro will seldom need to be defined.

`RETURN_ADDR_RTX (count, frameaddr)`

A C expression whose value is RTL representing the value of the return address for the frame count steps up from the current frame. frameaddr is the frame pointer of the count frame, or the frame pointer of the count - 1 frame if `RETURN_ADDR_IN_PREVIOUS_FRAME` is defined.

`RETURN_ADDR_IN_PREVIOUS_FRAME`

Define this if the return address of a particular stack frame is accessed from the frame pointer of the previous stack frame.

Registers That Address the Stack Frame

This discusses registers that address the stack frame.

`STACK_POINTER_REGNUM`

The register number of the stack pointer register, which must also be a fixed register according to `FIXED_REGISTERS`. On most machines, the hardware determines which register this is.

`FRAME_POINTER_REGNUM`

The register number of the frame pointer register, which is used to access automatic variables in the stack frame. On some machines, the hardware determines which register this is. On other machines, you can choose any register you wish for this purpose.

`HARD_FRAME_POINTER_REGNUM`

On some machines the offset between the frame pointer and starting offset of the automatic variables is not known until after register allocation has been done (for example, because the saved registers are between these two locations). On those machines, define `FRAME_POINTER_REGNUM` the number of a special, fixed register to be used internally until the offset is known, and define `HARD_FRAME_POINTER_REGNUM` to be actual the hard register number used for the frame pointer.

You should define this macro only in the very rare circumstances when it is not possible to calculate the offset between the frame pointer and the automatic variables until after register allocation has been completed. When this macro is defined, you must also indicate in your definition of `ELIMINABLE_REGS` how to eliminate `FRAME_POINTER_REGNUM` into either `HARD_FRAME_POINTER_REGNUM` or `STACK_POINTER_REGNUM`.

Do not define this macro if it would be the same as `FRAME_POINTER_REGNUM`.

`ARG_POINTER_REGNUM`

The register number of the arg pointer register, which is used to access the function's argument list. On some machines, this is the same as the frame pointer register. On some machines, the hardware determines which register this is. On other machines, you can choose any register you wish for this purpose. If this is not the same register as the frame pointer register, then you must mark it as a fixed register according to `FIXED_REGISTERS`, or arrange to be able to eliminate it (see section Eliminating Frame Pointer and Arg Pointer).

`STATIC_CHAIN_REGNUM`

`STATIC_CHAIN_INCOMING_REGNUM`

Register numbers used for passing a function's static chain pointer. If register windows are used, the register number as seen by the called function is `STATIC_CHAIN_INCOMING_REGNUM`,

while the register number as seen by the calling function is `STATIC_CHAIN_REGNUM`. If the registers are the same, `STATIC_CHAIN_INCOMING_REGNUM` need not be defined.

The static chain register need not be a fixed register.

If the static chain is passed in memory, these macros should not be defined; instead, the next two macros should be defined.

`STATIC_CHAIN`

`STATIC_CHAIN_INCOMING`

If the static chain is passed in memory, these macros provide rtx giving `mem` expressions that denote where they are stored. `STATIC_CHAIN` and `STATIC_CHAIN_INCOMING` give the locations as seen by the calling and called functions, respectively. Often the former will be at an offset from the stack pointer and the latter at an offset from the frame pointer.

The variables `stack_pointer_rtx`, `frame_pointer_rtx`, and `arg_pointer_rtx` will have been initialized prior to the use of these macros and should be used to refer to those items.

If the static chain is passed in a register, the two previous macros should be defined instead.

Eliminating Frame Pointer and Arg Pointer

This is about eliminating the frame pointer and arg pointer.

`FRAME_POINTER_REQUIRED`

A C expression which is nonzero if a function must have and use a frame pointer. This expression is evaluated in the reload pass. If its value is nonzero the function will have a frame pointer.

The expression can in principle examine the current function and decide according to the facts, but on most machines the constant 0 or the constant 1 suffices. Use 0 when the machine allows code to be generated with no frame pointer, and doing so saves some time or space. Use 1 when there is no possible advantage to avoiding a frame pointer.

In certain cases, the compiler does not know how to produce valid code without a frame pointer. The compiler recognizes those cases and automatically gives the function a frame pointer regardless of what `FRAME_POINTER_REQUIRED` says. You don't need to worry about them.

In a function that does not require a frame pointer, the frame pointer register can be allocated for ordinary usage, unless you mark it as a fixed register. See `FIXED_REGISTERS` for more information.

`INITIAL_FRAME_POINTER_OFFSET (depth-var)`

A C statement to store in the variable depth-var the difference between the frame pointer and the stack pointer values immediately after the function prologue. The value would be computed from information such as the result of `get_frame_size ()` and the tables of registers `regs_ever_live` and `call_used_regs`.

If `ELIMINABLE_REGS` is defined, this macro will be not be used and need not be defined. Otherwise, it must be defined even if `FRAME_POINTER_REQUIRED` is defined to always be true; in that case, you may set depth-var to anything.

`ELIMINABLE_REGS`

If defined, this macro specifies a table of register pairs used to eliminate unneeded registers that point into the stack frame. If it is not defined, the only elimination attempted by the compiler is to replace references to the frame pointer with references to the stack pointer.

The definition of this macro is a list of structure initializations, each of which specifies an original and replacement register.

On some machines, the position of the argument pointer is not known until the compilation is completed. In such a case, a separate hard register must be used for the argument pointer. This register can be eliminated by replacing it with either the frame pointer or the argument pointer, depending on whether or not the frame pointer has been eliminated.

In this case, you might specify:

```
#define ELIMINABLE_REGS  \
{{ARG_POINTER_REGNUM, STACK_POINTER_REGNUM}, \
 {ARG_POINTER_REGNUM, FRAME_POINTER_REGNUM}, \
 {FRAME_POINTER_REGNUM, STACK_POINTER_REGNUM}}
```

Note that the elimination of the argument pointer with the stack pointer is specified first since that is the preferred elimination.

`CAN_ELIMINATE (from-reg, to-reg)`

A C expression that returns non-zero if the compiler is allowed to try to replace register number from-reg with register number to-reg. This macro need only be defined if `ELIMINABLE_REGS` is defined, and will usually be the constant 1, since most of the cases preventing register elimination are things that the compiler already knows about.

`INITIAL_ELIMINATION_OFFSET (from-reg, to-reg, offset-var)`

This macro is similar to `INITIAL_FRAME_POINTER_OFFSET`. It specifies the initial difference between the specified pair of registers. This macro must be defined if `ELIMINABLE_REGS` is defined.

`LONGJMP_RESTORE_FROM_STACK`

Define this macro if the `longjmp` function restores registers from the stack frames, rather than from those saved specifically by `setjmp`. Certain quantities must not be kept in registers across a call to `setjmp` on such machines.

Passing Function Arguments on the Stack

The macros in this section control how arguments are passed on the stack. See the following section for other macros that control passing certain arguments in registers.

`PROMOTE_PROTOTYPES`

Define this macro if an argument declared in a prototype as an integral type smaller than `int` should actually be passed as an `int`. In addition to avoiding errors in certain cases of mismatch, it also makes for better code on certain machines.

`PUSH_ROUNDING (npushed)`

A C expression that is the number of bytes actually pushed onto the stack when an instruction attempts to push npushed bytes.

If the target machine does not have a push instruction, do not define this macro. That directs GNU CC to use an alternate strategy: to allocate the entire argument block and then store the arguments into it.

On some machines, the definition

```
#define PUSH_ROUNDING(BYTES) (BYTES)
```

will suffice. But on other machines, instructions that appear to push one byte actually push two bytes in an attempt to maintain alignment. Then the definition should be

```
#define PUSH_ROUNDING(BYTES) (((BYTES) + 1) & ~1)
```

`ACCUMULATE_OUTGOING_ARGS`

If defined, the maximum amount of space required for outgoing arguments will be computed and placed into the variable `current_function_outgoing_args_size`. No space will be pushed onto the stack for each call; instead, the function prologue should increase the stack frame size by this amount.

Defining both `PUSH_ROUNDING` and `ACCUMULATE_OUTGOING_ARGS` is not proper.

`REG_PARM_STACK_SPACE (fndecl)`

Define this macro if functions should assume that stack space has been allocated for arguments even when their values are passed in registers.

The value of this macro is the size, in bytes, of the area reserved for arguments passed in registers for the function represented by fndecl.

This space can be allocated by the caller, or be a part of the machine-dependent stack frame: `OUTGOING_REG_PARM_STACK_SPACE` says which.

`MAYBE_REG_PARM_STACK_SPACE`

`FINAL_REG_PARM_STACK_SPACE (const_size, var_size)`

Define these macros in addition to the one above if functions might allocate stack space for arguments even when their values are passed in registers. These should be used when the stack space allocated for arguments in registers is not a simple constant independent of the function declaration.

The value of the first macro is the size, in bytes, of the area that we should initially assume would be reserved for arguments passed in registers.

The value of the second macro is the actual size, in bytes, of the area that will be reserved for arguments passed in registers. This takes two arguments: an integer representing the number of bytes of fixed sized arguments on the stack, and a tree representing the number of bytes of variable sized arguments on the stack.

When these macros are defined, `REG_PARM_STACK_SPACE` will only be called for libcall functions, the current function, or for a function being called when it is known that such stack space must be allocated. In each case this value can be easily computed.

When deciding whether a called function needs such stack space, and how much space to reserve, GNU CC uses these two macros instead of `REG_PARM_STACK_SPACE`.

`OUTGOING_REG_PARM_STACK_SPACE`

Define this if it is the responsibility of the caller to allocate the area reserved for arguments passed in registers.

If `ACCUMULATE_OUTGOING_ARGS` is defined, this macro controls whether the space for these arguments counts in the value of `current_function_outgoing_args_size`.

`STACK_PARMS_IN_REG_PARM_AREA`

Define this macro if `REG_PARM_STACK_SPACE` is defined, but the stack parameters don't skip the area specified by it.

Normally, when a parameter is not passed in registers, it is placed on the stack beyond the `REG_PARM_STACK_SPACE` area. Defining this macro suppresses this behavior and causes the parameter to be passed on the stack in its natural location.

`RETURN_POPS_ARGS (fundecl, funtype, stack-size)`

A C expression that should indicate the number of bytes of its own arguments that a function pops on returning, or 0 if the function pops no arguments and the caller must therefore pop them all after the function returns.

fundecl is a C variable whose value is a tree node that describes the function in question. Normally it is a node of type `FUNCTION_DECL` that describes the declaration of the function. From this it is possible to obtain the DECL_MACHINE_ATTRIBUTES of the function.

funtype is a C variable whose value is a tree node that describes the function in question. Normally it is a node of type `FUNCTION_TYPE` that describes the data type of the function. From this it is possible to obtain the data types of the value and arguments (if known).

When a call to a library function is being considered, funtype will contain an identifier node for the library function. Thus, if you need to distinguish among various library functions, you can do so by their names. Note that "library function" in this context means a function used to perform arithmetic, whose name is known specially in the compiler and was not mentioned in the C code being compiled.

stack-size is the number of bytes of arguments passed on the stack. If a variable number of bytes is passed, it is zero, and argument popping will always be the responsibility of the calling function.

On the Vax, all functions always pop their arguments, so the definition of this macro is stack-size. On the 68000, using the standard calling convention, no functions pop their arguments, so the value of the macro is always 0 in this case. But an alternative calling convention is available in which functions that take a fixed number of arguments pop them but other functions (such as `printf`) pop nothing (the caller pops all). When this convention is in use, funtype is examined to determine whether a function takes a fixed number of arguments.

Passing Arguments in Registers

This section describes the macros which let you control how various types of arguments are passed in registers or how they are arranged in the stack.

`FUNCTION_ARG (cum, mode, type, named)`

A C expression that controls whether a function argument is passed in a register, and which register.

The arguments are cum, which summarizes all the previous arguments; mode, the machine mode of the argument; type, the data type of the argument as a tree node or 0 if that is not known (which happens for C support library functions); and named, which is 1 for an ordinary argument and 0 for nameless arguments that correspond to `...' in the called function's prototype.

The value of the expression should either be a `reg` RTX for the hard register in which to pass the argument, or zero to pass the argument on the stack.

For machines like the Vax and 68000, where normally all arguments are pushed, zero suffices as a definition.

The usual way to make the ANSI library `` `stdarg.h' `` work on a machine where some arguments are usually passed in registers, is to cause nameless arguments to be passed on the stack instead. This is done by making `FUNCTION_ARG` return 0 whenever named is 0.

You may use the macro `MUST_PASS_IN_STACK (mode, type)` in the definition of this macro to determine if this argument is of a type that must be passed in the stack. If `REG_PARM_STACK_SPACE` is not defined and `FUNCTION_ARG` returns non-zero for such an argument, the compiler will abort. If `REG_PARM_STACK_SPACE` is defined, the argument will be computed in the stack and then loaded into a register.

`FUNCTION_INCOMING_ARG (cum, mode, type, named)`

Define this macro if the target machine has "register windows", so that the register in which a function sees an arguments is not necessarily the same as the one in which the caller passed the argument.

For such machines, `FUNCTION_ARG` computes the register in which the caller passes the value, and `FUNCTION_INCOMING_ARG` should be defined in a similar fashion to tell the function being called where the arguments will arrive.

If `FUNCTION_INCOMING_ARG` is not defined, `FUNCTION_ARG` serves both purposes.

`FUNCTION_ARG_PARTIAL_NREGS (cum, mode, type, named)`

A C expression for the number of words, at the beginning of an argument, must be put in registers. The value must be zero for arguments that are passed entirely in registers or that are entirely pushed on the stack.

On some machines, certain arguments must be passed partially in registers and partially in memory. On these machines, typically the first n words of arguments are passed in registers, and the rest on the stack. If a multi-word argument (a `double` or a structure) crosses that boundary, its

first few words must be passed in registers and the rest must be pushed. This macro tells the compiler when this occurs, and how many of the words should go in registers.

`FUNCTION_ARG` for these arguments should return the first register to be used by the caller for this argument; likewise `FUNCTION_INCOMING_ARG`, for the called function.

`FUNCTION_ARG_PASS_BY_REFERENCE (cum, mode, type, named)`

A C expression that indicates when an argument must be passed by reference. If nonzero for an argument, a copy of that argument is made in memory and a pointer to the argument is passed instead of the argument itself. The pointer is passed in whatever way is appropriate for passing a pointer to that type.

On machines where `REG_PARM_STACK_SPACE` is not defined, a suitable definition of this macro might be

```
#define FUNCTION_ARG_PASS_BY_REFERENCE\
(CUM, MODE, TYPE, NAMED)  \
  MUST_PASS_IN_STACK (MODE, TYPE)
```

`FUNCTION_ARG_CALLEE_COPIES (cum, mode, type, named)`

If defined, a C expression that indicates when it is the called function's responsibility to make a copy of arguments passed by invisible reference. Normally, the caller makes a copy and passes the address of the copy to the routine being called. When FUNCTION_ARG_CALLEE_COPIES is defined and is nonzero, the caller does not make a copy. Instead, it passes a pointer to the "live" value. The called function must not modify this value. If it can be determined that the value won't be modified, it need not make a copy; otherwise a copy must be made.

`CUMULATIVE_ARGS`

A C type for declaring a variable that is used as the first argument of `FUNCTION_ARG` and other related values. For some target machines, the type `int` suffices and can hold the number of bytes of argument so far.

There is no need to record in `CUMULATIVE_ARGS` anything about the arguments that have been passed on the stack. The compiler has other variables to keep track of that. For target machines on which all arguments are passed on the stack, there is no need to store anything in `CUMULATIVE_ARGS`; however, the data structure must exist and should not be empty, so use `int`.

`INIT_CUMULATIVE_ARGS (cum, fntype, libname)`

A C statement (sans semicolon) for initializing the variable cum for the state at the beginning of the argument list. The variable has type `CUMULATIVE_ARGS`. The value of fntype is the tree node for the data type of the function which will receive the args, or 0 if the args are to a compiler support library function.

When processing a call to a compiler support library function, libname identifies which one. It is a `symbol_ref` rtx which contains the name of the function, as a string. libname is 0 when an ordinary C function call is being processed. Thus, each time this macro is called, either libname or fntype is nonzero, but never both of them at once.

`INIT_CUMULATIVE_INCOMING_ARGS (cum, fntype, libname)`

Like `INIT_CUMULATIVE_ARGS` but overrides it for the purposes of finding the arguments for the function being compiled. If this macro is undefined, `INIT_CUMULATIVE_ARGS` is used instead.

The value passed for libname is always 0, since library routines with special calling conventions are never compiled with GNU CC. The argument libname exists for symmetry with `INIT_CUMULATIVE_ARGS`.

`FUNCTION_ARG_ADVANCE (cum, mode, type, named)`

A C statement (sans semicolon) to update the summarizer variable cum to advance past an argument in the argument list. The values mode, type and named describe that argument. Once this is done, the variable cum is suitable for analyzing the *following* argument with `FUNCTION_ARG`, etc.

This macro need not do anything if the argument in question was passed on the stack. The compiler knows how to track the amount of stack space used for arguments without any special help.

`FUNCTION_ARG_PADDING (mode, type)`

If defined, a C expression which determines whether, and in which direction, to pad out an argument with extra space. The value should be of type `enum direction`: either `upward` to pad above the argument, `downward` to pad below, or `none` to inhibit padding.

The *amount* of padding is always just enough to reach the next multiple of `FUNCTION_ARG_BOUNDARY`; this macro does not control it.

This macro has a default definition which is right for most systems. For little-endian machines, the default is to pad upward. For big-endian machines, the default is to pad downward for an argument of constant size shorter than an `int`, and upward otherwise.

`FUNCTION_ARG_BOUNDARY (mode, type)`

If defined, a C expression that gives the alignment boundary, in bits, of an argument with the specified mode and type. If it is not defined, `PARM_BOUNDARY` is used for all arguments.

`FUNCTION_ARG_REGNO_P (regno)`

A C expression that is nonzero if regno is the number of a hard register in which function arguments are sometimes passed. This does *not* include implicit arguments such as the static chain and the structure-value address. On many machines, no registers can be used for this purpose since all function arguments are pushed on the stack.

How Scalar Function Values Are Returned

This section discusses the macros that control returning scalars as values--values that can fit in registers.

`TRADITIONAL_RETURN_FLOAT`

Define this macro if `-traditional' should not cause functions declared to return `float` to convert the value to `double`.

`FUNCTION_VALUE (valtype, func)`

A C expression to create an RTX representing the place where a function returns a value of da type valtype. valtype is a tree node representing a data type. Write `TYPE_MODE (valtype)` get the machine mode used to represent that type. On many machines, only the mode is relevant. (Actually, on most machines, scalar values are returned in the same place regardless of mode).

If `PROMOTE_FUNCTION_RETURN` is defined, you must apply the same promotion rules specified in `PROMOTE_MODE` if valtype is a scalar type.

If the precise function being called is known, func is a tree node (`FUNCTION_DECL`) for it; otherwise, func is a null pointer. This makes it possible to use a different value-returning convention for specific functions when all their calls are known.

`FUNCTION_VALUE` is not used for return vales with aggregate data types, because these are returned in another way. See `STRUCT_VALUE_REGNUM` and related macros, below.

`FUNCTION_OUTGOING_VALUE (valtype, func)`

Define this macro if the target machine has "register windows" so that the register in which a function returns its value is not the same as the one in which the caller sees the value.

For such machines, `FUNCTION_VALUE` computes the register in which the caller will see the value. `FUNCTION_OUTGOING_VALUE` should be defined in a similar fashion to tell the function where to put the value.

If `FUNCTION_OUTGOING_VALUE` is not defined, `FUNCTION_VALUE` serves both purposes.

`FUNCTION_OUTGOING_VALUE` is not used for return vales with aggregate data types, because these are returned in another way. See `STRUCT_VALUE_REGNUM` and related macros, below.

`LIBCALL_VALUE (mode)`

A C expression to create an RTX representing the place where a library function returns a value of mode mode. If the precise function being called is known, func is a tree node (`FUNCTION_DECL`) for it; otherwise, func is a null pointer. This makes it possible to use a different value-returning convention for specific functions when all their calls are known.

Note that "library function" in this context means a compiler support routine, used to perform arithmetic, whose name is known specially by the compiler and was not mentioned in the C code being compiled.

The definition of `LIBRARY_VALUE` need not be concerned aggregate data types, because none of the library functions returns such types.

`FUNCTION_VALUE_REGNO_P (regno)`

A C expression that is nonzero if regno is the number of a hard register in which the values of called function may come back.

A register whose use for returning values is limited to serving as the second of a pair (for a value of type `double`, say) need not be recognized by this macro. So for most machines, this definition suffices:

```
#define FUNCTION_VALUE_REGNO_P(N) ((N) == 0)
```

If the machine has register windows, so that the caller and the called function use different registers for the return value, this macro should recognize only the caller's register numbers.

`APPLY_RESULT_SIZE`

Define this macro if `untyped_call' and `untyped_return' need more space than is implied by `FUNCTION_VALUE_REGNO_P` for saving and restoring an arbitrary return value.

How Large Values Are Returned

When a function value's mode is `BLKmode` (and in some other cases), the value is not returned according to `FUNCTION_VALUE` (see section How Scalar Function Values Are Returned). Instead, the caller passes the address of a block of memory in which the value should be stored. This address is called the structure value address.

This section describes how to control returning structure values in memory.

`RETURN_IN_MEMORY (type)`

A C expression which can inhibit the returning of certain function values in registers, based on the type of value. A nonzero value says to return the function value in memory, just as large structures are always returned. Here type will be a C expression of type `tree`, representing the data type of the value.

Note that values of mode `BLKmode` must be explicitly handled by this macro. Also, the option `-fpcc-struct-return' takes effect regardless of this macro. On most systems, it is possible to leave the macro undefined; this causes a default definition to be used, whose value is the constant 1 for `BLKmode` values, and 0 otherwise.

Do not use this macro to indicate that structures and unions should always be returned in memory. You should instead use `DEFAULT_PCC_STRUCT_RETURN` to indicate this.

`DEFAULT_PCC_STRUCT_RETURN`

Define this macro to be 1 if all structure and union return values must be in memory. Since this results in slower code, this should be defined only if needed for compatibility with other compilers or with an ABI. If you define this macro to be 0, then the conventions used for structure and union return values are decided by the `RETURN_IN_MEMORY` macro.

If not defined, this defaults to the value 1.

`STRUCT_VALUE_REGNUM`

If the structure value address is passed in a register, then `STRUCT_VALUE_REGNUM` should be the number of that register.

`STRUCT_VALUE`

If the structure value address is not passed in a register, define `STRUCT_VALUE` as an expression returning an RTX for the place where the address is passed. If it returns 0, the address is passed as an "invisible" first argument.

`STRUCT_VALUE_INCOMING_REGNUM`

On some architectures the place where the structure value address is found by the called function is

not the same place that the caller put it. This can be due to register windows, or it could be bec the function prologue moves it to a different place.

If the incoming location of the structure value address is in a register, define this macro as the register number.

`STRUCT_VALUE_INCOMING`

If the incoming location is not a register, then you should define `STRUCT_VALUE_INCOMING` as an expression for an RTX for where the called function should find the value. If it should find the value on the stack, define this to create a `mem` which refers to the frame pointer. A definition of 0 means that the address is passed as an "invisible" first argument.

`PCC_STATIC_STRUCT_RETURN`

Define this macro if the usual system convention on the target machine for returning structures and unions is for the called function to return the address of a static variable containing the value.

Do not define this if the usual system convention is for the caller to pass an address to the subroutine.

This macro has effect in `-fpcc-struct-return' mode, but it does nothing when you use `-freg-struct-return' mode.

Caller-Saves Register Allocation

If you enable it, GNU CC can save registers around function calls. This makes it possible to use call-clobbered registers to hold variables that must live across calls.

`DEFAULT_CALLER_SAVES`

Define this macro if function calls on the target machine do not preserve any registers; in other words, if `CALL_USED_REGISTERS` has 1 for all registers. This macro enables `-fcaller-saves' by default. Eventually that option will be enabled by default on all machines and both the option and this macro will be eliminated.

`CALLER_SAVE_PROFITABLE (refs, calls)`

A C expression to determine whether it is worthwhile to consider placing a pseudo-register in a call-clobbered hard register and saving and restoring it around each function call. The expression should be 1 when this is worth doing, and 0 otherwise.

If you don't define this macro, a default is used which is good on most machines: `4 * calls < refs`.

Function Entry and Exit

This section describes the macros that output function entry (prologue) and exit (epilogue) code.

`FUNCTION_PROLOGUE (file, size)`

A C compound statement that outputs the assembler code for entry to a function. The prologue is responsible for setting up the stack frame, initializing the frame pointer register, saving registers that must be saved, and allocating size additional bytes of storage for the local variables. size is an

integer. file is a stdio stream to which the assembler code should be output.

The label for the beginning of the function need not be output by this macro. That has already been done when the macro is run.

To determine which registers to save, the macro can refer to the array `regs_ever_live`: element r is nonzero if hard register r is used anywhere within the function. This implies the function prologue should save register r, provided it is not one of the call-used registers. (`FUNCTION_EPILOGUE` must likewise use `regs_ever_live`.)

On machines that have "register windows", the function entry code does not save on the stack the registers that are in the windows, even if they are supposed to be preserved by function calls; instead it takes appropriate steps to "push" the register stack, if any non-call-used registers are used in the function.

On machines where functions may or may not have frame-pointers, the function entry code must vary accordingly; it must set up the frame pointer if one is wanted, and not otherwise. To determine whether a frame pointer is in wanted, the macro can refer to the variable `frame_pointer_needed`. The variable's value will be 1 at run time in a function that needs a frame pointer. See section Eliminating Frame Pointer and Arg Pointer.

The function entry code is responsible for allocating any stack space required for the function. This stack space consists of the regions listed below. In most cases, these regions are allocated in the order listed, with the last listed region closest to the top of the stack (the lowest address if `STACK_GROWS_DOWNWARD` is defined, and the highest address if it is not defined). You can use a different order for a machine if doing so is more convenient or required for compatibility reasons. Except in cases where required by standard or by a debugger, there is no reason why the stack layout used by GCC need agree with that used by other compilers for a machine.

A region of `current_function_pretend_args_size` bytes of uninitialized space just underneath the first argument arriving on the stack. (This may not be at the very start of the allocated stack region if the calling sequence has pushed anything else since pushing the stack arguments. But usually, on such machines, nothing else has been pushed yet, because the function prologue itself does all the pushing.) This region is used on machines where an argument may be passed partly in registers and partly in memory, and, in some cases to support the features in `` `varargs.h' `` and `` `stdargs.h' ``.

An area of memory used to save certain registers used by the function. The size of this area, which may also include space for such things as the return address and pointers to previous stack frames, is machine-specific and usually depends on which registers have been used in the function. Machines with register windows often do not require a save area.

A region of at least size bytes, possibly rounded up to an allocation boundary, to contain the local variables of the function. On some machines, this region and the save area may occur in the opposite order, with the save area closer to the top of the stack.

Optionally, when `ACCUMULATE_OUTGOING_ARGS` is defined, a region of `current_function_outgoing_args_size` bytes to be used for outgoing argument lists of the function. See section Passing Function Arguments on the Stack.

Normally, it is necessary for the macros `FUNCTION_PROLOGUE` and `FUNCTION_EPILOGUE` treat leaf functions specially. The C variable `leaf_function` is nonzero for such a function.

`EXIT_IGNORE_STACK`

Define this macro as a C expression that is nonzero if the return instruction or the function epilogue ignores the value of the stack pointer; in other words, if it is safe to delete an instruction to adjust the stack pointer before a return from the function.

Note that this macro's value is relevant only for functions for which frame pointers are maintained. It is never safe to delete a final stack adjustment in a function that has no frame pointer, and the compiler knows this regardless of `EXIT_IGNORE_STACK`.

`FUNCTION_EPILOGUE (file, size)`

A C compound statement that outputs the assembler code for exit from a function. The epilogue is responsible for restoring the saved registers and stack pointer to their values when the function was called, and returning control to the caller. This macro takes the same arguments as the macro `FUNCTION_PROLOGUE`, and the registers to restore are determined from `regs_ever_live` and `CALL_USED_REGISTERS` in the same way.

On some machines, there is a single instruction that does all the work of returning from the function. On these machines, give that instruction the name `return' and do not define the macro `FUNCTION_EPILOGUE` at all.

Do not define a pattern named `return' if you want the `FUNCTION_EPILOGUE` to be used. If you want the target switches to control whether return instructions or epilogues are used, define a `return' pattern with a validity condition that tests the target switches appropriately. If the `return' pattern's validity condition is false, epilogues will be used.

On machines where functions may or may not have frame-pointers, the function exit code must vary accordingly. Sometimes the code for these two cases is completely different. To determine whether a frame pointer is wanted, the macro can refer to the variable `frame_pointer_needed`. The variable's value will be 1 when compiling a function that needs a frame pointer.

Normally, `FUNCTION_PROLOGUE` and `FUNCTION_EPILOGUE` must treat leaf functions specially. The C variable `leaf_function` is nonzero for such a function. See section Handling Leaf Functions.

On some machines, some functions pop their arguments on exit while others leave that for the caller to do. For example, the 68020 when given `-mrtd' pops arguments in functions that take a fixed number of arguments.

Your definition of the macro `RETURN_POPS_ARGS` decides which functions pop their own arguments. `FUNCTION_EPILOGUE` needs to know what was decided. The variable that is called `current_function_pops_args` is the number of bytes of its arguments that a function should pop. See section How Scalar Function Values Are Returned.

`DELAY_SLOTS_FOR_EPILOGUE`

Define this macro if the function epilogue contains delay slots to which instructions from the rest

of the function can be "moved". The definition should be a C expression whose value is an int representing the number of delay slots there.

`ELIGIBLE_FOR_EPILOGUE_DELAY (insn, n)`

A C expression that returns 1 if insn can be placed in delay slot number n of the epilogue.

The argument n is an integer which identifies the delay slot now being considered (since different slots may have different rules of eligibility). It is never negative and is always less than the number of epilogue delay slots (what `DELAY_SLOTS_FOR_EPILOGUE` returns). If you reject a particular insn for a given delay slot, in principle, it may be reconsidered for a subsequent delay slot. Also, other insns may (at least in principle) be considered for the so far unfilled delay slot.

The insns accepted to fill the epilogue delay slots are put in an RTL list made with `insn_list` objects, stored in the variable `current_function_epilogue_delay_list`. The insn for the first delay slot comes first in the list. Your definition of the macro `FUNCTION_EPILOGUE` should fill the delay slots by outputting the insns in this list, usually by calling `final_scan_insn`.

You need not define this macro if you did not define `DELAY_SLOTS_FOR_EPILOGUE`.

Generating Code for Profiling

These macros will help you generate code for profiling.

`FUNCTION_PROFILER (file, labelno)`

A C statement or compound statement to output to file some assembler code to call the profiling subroutine `mcount`. Before calling, the assembler code must load the address of a counter variable into a register where `mcount` expects to find the address. The name of this variable is `LP' followed by the number labelno, so you would generate the name using `LP%d' in a `fprintf`.

The details of how the address should be passed to `mcount` are determined by your operating system environment, not by GNU CC. To figure them out, compile a small program for profiling using the system's installed C compiler and look at the assembler code that results.

`PROFILE_BEFORE_PROLOGUE`

Define this macro if the code for function profiling should come before the function prologue. Normally, the profiling code comes after.

`FUNCTION_BLOCK_PROFILER (file, labelno)`

A C statement or compound statement to output to file some assembler code to initialize basic-block profiling for the current object module. This code should call the subroutine `__bb_init_func` once per object module, passing it as its sole argument the address of a block allocated in the object module.

The name of the block is a local symbol made with this statement:

```
ASM_GENERATE_INTERNAL_LABEL (buffer, "LPBX", 0);
```

Of course, since you are writing the definition of `ASM_GENERATE_INTERNAL_LABEL` as well as that of this macro, you can take a short cut in the definition of this macro and use the name that you know will result.

The first word of this block is a flag which will be nonzero if the object module has already been initialized. So test this word first, and do not call `__bb_init_func` if the flag is nonzero.

`BLOCK_PROFILER (file, blockno)`

A C statement or compound statement to increment the count associated with the basic block number blockno. Basic blocks are numbered separately from zero within each compilation. The count associated with block number blockno is at index blockno in a vector of words; the name of this array is a local symbol made with this statement:

```
ASM_GENERATE_INTERNAL_LABEL (buffer, "LPBX", 2);
```

Of course, since you are writing the definition of `ASM_GENERATE_INTERNAL_LABEL` as well as that of this macro, you can take a short cut in the definition of this macro and use the name that you know will result.

`BLOCK_PROFILER_CODE`

A C function or functions which are needed in the library to support block profiling.

Implementing the Varargs Macros

GNU CC comes with an implementation of `` `varargs.h' `` and `` `stdarg.h' `` that work without change on machines that pass arguments on the stack. Other machines require their own implementations of varargs, and the two machine independent header files must have conditionals to include it.

ANSI `` `stdarg.h' `` differs from traditional `` `varargs.h' `` mainly in the calling convention for `va_start`. The traditional implementation takes just one argument, which is the variable in which to store the argument pointer. The ANSI implementation of `va_start` takes an additional second argument. The user is supposed to write the last named argument of the function here.

However, `va_start` should not use this argument. The way to find the end of the named arguments is with the built-in functions described below.

`__builtin_saveregs ()`

Use this built-in function to save the argument registers in memory so that the varargs mechanism can access them. Both ANSI and traditional versions of `va_start` must use `__builtin_saveregs`, unless you use `SETUP_INCOMING_VARARGS` (see below) instead.

On some machines, `__builtin_saveregs` is open-coded under the control of the macro `EXPAND_BUILTIN_SAVEREGS`. On other machines, it calls a routine written in assembler language, found in `` `libgcc2.c' ``.

Code generated for the call to `__builtin_saveregs` appears at the beginning of the function, as opposed to where the call to `__builtin_saveregs` is written, regardless of what the code is. This is because the registers must be saved before the function starts to use them for its own

purposes.

`__builtin_args_info (category)`

Use this built-in function to find the first anonymous arguments in registers.

In general, a machine may have several categories of registers used for arguments, each for a particular category of data types. (For example, on some machines, floating-point registers are used for floating-point arguments while other arguments are passed in the general registers.) To make non-varargs functions use the proper calling convention, you have defined the `CUMULATIVE_ARGS` data type to record how many registers in each category have been used so far

`__builtin_args_info` accesses the same data structure of type `CUMULATIVE_ARGS` after the ordinary argument layout is finished with it, with category specifying which word to access. Thus, the value indicates the first unused register in a given category.

Normally, you would use `__builtin_args_info` in the implementation of `va_start`, accessing each category just once and storing the value in the `va_list` object. This is because `va_list` will have to update the values, and there is no way to alter the values accessed by `__builtin_args_info`.

`__builtin_next_arg (lastarg)`

This is the equivalent of `__builtin_args_info`, for stack arguments. It returns the address of the first anonymous stack argument, as type `void *`. If `ARGS_GROW_DOWNWARD`, it returns the address of the location above the first anonymous stack argument. Use it in `va_start` to initialize the pointer for fetching arguments from the stack. Also use it in `va_start` to verify that the second parameter lastarg is the last named argument of the current function.

`__builtin_classify_type (object)`

Since each machine has its own conventions for which data types are passed in which kind of register, your implementation of `va_arg` has to embody these conventions. The easiest way to categorize the specified data type is to use `__builtin_classify_type` together with `sizeof` and `__alignof__`.

`__builtin_classify_type` ignores the value of object, considering only its data type. It returns an integer describing what kind of type that is--integer, floating, pointer, structure, and so on.

The file `` `typeclass.h' `` defines an enumeration that you can use to interpret the values of `__builtin_classify_type`.

These machine description macros help implement varargs:

`EXPAND_BUILTIN_SAVEREGS (args)`

If defined, is a C expression that produces the machine-specific code for a call to `__builtin_saveregs`. This code will be moved to the very beginning of the function, before any parameter access are made. The return value of this function should be an RTX that contains the value to use as the return of `__builtin_saveregs`.

The argument args is a `tree_list` containing the arguments that were passed to

`__builtin_saveregs`.

If this macro is not defined, the compiler will output an ordinary call to the library function `__builtin_saveregs'.

`SETUP_INCOMING_VARARGS (args_so_far, mode, type,`

pretend_args_size, second_time) This macro offers an alternative to using `__builtin_saveregs` and defining the macro `EXPAND_BUILTIN_SAVEREGS`. Use it to store the anonymous register arguments into the stack so that all the arguments appear to have been passed consecutively on the stack. Once this is done, you can use the standard implementation of varargs that works for machines that pass all their arguments on the stack.

The argument args_so_far is the `CUMULATIVE_ARGS` data structure, containing the values that obtain after processing of the named arguments. The arguments mode and type describe the last named argument--its machine mode and its data type as a tree node.

The macro implementation should do two things: first, push onto the stack all the argument registers *not* used for the named arguments, and second, store the size of the data thus pushed into the `int`-valued variable whose name is supplied as the argument pretend_args_size. The value that you store here will serve as additional offset for setting up the stack frame.

Because you must generate code to push the anonymous arguments at compile time without knowing their data types, `SETUP_INCOMING_VARARGS` is only useful on machines that have just a single category of argument register and use it uniformly for all data types.

If the argument second_time is nonzero, it means that the arguments of the function are being analyzed for the second time. This happens for an inline function, which is not actually compiled until the end of the source file. The macro `SETUP_INCOMING_VARARGS` should not generate any instructions in this case.

`STRICT_ARGUMENT_NAMING`

Define this macro if the location where a function argument is passed depends on whether or not it is a named argument.

This macro controls how the named argument to `FUNCTION_ARG` is set for varargs and stdarg functions. With this macro defined, the named argument is always true for named arguments, and false for unnamed arguments. If this is not defined, but `SETUP_INCOMING_VARARGS` is defined, then all arguments are treated as named. Otherwise, all named arguments except the last are treated as named.

Trampolines for Nested Functions

A trampoline is a small piece of code that is created at run time when the address of a nested function is taken. It normally resides on the stack, in the stack frame of the containing function. These macros tell GNU CC how to generate code to allocate and initialize a trampoline.

The instructions in the trampoline must do two things: load a constant address into the static chain register, and jump to the real address of the nested function. On CISC machines such as the m68k, this

requires two instructions, a move immediate and a jump. Then the two addresses exist in the trampol as word-long immediate operands. On RISC machines, it is often necessary to load each address into register in two parts. Then pieces of each address form separate immediate operands.

The code generated to initialize the trampoline must store the variable parts--the static chain value and the function address--into the immediate operands of the instructions. On a CISC machine, this is simply a matter of copying each address to a memory reference at the proper offset from the start of the trampoline. On a RISC machine, it may be necessary to take out pieces of the address and store them separately.

`TRAMPOLINE_TEMPLATE (file)`

A C statement to output, on the stream file, assembler code for a block of data that contains the constant parts of a trampoline. This code should not include a label--the label is taken care of automatically.

`TRAMPOLINE_SECTION`

The name of a subroutine to switch to the section in which the trampoline template is to be placed (see section Dividing the Output into Sections (Texts, Data, ...)). The default is a value of `readonly_data_section', which places the trampoline in the section containing read-only data.

`TRAMPOLINE_SIZE`

A C expression for the size in bytes of the trampoline, as an integer.

`TRAMPOLINE_ALIGNMENT`

Alignment required for trampolines, in bits.

If you don't define this macro, the value of `BIGGEST_ALIGNMENT` is used for aligning trampolines.

`INITIALIZE_TRAMPOLINE (addr, fnaddr, static_chain)`

A C statement to initialize the variable parts of a trampoline. addr is an RTX for the address of the trampoline; fnaddr is an RTX for the address of the nested function; static_chain is an RTX for the static chain value that should be passed to the function when it is called.

`ALLOCATE_TRAMPOLINE (fp)`

A C expression to allocate run-time space for a trampoline. The expression value should be an RTX representing a memory reference to the space for the trampoline.

If this macro is not defined, by default the trampoline is allocated as a stack slot. This default is right for most machines. The exceptions are machines where it is impossible to execute instructions in the stack area. On such machines, you may have to implement a separate stack, using this macro in conjunction with `FUNCTION_PROLOGUE` and `FUNCTION_EPILOGUE`.

fp points to a data structure, a `struct function`, which describes the compilation status of the immediate containing function of the function which the trampoline is for. Normally (when `ALLOCATE_TRAMPOLINE` is not defined), the stack slot for the trampoline is in the stack frame of this containing function. Other allocation strategies probably must do something analogous with this information.

Implementing trampolines is difficult on many machines because they have separate instruction and data

caches. Writing into a stack location fails to clear the memory in the instruction cache, so when the program jumps to that location, it executes the old contents.

Here are two possible solutions. One is to clear the relevant parts of the instruction cache whenever a trampoline is set up. The other is to make all trampolines identical, by having them jump to a standard subroutine. The former technique makes trampoline execution faster; the latter makes initialization faster.

To clear the instruction cache when a trampoline is initialized, define the following macros which describe the shape of the cache.

`INSN_CACHE_SIZE`

The total size in bytes of the cache.

`INSN_CACHE_LINE_WIDTH`

The length in bytes of each cache line. The cache is divided into cache lines which are disjoint slots, each holding a contiguous chunk of data fetched from memory. Each time data is brought into the cache, an entire line is read at once. The data loaded into a cache line is always aligned on a boundary equal to the line size.

`INSN_CACHE_DEPTH`

The number of alternative cache lines that can hold any particular memory location.

Alternatively, if the machine has system calls or instructions to clear the instruction cache directly, you can define the following macro.

`CLEAR_INSN_CACHE (BEG, END)`

If defined, expands to a C expression clearing the *instruction cache* in the specified interval. If it is not defined, and the macro INSN_CACHE_SIZE is defined, some generic code is generated to clear the cache. The definition of this macro would typically be a series of `asm` statements. Both BEG and END are both pointer expressions.

To use a standard subroutine, define the following macro. In addition, you must make sure that the instructions in a trampoline fill an entire cache line with identical instructions, or else ensure that the beginning of the trampoline code is always aligned at the same point in its cache line. Look in `` `m68k.h' `` as a guide.

`TRANSFER_FROM_TRAMPOLINE`

Define this macro if trampolines need a special subroutine to do their work. The macro should expand to a series of `asm` statements which will be compiled with GNU CC. They go in a library function named `__transfer_from_trampoline`.

If you need to avoid executing the ordinary prologue code of a compiled C function when you jump to the subroutine, you can do so by placing a special label of your own in the assembler code. Use one `asm` statement to generate an assembler label, and another to make the label global. Then trampolines can use that label to jump directly to your special assembler code.

Implicit Calls to Library Routines

Here is an explanation of implicit calls to library routines.

`MULSI3_LIBCALL`

A C string constant giving the name of the function to call for multiplication of one signed full-word by another. If you do not define this macro, the default name is used, which is `__mulsi3`, a function defined in `` `libgcc.a' ``.

`DIVSI3_LIBCALL`

A C string constant giving the name of the function to call for division of one signed full-word by another. If you do not define this macro, the default name is used, which is `__divsi3`, a function defined in `` `libgcc.a' ``.

`UDIVSI3_LIBCALL`

A C string constant giving the name of the function to call for division of one unsigned full-word by another. If you do not define this macro, the default name is used, which is `__udivsi3`, a function defined in `` `libgcc.a' ``.

`MODSI3_LIBCALL`

A C string constant giving the name of the function to call for the remainder in division of one signed full-word by another. If you do not define this macro, the default name is used, which is `__modsi3`, a function defined in `` `libgcc.a' ``.

`UMODSI3_LIBCALL`

A C string constant giving the name of the function to call for the remainder in division of one unsigned full-word by another. If you do not define this macro, the default name is used, which is `__umodsi3`, a function defined in `` `libgcc.a' ``.

`MULDI3_LIBCALL`

A C string constant giving the name of the function to call for multiplication of one signed double-word by another. If you do not define this macro, the default name is used, which is `__muldi3`, a function defined in `` `libgcc.a' ``.

`DIVDI3_LIBCALL`

A C string constant giving the name of the function to call for division of one signed double-word by another. If you do not define this macro, the default name is used, which is `__divdi3`, a function defined in `` `libgcc.a' ``.

`UDIVDI3_LIBCALL`

A C string constant giving the name of the function to call for division of one unsigned full-word by another. If you do not define this macro, the default name is used, which is `__udivdi3`, a function defined in `` `libgcc.a' ``.

`MODDI3_LIBCALL`

A C string constant giving the name of the function to call for the remainder in division of one signed double-word by another. If you do not define this macro, the default name is used, which is `__moddi3`, a function defined in `` `libgcc.a' ``.

`UMODDI3_LIBCALL`

A C string constant giving the name of the function to call for the remainder in division of one unsigned full-word by another. If you do not define this macro, the default name is used, which is `__umoddi3`, a function defined in `libgcc.a'.

`INIT_TARGET_OPTABS`

Define this macro as a C statement that declares additional library routines renames existing ones. `init_optabs` calls this macro after initializing all the normal library routines.

`TARGET_EDOM`

The value of `EDOM` on the target machine, as a C integer constant expression. If you don't define this macro, GNU CC does not attempt to deposit the value of `EDOM` into `errno` directly. Look in `/usr/include/errno.h' to find the value of `EDOM` on your system.

If you do not define `TARGET_EDOM`, then compiled code reports domain errors by calling the library function and letting it report the error. If mathematical functions on your system use `matherr` when there is an error, then you should leave `TARGET_EDOM` undefined so that `matherr` is used normally.

`GEN_ERRNO_RTX`

Define this macro as a C expression to create an rtl expression that refers to the global "variable" `errno`. (On certain systems, `errno` may not actually be a variable.) If you don't define this macro, a reasonable default is used.

`TARGET_MEM_FUNCTIONS`

Define this macro if GNU CC should generate calls to the System V (and ANSI C) library functions `memcpy` and `memset` rather than the BSD functions `bcopy` and `bzero`.

`LIBGCC_NEEDS_DOUBLE`

Define this macro if only `float` arguments cannot be passed to library routines (so they must be converted to `double`). This macro affects both how library calls are generated and how the library routines in `libgcc1.c' accept their arguments. It is useful on machines where floating and fixed point arguments are passed differently, such as the i860.

`FLOAT_ARG_TYPE`

Define this macro to override the type used by the library routines to pick up arguments of type `float`. (By default, they use a union of `float` and `int`.)

The obvious choice would be `float`---but that won't work with traditional C compilers that expect all arguments declared as `float` to arrive as `double`. To avoid this conversion, the library routines ask for the value as some other type and then treat it as a `float`.

On some systems, no other type will work for this. For these systems, you must use `LIBGCC_NEEDS_DOUBLE` instead, to force conversion of the values `double` before they are passed.

`FLOATIFY (passed-value)`

Define this macro to override the way library routines redesignate a `float` argument as a `float` instead of the type it was passed as. The default is an expression which takes the `float` field of the union.

`FLOAT_VALUE_TYPE`

Define this macro to override the type used by the library routines to return values that ought t have type `float`. (By default, they use `int`.)

The obvious choice would be `float`---but that won't work with traditional C compilers gratuitously convert values declared as `float` into `double`.

`INTIFY (float-value)`

Define this macro to override the way the value of a `float`-returning library routine should be packaged in order to return it. These functions are actually declared to return type `FLOAT_VALUE_TYPE` (normally `int`).

These values can't be returned as type `float` because traditional C compilers would gratuitously convert the value to a `double`.

A local variable named `intify` is always available when the macro `INTIFY` is used. It is a union of a `float` field named `f` and a field named `i` whose type is `FLOAT_VALUE_TYPE` or `int`.

If you don't define this macro, the default definition works by copying the value through that union.

`nongcc_SI_type`

Define this macro as the name of the data type corresponding to `SImode` in the system's own C compiler.

You need not define this macro if that type is `long int`, as it usually is.

`nongcc_word_type`

Define this macro as the name of the data type corresponding to the word_mode in the system's own C compiler.

You need not define this macro if that type is `long int`, as it usually is.

`perform_...`

Define these macros to supply explicit C statements to carry out various arithmetic operations on types `float` and `double` in the library routines in `` `libgcc1.c' ``. See that file for a full list of these macros and their arguments.

On most machines, you don't need to define any of these macros, because the C compiler that comes with the system takes care of doing them.

`NEXT_OBJC_RUNTIME`

Define this macro to generate code for Objective C message sending using the calling convention of the NeXT system. This calling convention involves passing the object, the selector and the method arguments all at once to the method-lookup library function.

The default calling convention passes just the object and the selector to the lookup function, which returns a pointer to the method.

Addressing Modes

This is about addressing modes.

`HAVE_POST_INCREMENT`

Define this macro if the machine supports post-increment addressing.

`HAVE_PRE_INCREMENT`

`HAVE_POST_DECREMENT`

`HAVE_PRE_DECREMENT`

Similar for other kinds of addressing.

`CONSTANT_ADDRESS_P (x)`

A C expression that is 1 if the RTX x is a constant which is a valid address. On most machines, this can be defined as `CONSTANT_P (x)`, but a few machines are more restrictive in which constant addresses are supported.

`CONSTANT_P` accepts integer-values expressions whose values are not explicitly known, such as `symbol_ref`, `label_ref`, and `high` expressions and `const` arithmetic expressions, in addition to `const_int` and `const_double` expressions.

`MAX_REGS_PER_ADDRESS`

A number, the maximum number of registers that can appear in a valid memory address. Note that it is up to you to specify a value equal to the maximum number that `GO_IF_LEGITIMATE_ADDRESS` would ever accept.

`GO_IF_LEGITIMATE_ADDRESS (mode, x, label)`

A C compound statement with a conditional `goto label;` executed if x (an RTX) is a legitimate memory address on the target machine for a memory operand of mode mode.

It usually pays to define several simpler macros to serve as subroutines for this one. Otherwise it may be too complicated to understand.

This macro must exist in two variants: a strict variant and a non-strict one. The strict variant is used in the reload pass. It must be defined so that any pseudo-register that has not been allocated a hard register is considered a memory reference. In contexts where some kind of register is required, a pseudo-register with no hard register must be rejected.

The non-strict variant is used in other passes. It must be defined to accept all pseudo-registers in every context where some kind of register is required.

Compiler source files that want to use the strict variant of this macro define the macro `REG_OK_STRICT`. You should use an `#ifdef REG_OK_STRICT` conditional to define the strict variant in that case and the non-strict variant otherwise.

Subroutines to check for acceptable registers for various purposes (one for base registers, one for index registers, and so on) are typically among the subroutines used to define `GO_IF_LEGITIMATE_ADDRESS`. Then only these subroutine macros need have two variants; the higher levels of macros may be the same whether strict or not.

Normally, constant addresses which are the sum of a `symbol_ref` and an integer are stored inside a `const` RTX to mark them as constant. Therefore, there is no need to recognize such sums specifically as legitimate addresses. Normally you would simply recognize any `const` as legitimate.

Usually `PRINT_OPERAND_ADDRESS` is not prepared to handle constant sums that are not marked with `const`. It assumes that a naked `plus` indicates indexing. If so, then you *must* reject such naked constant sums as illegitimate addresses, so that none of them will be given to `PRINT_OPERAND_ADDRESS`.

On some machines, whether a symbolic address is legitimate depends on the section that the address refers to. On these machines, define the macro `ENCODE_SECTION_INFO` to store the information into the `symbol_ref`, and then check for it here. When you see a `const`, you will have to look inside it to find the `symbol_ref` in order to determine the section. See section Defining the Output Assembler Language.

The best way to modify the name string is by adding text to the beginning, with suitable punctuation to prevent any ambiguity. Allocate the new name in `saveable_obstack`. You will have to modify `ASM_OUTPUT_LABELREF` to remove and decode the added text and output the name accordingly, and define `STRIP_NAME_ENCODING` to access the original name string.

You can check the information stored here into the `symbol_ref` in the definitions of the macros `GO_IF_LEGITIMATE_ADDRESS` and `PRINT_OPERAND_ADDRESS`.

`REG_OK_FOR_BASE_P (x)`

A C expression that is nonzero if x (assumed to be a `reg` RTX) is valid for use as a base register. For hard registers, it should always accept those which the hardware permits and reject the others. Whether the macro accepts or rejects pseudo registers must be controlled by `REG_OK_STRICT` as described above. This usually requires two variant definitions, of which `REG_OK_STRICT` controls the one actually used.

`REG_OK_FOR_INDEX_P (x)`

A C expression that is nonzero if x (assumed to be a `reg` RTX) is valid for use as an index register.

The difference between an index register and a base register is that the index register may be scaled. If an address involves the sum of two registers, neither one of them scaled, then either one may be labeled the "base" and the other the "index"; but whichever labeling is used must fit the machine's constraints of which registers may serve in each capacity. The compiler will try both labelings, looking for one that is valid, and will reload one or both registers only if neither labeling works.

`LEGITIMIZE_ADDRESS (x, oldx, mode, win)`

A C compound statement that attempts to replace x with a valid memory address for an operand of mode mode. win will be a C statement label elsewhere in the code; the macro definition may use

```
GO_IF_LEGITIMATE_ADDRESS (mode, x, win);
```

to avoid further processing if the address has become legitimate.

x will always be the result of a call to `break_out_memory_refs`, and oldx will be the operand that was given to that function to produce x.

The code generated by this macro should not alter the substructure of x. If it transforms x into a more legitimate form, it should assign x (which will always be a C variable) a new value.

It is not necessary for this macro to come up with a legitimate address. The compiler has standard ways of doing so in all cases. In fact, it is safe for this macro to do nothing. But often a machine-dependent strategy can generate better code.

`GO_IF_MODE_DEPENDENT_ADDRESS (addr, label)`

A C statement or compound statement with a conditional `goto label;` executed if memory address x (an RTX) can have different meanings depending on the machine mode of the memory reference it is used for or if the address is valid for some modes but not others.

Autoincrement and autodecrement addresses typically have mode-dependent effects because the amount of the increment or decrement is the size of the operand being addressed. Some machines have other mode-dependent addresses. Many RISC machines have no mode-dependent addresses.

You may assume that addr is a valid address for the machine.

`LEGITIMATE_CONSTANT_P (x)`

A C expression that is nonzero if x is a legitimate constant for an immediate operand on the target machine. You can assume that x satisfies `CONSTANT_P`, so you need not check this. In fact, `1' is a suitable definition for this macro on machines where anything `CONSTANT_P` is valid.

Condition Code Status

This describes the condition code status.

The file `` `conditions.h' `` defines a variable `cc_status` to describe how the condition code was computed (in case the interpretation of the condition code depends on the instruction that it was set by). This variable contains the RTL expressions on which the condition code is currently based, and several standard flags.

Sometimes additional machine-specific flags must be defined in the machine description header file. It can also add additional machine-specific information by defining `CC_STATUS_MDEP`.

`CC_STATUS_MDEP`

C code for a data type which is used for declaring the `mdep` component of `cc_status`. It defaults to `int`.

This macro is not used on machines that do not use `cc0`.

`CC_STATUS_MDEP_INIT`

A C expression to initialize the `mdep` field to "empty". The default definition does nothing, since most machines don't use the field anyway. If you want to use the field, you should probably define this macro to initialize it.

This macro is not used on machines that do not use `cc0`.

`NOTICE_UPDATE_CC (exp, insn)`

A C compound statement to set the components of `cc_status` appropriately for an insn insn whose body is exp. It is this macro's responsibility to recognize insns that set the condition code as a byproduct of other activity as well as those that explicitly set `(cc0)`.

This macro is not used on machines that do not use `cc0`.

If there are insns that do not set the condition code but do alter other machine registers, this macro must check to see whether they invalidate the expressions that the condition code is recorded as reflecting. For example, on the 68000, insns that store in address registers do not set the condition code, which means that usually `NOTICE_UPDATE_CC` can leave `cc_status` unaltered for such insns. But suppose that the previous insn set the condition code based on location `a4@(102)' and the current insn stores a new value in `a4'. Although the condition code is not changed by this, it will no longer be true that it reflects the contents of `a4@(102)'. Therefore, `NOTICE_UPDATE_CC` must alter `cc_status` in this case to say that nothing is known about the condition code value.

The definition of `NOTICE_UPDATE_CC` must be prepared to deal with the results of peephole optimization: insns whose patterns are `parallel` RTXs containing various `reg`, `mem` or constants which are just the operands. The RTL structure of these insns is not sufficient to indicate what the insns actually do. What `NOTICE_UPDATE_CC` should do when it sees one is just to run `CC_STATUS_INIT`.

A possible definition of `NOTICE_UPDATE_CC` is to call a function that looks at an attribute (see section Instruction Attributes) named, for example, `cc'. This avoids having detailed information about patterns in two places, the `md' file and in `NOTICE_UPDATE_CC`.

`EXTRA_CC_MODES`

A list of names to be used for additional modes for condition code values in registers (see section Defining Jump Instruction Patterns). These names are added to `enum machine_mode` and all have class `MODE_CC`. By convention, they should start with `CC' and end with `mode'.

You should only define this macro if your machine does not use `cc0` and only if additional modes are required.

`EXTRA_CC_NAMES`

A list of C strings giving the names for the modes listed in `EXTRA_CC_MODES`. For example, the Sparc defines this macro and `EXTRA_CC_MODES` as

```
#define EXTRA_CC_MODES CC_NOOVmode, CCFPmode, CCFPEmode
#define EXTRA_CC_NAMES "CC_NOOV", "CCFP", "CCFPE"
```

This macro is not required if `EXTRA_CC_MODES` is not defined.

`SELECT_CC_MODE (op, x, y)`

Returns a mode from class `MODE_CC` to be used when comparison operation code op is applied to rtx x and y. For example, on the Sparc, `SELECT_CC_MODE` is defined as (see see section

Defining Jump Instruction Patterns for a description of the reason for this definition)

```
#define SELECT_CC_MODE(OP,X,Y) \
  (GET_MODE_CLASS (GET_MODE (X)) == MODE_FLOAT          \
   ? ((OP == EQ || OP == NE) ? CCFPmode : CCFPEmode)    \
   : ((GET_CODE (X) == PLUS || GET_CODE (X) == MINUS    \
       || GET_CODE (X) == NEG) \
      ? CC_NOOVmode : CCmode))
```

You need not define this macro if `EXTRA_CC_MODES` is not defined.

`CANONICALIZE_COMPARISON (code, op0, op1)`

One some machines not all possible comparisons are defined, but you can convert an invalid comparison into a valid one. For example, the Alpha does not have a `GT` comparison, but you can use an `LT` comparison instead and swap the order of the operands.

On such machines, define this macro to be a C statement to do any required conversions. code is the initial comparison code and op0 and op1 are the left and right operands of the comparison, respectively. You should modify code, op0, and op1 as required.

GNU CC will not assume that the comparison resulting from this macro is valid but will see if the resulting insn matches a pattern in the `` `md' `` file.

You need not define this macro if it would never change the comparison code or operands.

`REVERSIBLE_CC_MODE (mode)`

A C expression whose value is one if it is always safe to reverse a comparison whose mode is mode. If `SELECT_CC_MODE` can ever return mode for a floating-point inequality comparison, then `REVERSIBLE_CC_MODE (mode)` must be zero.

You need not define this macro if it would always returns zero or if the floating-point format is anything other than `IEEE_FLOAT_FORMAT`. For example, here is the definition used on the Sparc, where floating-point inequality comparisons are always given `CCFPEmode`:

```
#define REVERSIBLE_CC_MODE(MODE)  ((MODE) != CCFPEmode)
```

Describing Relative Costs of Operations

These macros let you describe the relative speed of various operations on the target machine.

`CONST_COSTS (x, code, outer_code)`

A part of a C `switch` statement that describes the relative costs of constant RTL expressions. It must contain `case` labels for expression codes `const_int`, `const`, `symbol_ref`, `label_ref` and `const_double`. Each case must ultimately reach a `return` statement to return the relative cost of the use of that kind of constant value in an expression. The cost may depend on the precise value of the constant, which is available for examination in x, and the rtx code of the expression in which it is contained, found in outer_code.

code is the expression code--redundant, since it can be obtained with `GET_CODE (x)`.

`RTX_COSTS (x, code, outer_code)`

Like `CONST_COSTS` but applies to nonconstant RTL expressions. This can be used, for example, to indicate how costly a multiply instruction is. In writing this macro, you can use the construct `COSTS_N_INSNS (n)` to specify a cost equal to n fast instructions. outer_code is the code of the expression in which x is contained.

This macro is optional; do not define it if the default cost assumptions are adequate for the target machine.

`ADDRESS_COST (address)`

An expression giving the cost of an addressing mode that contains address. If not defined, the cost is computed from the address expression and the `CONST_COSTS` values.

For most CISC machines, the default cost is a good approximation of the true cost of the addressing mode. However, on RISC machines, all instructions normally have the same length and execution time. Hence all addresses will have equal costs.

In cases where more than one form of an address is known, the form with the lowest cost will be used. If multiple forms have the same, lowest, cost, the one that is the most complex will be used.

For example, suppose an address that is equal to the sum of a register and a constant is used twice in the same basic block. When this macro is not defined, the address will be computed in a register and memory references will be indirect through that register. On machines where the cost of the addressing mode containing the sum is no higher than that of a simple indirect reference, this will produce an additional instruction and possibly require an additional register. Proper specification of this macro eliminates this overhead for such machines.

Similar use of this macro is made in strength reduction of loops.

address need not be valid as an address. In such a case, the cost is not relevant and can be any value; invalid addresses need not be assigned a different cost.

On machines where an address involving more than one register is as cheap as an address computation involving only one register, defining `ADDRESS_COST` to reflect this can cause two registers to be live over a region of code where only one would have been if `ADDRESS_COST` were not defined in that manner. This effect should be considered in the definition of this macro. Equivalent costs should probably only be given to addresses with different numbers of registers on machines with lots of registers.

This macro will normally either not be defined or be defined as a constant.

`REGISTER_MOVE_COST (from, to)`

A C expression for the cost of moving data from a register in class from to one in class to. The classes are expressed using the enumeration values such as `GENERAL_REGS`. A value of 4 is the default; other values are interpreted relative to that.

It is not required that the cost always equal 2 when from is the same as to; on some machines it is expensive to move between registers if they are not general registers.

If reload sees an insn consisting of a single `set` between two hard registers, and if `REGISTER_MOVE_COST` applied to their classes returns a value of 2, reload does not check to ensure that the constraints of the insn are met. Setting a cost of other than 2 will allow reload to verify that the constraints are met. You should do this if the `movm' pattern's constraints do not allow such copying.

`MEMORY_MOVE_COST (m)`

A C expression for the cost of moving data of mode m between a register and memory. A value of 2 is the default; this cost is relative to those in `REGISTER_MOVE_COST`.

If moving between registers and memory is more expensive than between two registers, you should define this macro to express the relative cost.

`BRANCH_COST`

A C expression for the cost of a branch instruction. A value of 1 is the default; other values are interpreted relative to that.

Here are additional macros which do not specify precise relative costs, but only that certain actions are more expensive than GNU CC would ordinarily expect.

`SLOW_BYTE_ACCESS`

Define this macro as a C expression which is nonzero if accessing less than a word of memory (i.e. a `char` or a `short`) is no faster than accessing a word of memory, i.e., if such access require more than one instruction or if there is no difference in cost between byte and (aligned) word loads.

When this macro is not defined, the compiler will access a field by finding the smallest containing object; when it is defined, a fullword load will be used if alignment permits. Unless bytes accesses are faster than word accesses, using word accesses is preferable since it may eliminate subsequent memory access if subsequent accesses occur to other fields in the same word of the structure, but to different bytes.

`SLOW_ZERO_EXTEND`

Define this macro if zero-extension (of a `char` or `short` to an `int`) can be done faster if the destination is a register that is known to be zero.

If you define this macro, you must have instruction patterns that recognize RTL structures like this:

```
(set (strict_low_part (subreg:QI (reg:SI ...) 0)) ...)
```

and likewise for `HImode`.

`SLOW_UNALIGNED_ACCESS`

Define this macro to be the value 1 if unaligned accesses have a cost many times greater than aligned accesses, for example if they are emulated in a trap handler.

When this macro is non-zero, the compiler will act as if `STRICT_ALIGNMENT` were non-zero when generating code for block moves. This can cause significantly more instructions to be produced. Therefore, do not set this macro non-zero if unaligned accesses only add a cycle or two

to the time for a memory access.

If the value of this macro is always zero, it need not be defined.

`DONT_REDUCE_ADDR`

Define this macro to inhibit strength reduction of memory addresses. (On some machines, such strength reduction seems to do harm rather than good.)

`MOVE_RATIO`

The number of scalar move insns which should be generated instead of a string move insn or a library call. Increasing the value will always make code faster, but eventually incurs high cost in increased code size.

If you don't define this, a reasonable default is used.

`NO_FUNCTION_CSE`

Define this macro if it is as good or better to call a constant function address than to call an address kept in a register.

`NO_RECURSIVE_FUNCTION_CSE`

Define this macro if it is as good or better for a function to call itself with an explicit address than to call an address kept in a register.

`ADJUST_COST (insn, link, dep_insn, cost)`

A C statement (sans semicolon) to update the integer variable cost based on the relationship between insn that is dependent on dep_insn through the dependence link. The default is to make no adjustment to cost. This can be used for example to specify to the scheduler that an output- or anti-dependence does not incur the same cost as a data-dependence.

Dividing the Output into Sections (Texts, Data, ...)

An object file is divided into sections containing different types of data. In the most common case, there are three sections: the text section, which holds instructions and read-only data; the data section, which holds initialized writable data; and the bss section, which holds uninitialized data. Some systems have other kinds of sections.

The compiler must tell the assembler when to switch sections. These macros control what commands to output to tell the assembler this. You can also define additional sections.

`TEXT_SECTION_ASM_OP`

A C expression whose value is a string containing the assembler operation that should precede instructions and read-only data. Normally `".text"` is right.

`DATA_SECTION_ASM_OP`

A C expression whose value is a string containing the assembler operation to identify the following data as writable initialized data. Normally `".data"` is right.

`SHARED_SECTION_ASM_OP`

if defined, a C expression whose value is a string containing the assembler operation to identify the following data as shared data. If not defined, `DATA_SECTION_ASM_OP` will be used.

`INIT_SECTION_ASM_OP`

if defined, a C expression whose value is a string containing the assembler operation to identify the following data as initialization code. If not defined, GNU CC will assume such a section does not exist.

`EXTRA_SECTIONS`

A list of names for sections other than the standard two, which are `in_text` and `in_data`. You need not define this macro on a system with no other sections (that GCC needs to use).

`EXTRA_SECTION_FUNCTIONS`

One or more functions to be defined in `` `varasm.c' ``. These functions should do jobs analogous to those of `text_section` and `data_section`, for your additional sections. Do not define this macro if you do not define `EXTRA_SECTIONS`.

`READONLY_DATA_SECTION`

On most machines, read-only variables, constants, and jump tables are placed in the text section. If this is not the case on your machine, this macro should be defined to be the name of a function (either `data_section` or a function defined in `EXTRA_SECTIONS`) that switches to the section to be used for read-only items.

If these items should be placed in the text section, this macro should not be defined.

`SELECT_SECTION (exp, reloc)`

A C statement or statements to switch to the appropriate section for output of exp. You can assume that exp is either a `VAR_DECL` node or a constant of some sort. reloc indicates whether the initial value of exp requires link-time relocations. Select the section by calling `text_section` or one of the alternatives for other sections.

Do not define this macro if you put all read-only variables and constants in the read-only data section (usually the text section).

`SELECT_RTX_SECTION (mode, rtx)`

A C statement or statements to switch to the appropriate section for output of rtx in mode mode. You can assume that rtx is some kind of constant in RTL. The argument mode is redundant except in the case of a `const_int` rtx. Select the section by calling `text_section` or one of the alternatives for other sections.

Do not define this macro if you put all constants in the read-only data section.

`JUMP_TABLES_IN_TEXT_SECTION`

Define this macro if jump tables (for `tablejump` insns) should be output in the text section, along with the assembler instructions. Otherwise, the readonly data section is used.

This macro is irrelevant if there is no separate readonly data section.

`ENCODE_SECTION_INFO (decl)`

Define this macro if references to a symbol must be treated differently depending on something about the variable or function named by the symbol (such as what section it is in).

The macro definition, if any, is executed immediately after the rtl for decl has been created and stored in `DECL_RTL (decl)`. The value of the rtl will be a `mem` whose address is a

`symbol_ref`.

The usual thing for this macro to do is to record a flag in the `symbol_ref` (such as `SYMBOL_REF_FLAG`) or to store a modified name string in the `symbol_ref` (if one bit is not enough information).

`STRIP_NAME_ENCODING (var, sym_name)`

Decode sym_name and store the real name part in var, sans the characters that encode section info. Define this macro if `ENCODE_SECTION_INFO` alters the symbol's name string.

Position Independent Code

This section describes macros that help implement generation of position independent code. Simply defining these macros is not enough to generate valid PIC; you must also add support to the macros `GO_IF_LEGITIMATE_ADDRESS` and `PRINT_OPERAND_ADDRESS`, as well as `LEGITIMIZE_ADDRESS`. You must modify the definition of `movsi' to do something appropriate when the source operand contains a symbolic address. You may also need to alter the handling of switch statements so that they use relative addresses.

`PIC_OFFSET_TABLE_REGNUM`

The register number of the register used to address a table of static data addresses in memory. In some cases this register is defined by a processor's "application binary interface" (ABI). When this macro is defined, RTL is generated for this register once, as with the stack pointer and frame pointer registers. If this macro is not defined, it is up to the machine-dependent files to allocate such a register (if necessary).

`PIC_OFFSET_TABLE_REG_CALL_CLOBBERED`

Define this macro if the register defined by `PIC_OFFSET_TABLE_REGNUM` is clobbered by calls. Do not define this macro if `PPIC_OFFSET_TABLE_REGNUM` is not defined.

`FINALIZE_PIC`

By generating position-independent code, when two different programs (A and B) share a common library (libC.a), the text of the library can be shared whether or not the library is linked at the same address for both programs. In some of these environments, position-independent code requires not only the use of different addressing modes, but also special code to enable the use of these addressing modes.

The `FINALIZE_PIC` macro serves as a hook to emit these special codes once the function is being compiled into assembly code, but not before. (It is not done before, because in the case of compiling an inline function, it would lead to multiple PIC prologues being included in functions which used inline functions and were compiled to assembly language.)

`LEGITIMATE_PIC_OPERAND_P (x)`

A C expression that is nonzero if x is a legitimate immediate operand on the target machine when generating position independent code. You can assume that x satisfies `CONSTANT_P`, so you need not check this. You can also assume flag_pic is true, so you need not check it either. You need not define this macro if all constants (including `SYMBOL_REF`) can be immediate operands when generating position independent code.

Defining the Output Assembler Language

This section describes macros whose principal purpose is to describe how to write instructions in assembler language--rather than what the instructions do.

The Overall Framework of an Assembler File

This describes the overall framework of an assembler file.

`ASM_FILE_START (stream)`

A C expression which outputs to the stdio stream stream some appropriate text to go at the start of an assembler file.

Normally this macro is defined to output a line containing `#NO_APP', which is a comment that has no effect on most assemblers but tells the GNU assembler that it can save time by not checking for certain assembler constructs.

On systems that use SDB, it is necessary to output certain commands; see `attasm.h'.

`ASM_FILE_END (stream)`

A C expression which outputs to the stdio stream stream some appropriate text to go at the end of an assembler file.

If this macro is not defined, the default is to output nothing special at the end of the file. Most systems don't require any definition.

On systems that use SDB, it is necessary to output certain commands; see `attasm.h'.

`ASM_IDENTIFY_GCC (file)`

A C statement to output assembler commands which will identify the object file as having been compiled with GNU CC (or another GNU compiler).

If you don't define this macro, the string `gcc_compiled.:' is output. This string is calculated to define a symbol which, on BSD systems, will never be defined for any other reason. GDB checks for the presence of this symbol when reading the symbol table of an executable.

On non-BSD systems, you must arrange communication with GDB in some other fashion. If GDB is not used on your system, you can define this macro with an empty body.

`ASM_COMMENT_START`

A C string constant describing how to begin a comment in the target assembler language. The compiler assumes that the comment will end at the end of the line.

`ASM_APP_ON`

A C string constant for text to be output before each `asm` statement or group of consecutive ones. Normally this is `"#APP"`, which is a comment that has no effect on most assemblers but tells the GNU assembler that it must check the lines that follow for all valid assembler constructs.

`ASM_APP_OFF`

A C string constant for text to be output after each `asm` statement or group of consecutive one. Normally this is `"#NO_APP"`, which tells the GNU assembler to resume making the time-sav assumptions that are valid for ordinary compiler output.

`ASM_OUTPUT_SOURCE_FILENAME (stream, name)`

A C statement to output COFF information or DWARF debugging information which indicates that filename name is the current source file to the stdio stream stream.

This macro need not be defined if the standard form of output for the file format in use is appropriate.

`ASM_OUTPUT_SOURCE_LINE (stream, line)`

A C statement to output DBX or SDB debugging information before code for line number line of the current source file to the stdio stream stream.

This macro need not be defined if the standard form of debugging information for the debugger in use is appropriate.

`ASM_OUTPUT_IDENT (stream, string)`

A C statement to output something to the assembler file to handle a `#ident' directive containing the text string. If this macro is not defined, nothing is output for a `#ident' directive.

`ASM_OUTPUT_SECTION_NAME (stream, decl, name)`

A C statement to output something to the assembler file to switch to section name for object decl which is either a `FUNCTION_DECL`, a `VAR_DECL` or `NULL_TREE`. Some target formats do not support arbitrary sections. Do not define this macro in such cases.

At present this macro is only used to support section attributes. When this macro is undefined, section attributes are disabled.

`OBJC_PROLOGUE`

A C statement to output any assembler statements which are required to precede any Objective C object definitions or message sending. The statement is executed only when compiling an Objective C program.

Output of Data

This describes data output.

`ASM_OUTPUT_LONG_DOUBLE (stream, value)`

`ASM_OUTPUT_DOUBLE (stream, value)`

`ASM_OUTPUT_FLOAT (stream, value)`

`ASM_OUTPUT_THREE_QUARTER_FLOAT (stream, value)`

`ASM_OUTPUT_SHORT_FLOAT (stream, value)`

`ASM_OUTPUT_BYTE_FLOAT (stream, value)`

A C statement to output to the stdio stream stream an assembler instruction to assemble a floating-point constant of `TFmode`, `DFmode`, `SFmode`, `TQFmode`, `HFmode`, or `QFmode`, respectively, whose value is value. value will be a C expression of type `REAL_VALUE_TYPE`.

Macros such as `REAL_VALUE_TO_TARGET_DOUBLE` are useful for writing these definitions.

`ASM_OUTPUT_QUADRUPLE_INT (stream, exp)`

`ASM_OUTPUT_DOUBLE_INT (stream, exp)`

`ASM_OUTPUT_INT (stream, exp)`

`ASM_OUTPUT_SHORT (stream, exp)`

`ASM_OUTPUT_CHAR (stream, exp)`

A C statement to output to the stdio stream stream an assembler instruction to assemble an integer of 16, 8, 4, 2 or 1 bytes, respectively, whose value is value. The argument exp will be an RTL expression which represents a constant value. Use `output_addr_const (stream, exp)' to output this value as an assembler expression.

For sizes larger than `UNITS_PER_WORD`, if the action of a macro would be identical to repeatedly calling the macro corresponding to a size of `UNITS_PER_WORD`, once for each word, you need not define the macro.

`ASM_OUTPUT_BYTE (stream, value)`

A C statement to output to the stdio stream stream an assembler instruction to assemble a single byte containing the number value.

`ASM_BYTE_OP`

A C string constant giving the pseudo-op to use for a sequence of single-byte constants. If this macro is not defined, the default is `"byte"`.

`ASM_OUTPUT_ASCII (stream, ptr, len)`

A C statement to output to the stdio stream stream an assembler instruction to assemble a string constant containing the len bytes at ptr. ptr will be a C expression of type `char *` and len a C expression of type `int`.

If the assembler has a `.ascii` pseudo-op as found in the Berkeley Unix assembler, do not define the macro `ASM_OUTPUT_ASCII`.

`ASM_OUTPUT_POOL_PROLOGUE (file funname fundecl size)`

A C statement to output assembler commands to define the start of the constant pool for a function. funname is a string giving the name of the function. Should the return type of the function be required, it can be obtained via fundecl. size is the size, in bytes, of the constant pool that will be written immediately after this call.

If no constant-pool prefix is required, the usual case, this macro need not be defined.

`ASM_OUTPUT_SPECIAL_POOL_ENTRY (file, x, mode, align, labelno, jumpto)`

A C statement (with or without semicolon) to output a constant in the constant pool, if it needs special treatment. (This macro need not do anything for RTL expressions that can be output normally.)

The argument file is the standard I/O stream to output the assembler code on. x is the RTL expression for the constant to output, and mode is the machine mode (in case x is a `const_int'). align is the required alignment for the value x; you should output an assembler directive to force this much alignment.

The argument labelno is a number to use in an internal label for the address of this pool entry. The definition of this macro is responsible for outputting the label definition at the proper place. Here is how to do this:

```
ASM_OUTPUT_INTERNAL_LABEL (file, "LC", labelno);
```

When you output a pool entry specially, you should end with a `goto` to the label jumpto. This will prevent the same pool entry from being output a second time in the usual manner.

You need not define this macro if it would do nothing.

`IS_ASM_LOGICAL_LINE_SEPARATOR (C)`

Define this macro as a C expression which is nonzero if C is used as a logical line separator by the assembler.

If you do not define this macro, the default is that only the character `;' is treated as a logical line separator.

`ASM_OPEN_PAREN`

`ASM_CLOSE_PAREN`

These macros are defined as C string constant, describing the syntax in the assembler for grouping arithmetic expressions. The following definitions are correct for most assemblers:

```
#define ASM_OPEN_PAREN "("
#define ASM_CLOSE_PAREN ")"
```

These macros are provided by `real.h' for writing the definitions of `ASM_OUTPUT_DOUBLE` and the like:

`REAL_VALUE_TO_TARGET_SINGLE (x, l)`

`REAL_VALUE_TO_TARGET_DOUBLE (x, l)`

`REAL_VALUE_TO_TARGET_LONG_DOUBLE (x, l)`

These translate x, of type `REAL_VALUE_TYPE`, to the target's floating point representation, and store its bit pattern in the array of `long int` whose address is l. The number of elements in the output array is determined by the size of the desired target floating point data type: 32 bits of it go in each `long int` array element. Each array element holds 32 bits of the result, even if `long int` is wider than 32 bits on the host machine.

The array element values are designed so that you can print them out using `fprintf` in the order they should appear in the target machine's memory.

`REAL_VALUE_TO_DECIMAL (x, format, string)`

This macro converts x, of type `REAL_VALUE_TYPE`, to a decimal number and stores it as a string into string. You must pass, as string, the address of a long enough block of space to hold the result.

The argument format is a `printf`-specification that serves as a suggestion for how to format the output string.

Output of Uninitialized Variables

Each of the macros in this section is used to do the whole job of outputting a single uninitialized variable.

`ASM_OUTPUT_COMMON (stream, name, size, rounded)`

A C statement (sans semicolon) to output to the stdio stream stream the assembler definition of a common-label named name whose size is size bytes. The variable rounded is the size rounded up to whatever alignment the caller wants.

Use the expression `assemble_name (stream, name)` to output the name itself; before and after that, output the additional assembler syntax for defining the name, and a newline.

This macro controls how the assembler definitions of uninitialized global variables are output.

`ASM_OUTPUT_ALIGNED_COMMON (stream, name, size, alignment)`

Like `ASM_OUTPUT_COMMON` except takes the required alignment as a separate, explicit argument. If you define this macro, it is used in place of `ASM_OUTPUT_COMMON`, and gives you more flexibility in handling the required alignment of the variable. The alignment is specified as the number of bits.

`ASM_OUTPUT_SHARED_COMMON (stream, name, size, rounded)`

If defined, it is similar to `ASM_OUTPUT_COMMON`, except that it is used when name is shared. If not defined, `ASM_OUTPUT_COMMON` will be used.

`ASM_OUTPUT_LOCAL (stream, name, size, rounded)`

A C statement (sans semicolon) to output to the stdio stream stream the assembler definition of a local-common-label named name whose size is size bytes. The variable rounded is the size rounded up to whatever alignment the caller wants.

Use the expression `assemble_name (stream, name)` to output the name itself; before and after that, output the additional assembler syntax for defining the name, and a newline.

This macro controls how the assembler definitions of uninitialized static variables are output.

`ASM_OUTPUT_ALIGNED_LOCAL (stream, name, size, alignment)`

Like `ASM_OUTPUT_LOCAL` except takes the required alignment as a separate, explicit argument. If you define this macro, it is used in place of `ASM_OUTPUT_LOCAL`, and gives you more flexibility in handling the required alignment of the variable. The alignment is specified as the number of bits.

`ASM_OUTPUT_SHARED_LOCAL (stream, name, size, rounded)`

If defined, it is similar to `ASM_OUTPUT_LOCAL`, except that it is used when name is shared. If not defined, `ASM_OUTPUT_LOCAL` will be used.

Output and Generation of Labels

This is about outputting labels.

`ASM_OUTPUT_LABEL (stream, name)`

A C statement (sans semicolon) to output to the stdio stream stream the assembler definition of a

label named name. Use the expression `assemble_name (stream, name)` to output the name itself; before and after that, output the additional assembler syntax for defining the name, and a newline.

`ASM_DECLARE_FUNCTION_NAME (stream, name, decl)`

A C statement (sans semicolon) to output to the stdio stream stream any text necessary for declaring the name name of a function which is being defined. This macro is responsible for outputting the label definition (perhaps using `ASM_OUTPUT_LABEL`). The argument decl is the `FUNCTION_DECL` tree node representing the function.

If this macro is not defined, then the function name is defined in the usual manner as a label (by means of `ASM_OUTPUT_LABEL`).

`ASM_DECLARE_FUNCTION_SIZE (stream, name, decl)`

A C statement (sans semicolon) to output to the stdio stream stream any text necessary for declaring the size of a function which is being defined. The argument name is the name of the function. The argument decl is the `FUNCTION_DECL` tree node representing the function.

If this macro is not defined, then the function size is not defined.

`ASM_DECLARE_OBJECT_NAME (stream, name, decl)`

A C statement (sans semicolon) to output to the stdio stream stream any text necessary for declaring the name name of an initialized variable which is being defined. This macro must output the label definition (perhaps using `ASM_OUTPUT_LABEL`). The argument decl is the `VAR_DECL` tree node representing the variable.

If this macro is not defined, then the variable name is defined in the usual manner as a label (by means of `ASM_OUTPUT_LABEL`).

`ASM_FINISH_DECLARE_OBJECT (stream, decl, toplevel, atend)`

A C statement (sans semicolon) to finish up declaring a variable name once the compiler has processed its initializer fully and thus has had a chance to determine the size of an array when controlled by an initializer. This is used on systems where it's necessary to declare something about the size of the object.

If you don't define this macro, that is equivalent to defining it to do nothing.

`ASM_GLOBALIZE_LABEL (stream, name)`

A C statement (sans semicolon) to output to the stdio stream stream some commands that will make the label name global; that is, available for reference from other files. Use the expression `assemble_name (stream, name)` to output the name itself; before and after that, output the additional assembler syntax for making that name global, and a newline.

`ASM_WEAKEN_LABEL`

A C statement (sans semicolon) to output to the stdio stream stream some commands that will make the label name weak; that is, available for reference from other files but only used if no other definition is available. Use the expression `assemble_name (stream, name)` to output the name itself; before and after that, output the additional assembler syntax for making that name weak, and a newline.

If you don't define this macro, GNU CC will not support weak symbols and you should not de the SUPPORTS_WEAK macro.

SUPPORTS_WEAK

A C expression which evaluates to true if the target supports weak symbols.

If you don't define this macro, `defaults.h' provides a default definition. If ASM_WEAKEN_LABEL is defined, the default definition is `1'; otherwise, it is `0'. Define this macro if you want to control weak symbol support with a compiler flag such as `-melf'.

ASM_OUTPUT_EXTERNAL (stream, decl, name)

A C statement (sans semicolon) to output to the stdio stream stream any text necessary for declaring the name of an external symbol named name which is referenced in this compilation but not defined. The value of decl is the tree node for the declaration.

This macro need not be defined if it does not need to output anything. The GNU assembler and most Unix assemblers don't require anything.

ASM_OUTPUT_EXTERNAL_LIBCALL (stream, symref)

A C statement (sans semicolon) to output on stream an assembler pseudo-op to declare a library function name external. The name of the library function is given by symref, which has type rtx and is a symbol_ref.

This macro need not be defined if it does not need to output anything. The GNU assembler and most Unix assemblers don't require anything.

ASM_OUTPUT_LABELREF (stream, name)

A C statement (sans semicolon) to output to the stdio stream stream a reference in assembler syntax to a label named name. This should add `_' to the front of the name, if that is customary on your operating system, as it is in most Berkeley Unix systems. This macro is used in assemble_name.

ASM_OUTPUT_INTERNAL_LABEL (stream, prefix, num)

A C statement to output to the stdio stream stream a label whose name is made from the string prefix and the number num.

It is absolutely essential that these labels be distinct from the labels used for user-level functions and variables. Otherwise, certain programs will have name conflicts with internal labels.

It is desirable to exclude internal labels from the symbol table of the object file. Most assemblers have a naming convention for labels that should be excluded; on many systems, the letter `L' at the beginning of a label has this effect. You should find out what convention your system uses, and follow it.

The usual definition of this macro is as follows:

```
fprintf (stream, "L%s%d:\n", prefix, num)
```

ASM_GENERATE_INTERNAL_LABEL (string, prefix, num)

A C statement to store into the string string a label whose name is made from the string prefix and

the number num.

This string, when output subsequently by `assemble_name`, should produce the output that `ASM_OUTPUT_INTERNAL_LABEL` would produce with the same prefix and num.

If the string begins with `*', then `assemble_name` will output the rest of the string unchanged. It is often convenient for `ASM_GENERATE_INTERNAL_LABEL` to use `*' in this way. If the string doesn't start with `*', then `ASM_OUTPUT_LABELREF` gets to output the string, and may change it. (Of course, `ASM_OUTPUT_LABELREF` is also part of your machine description, so you should know what it does on your machine.)

`ASM_FORMAT_PRIVATE_NAME (outvar, name, number)`

A C expression to assign to outvar (which is a variable of type `char *`) a newly allocated string made from the string name and the number number, with some suitable punctuation added. Use `alloca` to get space for the string.

The string will be used as an argument to `ASM_OUTPUT_LABELREF` to produce an assembler label for an internal static variable whose name is name. Therefore, the string must be such as to result in valid assembler code. The argument number is different each time this macro is executed; it prevents conflicts between similarly-named internal static variables in different scopes.

Ideally this string should not be a valid C identifier, to prevent any conflict with the user's own symbols. Most assemblers allow periods or percent signs in assembler symbols; putting at least one of these between the name and the number will suffice.

`ASM_OUTPUT_DEF (stream, name, value)`

A C statement to output to the stdio stream stream assembler code which defines (equates) the symbol name to have the value value.

If SET_ASM_OP is defined, a default definition is provided which is correct for most systems.

`OBJC_GEN_METHOD_LABEL (buf, is_inst, class_name, cat_name, sel_name)`

Define this macro to override the default assembler names used for Objective C methods.

The default name is a unique method number followed by the name of the class (e.g. `_1_Foo'). For methods in categories, the name of the category is also included in the assembler name (e.g. `_1_Foo_Bar').

These names are safe on most systems, but make debugging difficult since the method's selector is not present in the name. Therefore, particular systems define other ways of computing names.

buf is an expression of type `char *` which gives you a buffer in which to store the name; its length is as long as class_name, cat_name and sel_name put together, plus 50 characters extra.

The argument is_inst specifies whether the method is an instance method or a class method; class_name is the name of the class; cat_name is the name of the category (or NULL if the method is not in a category); and sel_name is the name of the selector.

On systems where the assembler can handle quoted names, you can use this macro to provide more human-readable names.

How Initialization Functions Are Handled

The compiled code for certain languages includes constructors (also called initialization routines)---functions to initialize data in the program when the program is started. These functions need to be called before the program is "started"---that is to say, before `main` is called.

Compiling some languages generates destructors (also called termination routines) that should be called when the program terminates.

To make the initialization and termination functions work, the compiler must output something in the assembler code to cause those functions to be called at the appropriate time. When you port the compiler to a new system, you need to specify how to do this.

There are two major ways that GCC currently supports the execution of initialization and termination functions. Each way has two variants. Much of the structure is common to all four variations.

The linker must build two lists of these functions--a list of initialization functions, called `__CTOR_LIST__`, and a list of termination functions, called `__DTOR_LIST__`.

Each list always begins with an ignored function pointer (which may hold 0, -1, or a count of the function pointers after it, depending on the environment). This is followed by a series of zero or more function pointers to constructors (or destructors), followed by a function pointer containing zero.

Depending on the operating system and its executable file format, either `` `crtstuff.c' `` or `` `libgcc2.c' `` traverses these lists at startup time and exit time. Constructors are called in reverse order of the list; destructors in forward order.

The best way to handle static constructors works only for object file formats which provide arbitrarily-named sections. A section is set aside for a list of constructors, and another for a list of destructors. Traditionally these are called `.ctors' and `.dtors'. Each object file that defines an initialization function also puts a word in the constructor section to point to that function. The linker accumulates all these words into one contiguous `.ctors' section. Termination functions are handled similarly.

To use this method, you need appropriate definitions of the macros `ASM_OUTPUT_CONSTRUCTOR` and `ASM_OUTPUT_DESTRUCTOR`. Usually you can get them by including `` `svr4.h' ``.

When arbitrary sections are available, there are two variants, depending upon how the code in `` `crtstuff.c' `` is called. On systems that support an init section which is executed at program startup, parts of `` `crtstuff.c' `` are compiled into that section. The program is linked by the `gcc` driver like this:

```
ld -o output_file crtbegin.o ... crtend.o -lgcc
```

The head of a function (`__do_global_ctors`) appears in the init section of `` `crtbegin.o' ``; the remainder of the function appears in the init section of `` `crtend.o' ``. The linker will pull these two parts of the section together, making a whole function. If any of the user's object files linked into the middle of it contribute code, then that code will be executed as part of the body of `__do_global_ctors`.

To use this variant, you must define the `INIT_SECTION_ASM_OP` macro properly.

If no init section is available, do not define `INIT_SECTION_ASM_OP`. Then `__do_global_ctors` is built into the text section like all other functions, and resides in `` `libgcc.a' ``. When GCC compiles any function called `main`, it inserts a procedure call to `__main` as the first executable code after the function prologue. The `__main` function, also defined in `` `libgcc2.c' ``, simply calls `` `__do_global_ctors' ``.

In file formats that don't support arbitrary sections, there are again two variants. In the simplest variant, the GNU linker (GNU `ld`) and an `a.out' format must be used. In this case, `ASM_OUTPUT_CONSTRUCTOR` is defined to produce a `.stabs` entry of type `N_SETT', referencing the name `__CTOR_LIST__`, and with the address of the void function containing the initialization code as its value. The GNU linker recognizes this as a request to add the value to a "set"; the values are accumulated, and are eventually placed in the executable as a vector in the format described above, with a leading (ignored) count and a trailing zero element. `ASM_OUTPUT_DESTRUCTOR` is handled similarly. Since no init section is available, the absence of `INIT_SECTION_ASM_OP` causes the compilation of `main` to call `__main` as above, starting the initialization process.

The last variant uses neither arbitrary sections nor the GNU linker. This is preferable when you want to do dynamic linking and when using file formats which the GNU linker does not support, such as `ECOFF'. In this case, `ASM_OUTPUT_CONSTRUCTOR` does not produce an `N_SETT` symbol; initialization and termination functions are recognized simply by their names. This requires an extra program in the linkage step, called `collect2`. This program pretends to be the linker, for use with GNU CC; it does its job by running the ordinary linker, but also arranges to include the vectors of initialization and termination functions. These functions are called via `__main` as described above.

Choosing among these configuration options has been simplified by a set of operating-system-dependent files in the `` `config' `` subdirectory. These files define all of the relevant parameters. Usually it is sufficient to include one into your specific machine-dependent configuration file. These files are:

`` `aoutos.h' ``
: For operating systems using the `a.out' format.

`` `next.h' ``
: For operating systems using the `MachO' format.

`` `svr3.h' ``
: For System V Release 3 and similar systems using `COFF' format.

`` `svr4.h' ``
: For System V Release 4 and similar systems using `ELF' format.

`` `vms.h' ``
: For the VMS operating system.

Macros Controlling Initialization Routines

Here are the macros that control how the compiler handles initialization and termination functions:

`INIT_SECTION_ASM_OP`

If defined, a C string constant for the assembler operation to identify the following data as initialization code. If not defined, GNU CC will assume such a section does not exist. When you are using special sections for initialization and termination functions, this macro also controls how `` `crtstuff.c' `` and `` `libgcc2.c' `` arrange to run the initialization functions.

`HAS_INIT_SECTION`

If defined, `main` will not call `__main` as described above. This macro should be defined for systems that control the contents of the init section on a symbol-by-symbol basis, such as OSF/1, and should not be defined explicitly for systems that support `INIT_SECTION_ASM_OP`.

`LD_INIT_SWITCH`

If defined, a C string constant for a switch that tells the linker that the following symbol is an initialization routine.

`LD_FINI_SWITCH`

If defined, a C string constant for a switch that tells the linker that the following symbol is a finalization routine.

`INVOKE__main`

If defined, `main` will call `__main` despite the presence of `INIT_SECTION_ASM_OP`. This macro should be defined for systems where the init section is not actually run automatically, but is still useful for collecting the lists of constructors and destructors.

`ASM_OUTPUT_CONSTRUCTOR (stream, name)`

Define this macro as a C statement to output on the stream stream the assembler code to arrange to call the function named name at initialization time.

Assume that name is the name of a C function generated automatically by the compiler. This function takes no arguments. Use the function `assemble_name` to output the name name; this performs any system-specific syntactic transformations such as adding an underscore.

If you don't define this macro, nothing special is output to arrange to call the function. This is correct when the function will be called in some other manner--for example, by means of the `collect2` program, which looks through the symbol table to find these functions by their names.

`ASM_OUTPUT_DESTRUCTOR (stream, name)`

This is like `ASM_OUTPUT_CONSTRUCTOR` but used for termination functions rather than initialization functions.

If your system uses `collect2` as the means of processing constructors, then that program normally uses `nm` to scan an object file for constructor functions to be called. On certain kinds of systems, you can define these macros to make `collect2` work faster (and, in some cases, make it work at all):

`OBJECT_FORMAT_COFF`

Define this macro if the system uses COFF (Common Object File Format) object files, so that

`collect2` can assume this format and scan object files directly for dynamic constructor/destructor functions.

`OBJECT_FORMAT_ROSE`

Define this macro if the system uses ROSE format object files, so that `collect2` can assume this format and scan object files directly for dynamic constructor/destructor functions.

These macros are effective only in a native compiler; `collect2` as part of a cross compiler always uses `nm` for the target machine.

`REAL_NM_FILE_NAME`

Define this macro as a C string constant containing the file name to use to execute `nm`. The default is to search the path normally for `nm`.

If your system supports shared libraries and has a program to list the dynamic dependencies of a given library or executable, you can define these macros to enable support for running initialization and termination functions in shared libraries:

`LDD_SUFFIX`

Define this macro to a C string constant containing the name of the program which lists dynamic dependencies, like `"ldd"` under SunOS 4.

`PARSE_LDD_OUTPUT (PTR)`

Define this macro to be C code that extracts filenames from the output of the program denoted by `LDD_SUFFIX`. PTR is a variable of type `char *` that points to the beginning of a line of output from `LDD_SUFFIX`. If the line lists a dynamic dependency, the code must advance PTR to the beginning of the filename on that line. Otherwise, it must set PTR to `NULL`.

Output of Assembler Instructions

This describes assembler instruction output.

`REGISTER_NAMES`

A C initializer containing the assembler's names for the machine registers, each one as a C string constant. This is what translates register numbers in the compiler into assembler language.

`ADDITIONAL_REGISTER_NAMES`

If defined, a C initializer for an array of structures containing a name and a register number. This macro defines additional names for hard registers, thus allowing the `asm` option in declarations to refer to registers using alternate names.

`ASM_OUTPUT_OPCODE (stream, ptr)`

Define this macro if you are using an unusual assembler that requires different names for the machine instructions.

The definition is a C statement or statements which output an assembler instruction opcode to the stdio stream stream. The macro-operand ptr is a variable of type `char *` which points to the opcode name in its "internal" form--the form that is written in the machine description. The definition should output the opcode name to stream, performing any translation you desire, and increment the variable ptr to point at the end of the opcode so that it will not be output twice.

In fact, your macro definition may process less than the entire opcode name, or more than the opcode name; but if you want to process text that includes `%'-sequences to substitute operands, you must take care of the substitution yourself. Just be sure to increment ptr over whatever text should not be output normally.

If you need to look at the operand values, they can be found as the elements of `recog_operand`.

If the macro definition does nothing, the instruction is output in the usual way.

`FINAL_PRESCAN_INSN (insn, opvec, noperands)`

If defined, a C statement to be executed just prior to the output of assembler code for insn, to modify the extracted operands so they will be output differently.

Here the argument opvec is the vector containing the operands extracted from insn, and noperands is the number of elements of the vector which contain meaningful data for this insn. The contents of this vector are what will be used to convert the insn template into assembler code, so you can change the assembler output by changing the contents of the vector.

This macro is useful when various assembler syntaxes share a single file of instruction patterns; by defining this macro differently, you can cause a large class of instructions to be output differently (such as with rearranged operands). Naturally, variations in assembler syntax affecting individual insn patterns ought to be handled by writing conditional output routines in those patterns.

If this macro is not defined, it is equivalent to a null statement.

`PRINT_OPERAND (stream, x, code)`

A C compound statement to output to stdio stream stream the assembler syntax for an instruction operand x. x is an RTL expression.

code is a value that can be used to specify one of several ways of printing the operand. It is used when identical operands must be printed differently depending on the context. code comes from the `%' specification that was used to request printing of the operand. If the specification was just `%digit' then code is 0; if the specification was `%ltr digit' then code is the ASCII code for ltr.

If x is a register, this macro should print the register's name. The names can be found in an array `reg_names` whose type is `char *[]`. `reg_names` is initialized from `REGISTER_NAMES`.

When the machine description has a specification `%punct' (a `%' followed by a punctuation character), this macro is called with a null pointer for x and the punctuation character for code.

`PRINT_OPERAND_PUNCT_VALID_P (code)`

A C expression which evaluates to true if code is a valid punctuation character for use in the `PRINT_OPERAND` macro. If `PRINT_OPERAND_PUNCT_VALID_P` is not defined, it means that no punctuation characters (except for the standard one, `%') are used in this way.

`PRINT_OPERAND_ADDRESS (stream, x)`

A C compound statement to output to stdio stream stream the assembler syntax for an instruction operand that is a memory reference whose address is x. x is an RTL expression.

On some machines, the syntax for a symbolic address depends on the section that the address refers to. On these machines, define the macro `ENCODE_SECTION_INFO` to store the

information into the `symbol_ref`, and then check for it here. See section Defining the Outpu Assembler Language.

`DBR_OUTPUT_SEQEND(file)`

A C statement, to be executed after all slot-filler instructions have been output. If necessary, call `dbr_sequence_length` to determine the number of slots filled in a sequence (zero if not currently outputting a sequence), to decide how many no-ops to output, or whatever.

Don't define this macro if it has nothing to do, but it is helpful in reading assembly output if the extent of the delay sequence is made explicit (e.g. with white space).

Note that output routines for instructions with delay slots must be prepared to deal with not being output as part of a sequence (i.e. when the scheduling pass is not run, or when no slot fillers could be found.) The variable `final_sequence` is null when not processing a sequence, otherwise it contains the `sequence` rtx being output.

`REGISTER_PREFIX`

`LOCAL_LABEL_PREFIX`

`USER_LABEL_PREFIX`

`IMMEDIATE_PREFIX`

If defined, C string expressions to be used for the `%R', `%L', `%U', and `%I' options of `asm_fprintf` (see `final.c'). These are useful when a single `md' file must support multiple assembler formats. In that case, the various `tm.h' files can define these macros differently.

`ASSEMBLER_DIALECT`

If your target supports multiple dialects of assembler language (such as different opcodes), define this macro as a C expression that gives the numeric index of the assembler language dialect to use, with zero as the first variant.

If this macro is defined, you may use `{option0|option1|option2...}' constructs in the output templates of patterns (see section Output Templates and Operand Substitution) or in the first argument of `asm_fprintf`. This construct outputs `option0', `option1' or `option2', etc., if the value of `ASSEMBLER_DIALECT` is zero, one or two, etc. Any special characters within these strings retain their usual meaning.

If you do not define this macro, the characters `{', `|' and `}' do not have any special meaning when used in templates or operands to `asm_fprintf`.

Define the macros `REGISTER_PREFIX`, `LOCAL_LABEL_PREFIX`, `USER_LABEL_PREFIX` and `IMMEDIATE_PREFIX` if you can express the variations in assemble language syntax with that mechanism. Define `ASSEMBLER_DIALECT` and use the `{option0|option1}' syntax if the syntax variant are larger and involve such things as different opcodes or operand order.

`ASM_OUTPUT_REG_PUSH (stream, regno)`

A C expression to output to stream some assembler code which will push hard register number regno onto the stack. The code need not be optimal, since this macro is used only when profiling.

`ASM_OUTPUT_REG_POP (stream, regno)`

A C expression to output to stream some assembler code which will pop hard register number regno off of the stack. The code need not be optimal, since this macro is used only when profil

Output of Dispatch Tables

This concerns dispatch tables.

`ASM_OUTPUT_ADDR_DIFF_ELT (stream, value, rel)`

This macro should be provided on machines where the addresses in a dispatch table are relative to the table's own address.

The definition should be a C statement to output to the stdio stream stream an assembler pseudo-instruction to generate a difference between two labels. value and rel are the numbers of two internal labels. The definitions of these labels are output using `ASM_OUTPUT_INTERNAL_LABEL`, and they must be printed in the same way here. For example,

```
fprintf (stream, "\t.word L%d-L%d\n",
         value, rel)
```

`ASM_OUTPUT_ADDR_VEC_ELT (stream, value)`

This macro should be provided on machines where the addresses in a dispatch table are absolute.

The definition should be a C statement to output to the stdio stream stream an assembler pseudo-instruction to generate a reference to a label. value is the number of an internal label whose definition is output using `ASM_OUTPUT_INTERNAL_LABEL`. For example,

```
fprintf (stream, "\t.word L%d\n", value)
```

`ASM_OUTPUT_CASE_LABEL (stream, prefix, num, table)`

Define this if the label before a jump-table needs to be output specially. The first three arguments are the same as for `ASM_OUTPUT_INTERNAL_LABEL`; the fourth argument is the jump-table which follows (a `jump_insn` containing an `addr_vec` or `addr_diff_vec`).

This feature is used on system V to output a `swbeg` statement for the table.

If this macro is not defined, these labels are output with `ASM_OUTPUT_INTERNAL_LABEL`.

`ASM_OUTPUT_CASE_END (stream, num, table)`

Define this if something special must be output at the end of a jump-table. The definition should be a C statement to be executed after the assembler code for the table is written. It should write the appropriate code to stdio stream stream. The argument table is the jump-table insn, and num is the label-number of the preceding label.

If this macro is not defined, nothing special is output at the end of the jump-table.

Assembler Commands for Alignment

This describes commands for alignment.

`ASM_OUTPUT_ALIGN_CODE (file)`

A C expression to output text to align the location counter in the way that is desirable at a point in the code that is reached only by jumping.

This macro need not be defined if you don't want any special alignment to be done at such a time. Most machine descriptions do not currently define the macro.

`ASM_OUTPUT_LOOP_ALIGN (file)`

A C expression to output text to align the location counter in the way that is desirable at the beginning of a loop.

This macro need not be defined if you don't want any special alignment to be done at such a time. Most machine descriptions do not currently define the macro.

`ASM_OUTPUT_SKIP (stream, nbytes)`

A C statement to output to the stdio stream stream an assembler instruction to advance the location counter by nbytes bytes. Those bytes should be zero when loaded. nbytes will be a C expression of type `int`.

`ASM_NO_SKIP_IN_TEXT`

Define this macro if `ASM_OUTPUT_SKIP` should not be used in the text section because it fails put zeros in the bytes that are skipped. This is true on many Unix systems, where the pseudo--op to skip bytes produces no-op instructions rather than zeros when used in the text section.

`ASM_OUTPUT_ALIGN (stream, power)`

A C statement to output to the stdio stream stream an assembler command to advance the location counter to a multiple of 2 to the power bytes. power will be a C expression of type `int`.

Controlling Debugging Information Format

This describes how to specify debugging information.

Macros Affecting All Debugging Formats

These macros affect all debugging formats.

`DBX_REGISTER_NUMBER (regno)`

A C expression that returns the DBX register number for the compiler register number regno. In simple cases, the value of this expression may be regno itself. But sometimes there are some registers that the compiler knows about and DBX does not, or vice versa. In such cases, some register may need to have one number in the compiler and another for DBX.

If two registers have consecutive numbers inside GNU CC, and they can be used as a pair to hold a multiword value, then they *must* have consecutive numbers after renumbering with

`DBX_REGISTER_NUMBER`. Otherwise, debuggers will be unable to access such a pair, because they expect register pairs to be consecutive in their own numbering scheme.

If you find yourself defining `DBX_REGISTER_NUMBER` in way that does not preserve register pairs, then what you must do instead is redefine the actual register numbering scheme.

`DEBUGGER_AUTO_OFFSET (x)`

A C expression that returns the integer offset value for an automatic variable having address x (an RTL expression). The default computation assumes that x is based on the frame-pointer and gives the offset from the frame-pointer. This is required for targets that produce debugging output for DBX or COFF-style debugging output for SDB and allow the frame-pointer to be eliminated when the `-g' options is used.

`DEBUGGER_ARG_OFFSET (offset, x)`

A C expression that returns the integer offset value for an argument having address x (an RTL expression). The nominal offset is offset.

`PREFERRED_DEBUGGING_TYPE`

A C expression that returns the type of debugging output GNU CC produces when the user specifies `-g' or `-ggdb'. Define this if you have arranged for GNU CC to support more than one format of debugging output. Currently, the allowable values are `DBX_DEBUG`, `SDB_DEBUG`, `DWARF_DEBUG`, and `XCOFF_DEBUG`.

The value of this macro only affects the default debugging output; the user can always get a specific type of output by using `-gstabs', `-gcoff', `-gdwarf', or `-gxcoff'.

Specific Options for DBX Output

These are specific options for DBX output.

`DBX_DEBUGGING_INFO`

Define this macro if GNU CC should produce debugging output for DBX in response to the `-g' option.

`XCOFF_DEBUGGING_INFO`

Define this macro if GNU CC should produce XCOFF format debugging output in response to the `-g' option. This is a variant of DBX format.

`DEFAULT_GDB_EXTENSIONS`

Define this macro to control whether GNU CC should by default generate GDB's extended version of DBX debugging information (assuming DBX-format debugging information is enabled at all). If you don't define the macro, the default is 1: always generate the extended information if there is any occasion to.

`DEBUG_SYMS_TEXT`

Define this macro if all `.stabs` commands should be output while in the text section.

`ASM_STABS_OP`

A C string constant naming the assembler pseudo op to use instead of `.stabs` to define an ordinary debugging symbol. If you don't define this macro, `.stabs` is used. This macro applies

only to DBX debugging information format.

`ASM_STABD_OP`

A C string constant naming the assembler pseudo op to use instead of `.stabd` to define a debugging symbol whose value is the current location. If you don't define this macro, `.stabd` is used. This macro applies only to DBX debugging information format.

`ASM_STABN_OP`

A C string constant naming the assembler pseudo op to use instead of `.stabn` to define a debugging symbol with no name. If you don't define this macro, `.stabn` is used. This macro applies only to DBX debugging information format.

`DBX_NO_XREFS`

Define this macro if DBX on your system does not support the construct `xstagname'. On some systems, this construct is used to describe a forward reference to a structure named tagname. On other systems, this construct is not supported at all.

`DBX_CONTIN_LENGTH`

A symbol name in DBX-format debugging information is normally continued (split into two separate `.stabs` directives) when it exceeds a certain length (by default, 80 characters). On some operating systems, DBX requires this splitting; on others, splitting must not be done. You can inhibit splitting by defining this macro with the value zero. You can override the default splitting-length by defining this macro as an expression for the length you desire.

`DBX_CONTIN_CHAR`

Normally continuation is indicated by adding a `\' character to the end of a `.stabs` string when a continuation follows. To use a different character instead, define this macro as a character constant for the character you want to use. Do not define this macro if backslash is correct for your system.

`DBX_STATIC_STAB_DATA_SECTION`

Define this macro if it is necessary to go to the data section before outputting the `.stabs' pseudo-op for a non-global static variable.

`DBX_TYPE_DECL_STABS_CODE`

The value to use in the "code" field of the `.stabs` directive for a typedef. The default is `N_LSYM`.

`DBX_STATIC_CONST_VAR_CODE`

The value to use in the "code" field of the `.stabs` directive for a static variable located in the text section. DBX format does not provide any "right" way to do this. The default is `N_FUN`.

`DBX_REGPARM_STABS_CODE`

The value to use in the "code" field of the `.stabs` directive for a parameter passed in registers. DBX format does not provide any "right" way to do this. The default is `N_RSYM`.

`DBX_REGPARM_STABS_LETTER`

The letter to use in DBX symbol data to identify a symbol as a parameter passed in registers. DBX format does not customarily provide any way to do this. The default is `'P'`.

`DBX_MEMPARM_STABS_LETTER`

The letter to use in DBX symbol data to identify a symbol as a stack parameter. The default is

'p'.

`DBX_FUNCTION_FIRST`

Define this macro if the DBX information for a function and its arguments should precede the assembler code for the function. Normally, in DBX format, the debugging information entirely follows the assembler code.

`DBX_LBRAC_FIRST`

Define this macro if the `N_LBRAC` symbol for a block should precede the debugging information for variables and functions defined in that block. Normally, in DBX format, the `N_LBRAC` symbol comes first.

`DBX_BLOCKS_FUNCTION_RELATIVE`

Define this macro if the value of a symbol describing the scope of a block (`N_LBRAC` or `N_RBRAC`) should be relative to the start of the enclosing function. Normally, GNU C uses an absolute address.

Open-Ended Hooks for DBX Format

These are hooks for DBX format.

`DBX_OUTPUT_LBRAC (stream, name)`

Define this macro to say how to output to stream the debugging information for the start of a scope level for variable names. The argument name is the name of an assembler symbol (for use with `assemble_name`) whose value is the address where the scope begins.

`DBX_OUTPUT_RBRAC (stream, name)`

Like `DBX_OUTPUT_LBRAC`, but for the end of a scope level.

`DBX_OUTPUT_ENUM (stream, type)`

Define this macro if the target machine requires special handling to output an enumeration type. The definition should be a C statement (sans semicolon) to output the appropriate information to stream for the type type.

`DBX_OUTPUT_FUNCTION_END (stream, function)`

Define this macro if the target machine requires special output at the end of the debugging information for a function. The definition should be a C statement (sans semicolon) to output the appropriate information to stream. function is the `FUNCTION_DECL` node for the function.

`DBX_OUTPUT_STANDARD_TYPES (syms)`

Define this macro if you need to control the order of output of the standard data types at the beginning of compilation. The argument syms is a `tree` which is a chain of all the predefined global symbols, including names of data types.

Normally, DBX output starts with definitions of the types for integers and characters, followed by all the other predefined types of the particular language in no particular order.

On some machines, it is necessary to output different particular types first. To do this, define `DBX_OUTPUT_STANDARD_TYPES` to output those symbols in the necessary order. Any predefined types that you don't explicitly output will be output afterward in no particular order.

Be careful not to define this macro so that it works only for C. There are no global variables to access most of the built-in types, because another language may have another set of types. The way to output a particular type is to look through syms to see if you can find it. Here is an example:

```
{
  tree decl;
  for (decl = syms; decl; decl = TREE_CHAIN (decl))
    if (!strcmp (IDENTIFIER_POINTER (DECL_NAME (decl)),
                 "long int"))
      dbxout_symbol (decl);
  ...
}
```

This does nothing if the expected type does not exist.

See the function `init_decl_processing` in `c-decl.c' to find the names to use for all the built-in C types.

Here is another way of finding a particular type:

```
{
  tree decl;
  for (decl = syms; decl; decl = TREE_CHAIN (decl))
    if (TREE_CODE (decl) == TYPE_DECL
        && (TREE_CODE (TREE_TYPE (decl))
            == INTEGER_CST)
        && TYPE_PRECISION (TREE_TYPE (decl)) == 16
        && TYPE_UNSIGNED (TREE_TYPE (decl)))
      /* This must be unsigned short.  */
      dbxout_symbol (decl);
  ...
}
```

File Names in DBX Format

This describes file names in DBX format.

`DBX_WORKING_DIRECTORY`

Define this if DBX wants to have the current directory recorded in each object file.

Note that the working directory is always recorded if GDB extensions are enabled.

`DBX_OUTPUT_MAIN_SOURCE_FILENAME (stream, name)`

A C statement to output DBX debugging information to the stdio stream stream which indicates that file name is the main source file--the file specified as the input file for compilation. This macro is called only once, at the beginning of compilation.

This macro need not be defined if the standard form of output for DBX debugging information appropriate.

`DBX_OUTPUT_MAIN_SOURCE_DIRECTORY (stream, name)`

A C statement to output DBX debugging information to the stdio stream stream which indicates that the current directory during compilation is named name.

This macro need not be defined if the standard form of output for DBX debugging information is appropriate.

`DBX_OUTPUT_MAIN_SOURCE_FILE_END (stream, name)`

A C statement to output DBX debugging information at the end of compilation of the main source file name.

If you don't define this macro, nothing special is output at the end of compilation, which is correct for most machines.

`DBX_OUTPUT_SOURCE_FILENAME (stream, name)`

A C statement to output DBX debugging information to the stdio stream stream which indicates that file name is the current source file. This output is generated each time input shifts to a different source file as a result of `#include', the end of an included file, or a `#line' command.

This macro need not be defined if the standard form of output for DBX debugging information is appropriate.

Macros for SDB and DWARF Output

Here are macros for SDB and DWARF output.

`SDB_DEBUGGING_INFO`

Define this macro if GNU CC should produce COFF-style debugging output for SDB in response to the `-g' option.

`DWARF_DEBUGGING_INFO`

Define this macro if GNU CC should produce dwarf format debugging output in response to the `-g' option.

`PUT_SDB_...`

Define these macros to override the assembler syntax for the special SDB assembler directives. See `sdbout.c' for a list of these macros and their arguments. If the standard syntax is used, you need not define them yourself.

`SDB_DELIM`

Some assemblers do not support a semicolon as a delimiter, even between SDB assembler directives. In that case, define this macro to be the delimiter to use (usually `\n'). It is not necessary to define a new set of `PUT_SDB_op` macros if this is the only change required.

`SDB_GENERATE_FAKE`

Define this macro to override the usual method of constructing a dummy name for anonymous structure and union types. See `sdbout.c' for more information.

`SDB_ALLOW_UNKNOWN_REFERENCES`

Define this macro to allow references to unknown structure, union, or enumeration tags to be emitted. Standard COFF does not allow handling of unknown references, MIPS ECOFF has support for it.

`SDB_ALLOW_FORWARD_REFERENCES`

Define this macro to allow references to structure, union, or enumeration tags that have not yet been seen to be handled. Some assemblers choke if forward tags are used, while some require it.

Cross Compilation and Floating Point

While all modern machines use 2's complement representation for integers, there are a variety of representations for floating point numbers. This means that in a cross-compiler the representation of floating point numbers in the compiled program may be different from that used in the machine doing the compilation.

Because different representation systems may offer different amounts of range and precision, the cross compiler cannot safely use the host machine's floating point arithmetic. Therefore, floating point constants must be represented in the target machine's format. This means that the cross compiler cannot use `atof` to parse a floating point constant; it must have its own special routine to use instead. Also, constant folding must emulate the target machine's arithmetic (or must not be done at all).

The macros in the following table should be defined only if you are cross compiling between different floating point formats.

Otherwise, don't define them. Then default definitions will be set up which use `double` as the data type, `==` to test for equality, etc.

You don't need to worry about how many times you use an operand of any of these macros. The compiler never uses operands which have side effects.

`REAL_VALUE_TYPE`

A macro for the C data type to be used to hold a floating point value in the target machine's format. Typically this would be a `struct` containing an array of `int`.

`REAL_VALUES_EQUAL (x, y)`

A macro for a C expression which compares for equality the two values, x and y, both of type `REAL_VALUE_TYPE`.

`REAL_VALUES_LESS (x, y)`

A macro for a C expression which tests whether x is less than y, both values being of type `REAL_VALUE_TYPE` and interpreted as floating point numbers in the target machine's representation.

`REAL_VALUE_LDEXP (x, scale)`

A macro for a C expression which performs the standard library function `ldexp`, but using the target machine's floating point representation. Both x and the value of the expression have type `REAL_VALUE_TYPE`. The second argument, scale, is an integer.

`REAL_VALUE_FIX (x)`

A macro whose definition is a C expression to convert the target-machine floating point value x to a signed integer. x has type `REAL_VALUE_TYPE`.

`REAL_VALUE_UNSIGNED_FIX (x)`

A macro whose definition is a C expression to convert the target-machine floating point value x to an unsigned integer. x has type `REAL_VALUE_TYPE`.

`REAL_VALUE_RNDZINT (x)`

A macro whose definition is a C expression to round the target-machine floating point value x towards zero to an integer value (but still as a floating point number). x has type `REAL_VALUE_TYPE`, and so does the value.

`REAL_VALUE_UNSIGNED_RNDZINT (x)`

A macro whose definition is a C expression to round the target-machine floating point value x towards zero to an unsigned integer value (but still represented as a floating point number). x has type `REAL_VALUE_TYPE`, and so does the value.

`REAL_VALUE_ATOF (string, mode)`

A macro for a C expression which converts string, an expression of type `char *`, into a floating point number in the target machine's representation for mode mode. The value has type `REAL_VALUE_TYPE`.

`REAL_INFINITY`

Define this macro if infinity is a possible floating point value, and therefore division by 0 is legitimate.

`REAL_VALUE_ISINF (x)`

A macro for a C expression which determines whether x, a floating point value, is infinity. The value has type `int`. By default, this is defined to call `isinf`.

`REAL_VALUE_ISNAN (x)`

A macro for a C expression which determines whether x, a floating point value, is a "nan" (not-a-number). The value has type `int`. By default, this is defined to call `isnan`.

Define the following additional macros if you want to make floating point constant folding work while cross compiling. If you don't define them, cross compilation is still possible, but constant folding will not happen for floating point values.

`REAL_ARITHMETIC (output, code, x, y)`

A macro for a C statement which calculates an arithmetic operation of the two floating point values x and y, both of type `REAL_VALUE_TYPE` in the target machine's representation, to produce a result of the same type and representation which is stored in output (which will be a variable).

The operation to be performed is specified by code, a tree code which will always be one of the following: `PLUS_EXPR`, `MINUS_EXPR`, `MULT_EXPR`, `RDIV_EXPR`, `MAX_EXPR`, `MIN_EXPR`.

The expansion of this macro is responsible for checking for overflow. If overflow happens, the macro expansion should execute the statement `return 0;`, which indicates the inability to

perform the arithmetic operation requested.

`REAL_VALUE_NEGATE (x)`

A macro for a C expression which returns the negative of the floating point value x. Both x and the value of the expression have type `REAL_VALUE_TYPE` and are in the target machine's floating point representation.

There is no way for this macro to report overflow, since overflow can't happen in the negation operation.

`REAL_VALUE_TRUNCATE (mode, x)`

A macro for a C expression which converts the floating point value x to mode mode.

Both x and the value of the expression are in the target machine's floating point representation and have type `REAL_VALUE_TYPE`. However, the value should have an appropriate bit pattern to be output properly as a floating constant whose precision accords with mode mode.

There is no way for this macro to report overflow.

`REAL_VALUE_TO_INT (low, high, x)`

A macro for a C expression which converts a floating point value x into a double-precision integer which is then stored into low and high, two variables of type int.

`REAL_VALUE_FROM_INT (x, low, high)`

A macro for a C expression which converts a double-precision integer found in low and high, two variables of type int, into a floating point value which is then stored into x.

Miscellaneous Parameters

Here are several miscellaneous parameters.

`PREDICATE_CODES`

Define this if you have defined special-purpose predicates in the file `` `machine.c' ``. This macro is called within an initializer of an array of structures. The first field in the structure is the name of a predicate and the second field is an array of rtl codes. For each predicate, list all rtl codes that can be in expressions matched by the predicate. The list should have a trailing comma. Here is an example of two entries in the list for a typical RISC machine:

```
#define PREDICATE_CODES \
  {"gen_reg_rtx_operand", {SUBREG, REG}},  \
  {"reg_or_short_cint_operand", {SUBREG, REG, CONST_INT}},
```

Defining this macro does not affect the generated code (however, incorrect definitions that omit an rtl code that may be matched by the predicate can cause the compiler to malfunction). Instead, it allows the table built by `` `genrecog' `` to be more compact and efficient, thus speeding up the compiler. The most important predicates to include in the list specified by this macro are thoses used in the most insn patterns.

`CASE_VECTOR_MODE`

An alias for a machine mode name. This is the machine mode that elements of a jump-table sh have.

`CASE_VECTOR_PC_RELATIVE`

Define this macro if jump-tables should contain relative addresses.

`CASE_DROPS_THROUGH`

Define this if control falls through a `case` insn when the index value is out of range. This means the specified default-label is actually ignored by the `case` insn proper.

`CASE_VALUES_THRESHOLD`

Define this to be the smallest number of different values for which it is best to use a jump-table instead of a tree of conditional branches. The default is four for machines with a `casesi` instruction and five otherwise. This is best for most machines.

`WORD_REGISTER_OPERATIONS`

Define this macro if operations between registers with integral mode smaller than a word are always performed on the entire register. Most RISC machines have this property and most CISC machines do not.

`LOAD_EXTEND_OP (mode)`

Define this macro to be a C expression indicating when insns that read memory in mode, an integral mode narrower than a word, set the bits outside of mode to be either the sign-extension or the zero-extension of the data read. Return `SIGN_EXTEND` for values of mode for which the insn sign-extends, `ZERO_EXTEND` for which it zero-extends, and `NIL` for other modes.

This macro is not called with mode non-integral or with a width greater than or equal to `BITS_PER_WORD`, so you may return any value in this case. Do not define this macro if it would always return `NIL`. On machines where this macro is defined, you will normally define it as the constant `SIGN_EXTEND` or `ZERO_EXTEND`.

`IMPLICIT_FIX_EXPR`

An alias for a tree code that should be used by default for conversion of floating point values to fixed point. Normally, `FIX_ROUND_EXPR` is used.

`FIXUNS_TRUNC_LIKE_FIX_TRUNC`

Define this macro if the same instructions that convert a floating point number to a signed fixed point number also convert validly to an unsigned one.

`EASY_DIV_EXPR`

An alias for a tree code that is the easiest kind of division to compile code for in the general case. It may be `TRUNC_DIV_EXPR`, `FLOOR_DIV_EXPR`, `CEIL_DIV_EXPR` or `ROUND_DIV_EXPR`. These four division operators differ in how they round the result to an integer. `EASY_DIV_EXPR` is used when it is permissible to use any of those kinds of division and the choice should be made on the basis of efficiency.

`MOVE_MAX`

The maximum number of bytes that a single instruction can move quickly from memory to memory.

`MAX_MOVE_MAX`

The maximum number of bytes that a single instruction can move quickly from memory to memory. If this is undefined, the default is `MOVE_MAX`. Otherwise, it is the constant value that the largest value that `MOVE_MAX` can have at run-time.

`SHIFT_COUNT_TRUNCATED`

A C expression that is nonzero if on this machine the number of bits actually used for the count of a shift operation is equal to the number of bits needed to represent the size of the object being shifted. When this macro is non-zero, the compiler will assume that it is safe to omit a sign-extend, zero-extend, and certain bitwise `and' instructions that truncates the count of a shift operation. On machines that have instructions that act on bitfields at variable positions, which may include `bit test' instructions, a nonzero `SHIFT_COUNT_TRUNCATED` also enables deletion of truncations of the values that serve as arguments to bitfield instructions.

If both types of instructions truncate the count (for shifts) and position (for bitfield operations), or if no variable-position bitfield instructions exist, you should define this macro.

However, on some machines, such as the 80386 and the 680x0, truncation only applies to shift operations and not the (real or pretended) bitfield operations. Define `SHIFT_COUNT_TRUNCATED` to be zero on such machines. Instead, add patterns to the `md' file that include the implied truncation of the shift instructions.

You need not define this macro if it would always have the value of zero.

`TRULY_NOOP_TRUNCATION (outprec, inprec)`

A C expression which is nonzero if on this machine it is safe to "convert" an integer of inprec bits to one of outprec bits (where outprec is smaller than inprec) by merely operating on it as if it had only outprec bits.

On many machines, this expression can be 1.

When `TRULY_NOOP_TRUNCATION` returns 1 for a pair of sizes for modes for which `MODES_TIEABLE_P` is 0, suboptimal code can result. If this is the case, making `TRULY_NOOP_TRUNCATION` return 0 in such cases may improve things.

`STORE_FLAG_VALUE`

A C expression describing the value returned by a comparison operator with an integral mode and stored by a store-flag instruction (`scond') when the condition is true. This description must apply to *all* the `scond' patterns and all the comparison operators whose results have a `MODE_INT` mode.

A value of 1 or -1 means that the instruction implementing the comparison operator returns exactly 1 or -1 when the comparison is true and 0 when the comparison is false. Otherwise, the value indicates which bits of the result are guaranteed to be 1 when the comparison is true. This value is interpreted in the mode of the comparison operation, which is given by the mode of the first operand in the `scond' pattern. Either the low bit or the sign bit of `STORE_FLAG_VALUE` be on. Presently, only those bits are used by the compiler.

If `STORE_FLAG_VALUE` is neither 1 or -1, the compiler will generate code that depends only on the specified bits. It can also replace comparison operators with equivalent operations if they cause the required bits to be set, even if the remaining bits are undefined. For example, on a machine whose comparison operators return an `SImode` value and where `STORE_FLAG_VALUE` is

defined as `0x80000000', saying that just the sign bit is relevant, the expression

```
(ne:SI (and:SI x (const_int power-of-2)) (const_int 0))
```

can be converted to

```
(ashift:SI x (const_int n))
```

where n is the appropriate shift count to move the bit being tested into the sign bit.

There is no way to describe a machine that always sets the low-order bit for a true value, but does not guarantee the value of any other bits, but we do not know of any machine that has such an instruction. If you are trying to port GNU CC to such a machine, include an instruction to perform a logical-and of the result with 1 in the pattern for the comparison operators and let us know (see section How to Report Bugs).

Often, a machine will have multiple instructions that obtain a value from a comparison (or the condition codes). Here are rules to guide the choice of value for `STORE_FLAG_VALUE`, and hence the instructions to be used:

> Use the shortest sequence that yields a valid definition for `STORE_FLAG_VALUE`. It is more efficient for the compiler to "normalize" the value (convert it to, e.g., 1 or 0) than for the comparison operators to do so because there may be opportunities to combine the normalization with other operations.
>
> For equal-length sequences, use a value of 1 or -1, with -1 being slightly preferred on machines with expensive jumps and 1 preferred on other machines.
>
> As a second choice, choose a value of `0x80000001' if instructions exist that set both the sign and low-order bits but do not define the others.
>
> Otherwise, use a value of `0x80000000'.

Many machines can produce both the value chosen for `STORE_FLAG_VALUE` and its negation in the same number of instructions. On those machines, you should also define a pattern for those cases, e.g., one matching

```
(set A (neg:m (ne:m B C)))
```

Some machines can also perform `and` or `plus` operations on condition code values with less instructions than the corresponding `scond' insn followed by `and` or `plus`. On those machines, define the appropriate patterns. Use the names `incscc` and `decscc`, respectively, for the the patterns which perform `plus` or `minus` operations on condition code values. See `` `rs6000.md' `` for some examples. The GNU Superoptizer can be used to find such instruction sequences on other machines.

You need not define `STORE_FLAG_VALUE` if the machine has no store-flag instructions.

`FLOAT_STORE_FLAG_VALUE`

A C expression that gives a non-zero floating point value that is returned when comparison operators with floating-point results are true. Define this macro on machine that have comparison

operations that return floating-point values. If there are no such operations, do not define this macro.

`Pmode`

An alias for the machine mode for pointers. On most machines, define this to be the integer mode corresponding to the width of a hardware pointer; `SImode` on 32-bit machine or `DImode` on 64-bit machines. On some machines you must define this to be one of the partial integer modes, such as `PSImode`.

The width of `Pmode` must be at least as large as the value of `POINTER_SIZE`. If it is not equal, you must define the macro `POINTERS_EXTEND_UNSIGNED` to specify how pointers are extended to `Pmode`.

`FUNCTION_MODE`

An alias for the machine mode used for memory references to functions being called, in `call` RTL expressions. On most machines this should be `QImode`.

`INTEGRATE_THRESHOLD (decl)`

A C expression for the maximum number of instructions above which the function decl should not be inlined. decl is a `FUNCTION_DECL` node.

The default definition of this macro is 64 plus 8 times the number of arguments that the function accepts. Some people think a larger threshold should be used on RISC machines.

`SCCS_DIRECTIVE`

Define this if the preprocessor should ignore `#sccs` directives and print no error message.

`NO_IMPLICIT_EXTERN_C`

Define this macro if the system header files support C++ as well as C. This macro inhibits the usual method of using system header files in C++, which is to pretend that the file's contents are enclosed in `extern "C" {...}'.

`HANDLE_PRAGMA (stream)`

Define this macro if you want to implement any pragmas. If defined, it should be a C statement to be executed when `#pragma` is seen. The argument stream is the stdio input stream from which the source text can be read.

It is generally a bad idea to implement new uses of `#pragma`. The only reason to define this macro is for compatibility with other compilers that do support `#pragma` for the sake of any user programs which already use it.

`VALID_MACHINE_DECL_ATTRIBUTE (decl, attributes, identifier, args)`

If defined, a C expression whose value is nonzero if identifier with arguments args is a valid machine specific attribute for decl. The attributes in attributes have previously been assigned to decl.

`VALID_MACHINE_TYPE_ATTRIBUTE (type, attributes, identifier, args)`

If defined, a C expression whose value is nonzero if identifier with arguments args is a valid machine specific attribute for type. The attributes in attributes have previously been assigned to type.

`COMP_TYPE_ATTRIBUTES (type1, type2)`

If defined, a C expression whose value is zero if the attributes on type1 and type2 are incompatible, one if they are compatible, and two if they are nearly compatible (which causes a warning to be generated).

`SET_DEFAULT_TYPE_ATTRIBUTES (type)`

If defined, a C statement that assigns default attributes to newly defined type.

`DOLLARS_IN_IDENTIFIERS`

Define this macro to control use of the character `$' in identifier names. The value should be 0, 1, or 2. 0 means `$' is not allowed by default; 1 means it is allowed by default if `-traditional' is used; 2 means it is allowed by default provided `-ansi' is not used. 1 is the default; there is no need to define this macro in that case.

`NO_DOLLAR_IN_LABEL`

Define this macro if the assembler does not accept the character `$' in label names. By default constructors and destructors in G++ have `$' in the identifiers. If this macro is defined, `.' is used instead.

`NO_DOT_IN_LABEL`

Define this macro if the assembler does not accept the character `.' in label names. By default constructors and destructors in G++ have names that use `.'. If this macro is defined, these names are rewritten to avoid `.'.

`DEFAULT_MAIN_RETURN`

Define this macro if the target system expects every program's `main` function to return a standard "success" value by default (if no other value is explicitly returned).

The definition should be a C statement (sans semicolon) to generate the appropriate rtl instructions. It is used only when compiling the end of `main`.

`HAVE_ATEXIT`

Define this if the target system supports the function `atexit` from the ANSI C standard. If this is not defined, and `INIT_SECTION_ASM_OP` is not defined, a default `exit` function will be provided to support C++.

`EXIT_BODY`

Define this if your `exit` function needs to do something besides calling an external function `_cleanup` before terminating with `_exit`. The `EXIT_BODY` macro is only needed if netiher `HAVE_ATEXIT` nor `INIT_SECTION_ASM_OP` are defined.

`INSN_SETS_ARE_DELAYED (insn)`

Define this macro as a C expression that is nonzero if it is safe for the delay slot scheduler to place instructions in the delay slot of insn, even if they appear to use a resource set or clobbered in insn. insn is always a `jump_insn` or an `insn`; GNU CC knows that every `call_insn` has this behavior. On machines where some `insn` or `jump_insn` is really a function call and hence has this behavior, you should define this macro.

You need not define this macro if it would always return zero.

`INSN_REFERENCES_ARE_DELAYED (insn)`

Define this macro as a C expression that is nonzero if it is safe for the delay slot scheduler to place instructions in the delay slot of insn, even if they appear to set or clobber a resource referenced in insn. insn is always a `jump_insn` or an `insn`. On machines where some `insn` or `jump_insn` is really a function call and its operands are registers whose use is actually in the subroutine it calls, you should define this macro. Doing so allows the delay slot scheduler to move instructions which copy arguments into the argument registers into the delay slot of insn.

You need not define this macro if it would always return zero.

`MACHINE_DEPENDENT_REORG (insn)`

In rare cases, correct code generation requires extra machine dependent processing between the second jump optimization pass and delayed branch scheduling. On those machines, define this macro as a C statement to act on the code starting at insn.

Go to the previous, next section.

Go to the previous, next section.

The Configuration File

The configuration file `xm-machine.h' contains macro definitions that describe the machine and system on which the compiler is running, unlike the definitions in `machine.h', which describe the machine for which the compiler is producing output. Most of the values in `xm-machine.h' are actually the same on all machines that GNU CC runs on, so large parts of all configuration files are identical. But there are some macros that vary:

`USG`

Define this macro if the host system is System V.

`VMS`

Define this macro if the host system is VMS.

`FATAL_EXIT_CODE`

A C expression for the status code to be returned when the compiler exits after serious errors.

`SUCCESS_EXIT_CODE`

A C expression for the status code to be returned when the compiler exits without serious errors.

`HOST_WORDS_BIG_ENDIAN`

Defined if the host machine stores words of multi-word values in big-endian order. (GNU CC does not depend on the host byte ordering within a word.)

`HOST_FLOAT_WORDS_BIG_ENDIAN`

Define this macro to be 1 if the host machine stores `DFmode`, `XFmode` or `TFmode` floating point numbers in memory with the word containing the sign bit at the lowest address; otherwise, define it to be zero.

This macro need not be defined if the ordering is the same as for multi-word integers.

`HOST_FLOAT_FORMAT`

A numeric code distinguishing the floating point format for the host machine. See `TARGET_FLOAT_FORMAT` in section Storage Layout for the alternatives and default.

`HOST_BITS_PER_CHAR`

A C expression for the number of bits in `char` on the host machine.

`HOST_BITS_PER_SHORT`

A C expression for the number of bits in `short` on the host machine.

`HOST_BITS_PER_INT`

A C expression for the number of bits in `int` on the host machine.

`HOST_BITS_PER_LONG`

A C expression for the number of bits in `long` on the host machine.

`ONLY_INT_FIELDS`

Define this macro to indicate that the host compiler only supports `int` bit fields, rather than other integral types, including `enum`, as do most C compilers.

`OBSTACK_CHUNK_SIZE`

A C expression for the size of ordinary obstack chunks. If you don't define this, a usually-reasonable default is used.

`OBSTACK_CHUNK_ALLOC`

The function used to allocate obstack chunks. If you don't define this, `xmalloc` is used.

`OBSTACK_CHUNK_FREE`

The function used to free obstack chunks. If you don't define this, `free` is used.

`USE_C_ALLOCA`

Define this macro to indicate that the compiler is running with the `alloca` implemented in C. This version of `alloca` can be found in the file `` `alloca.c' ``; to use it, you must also alter the `` `Makefile' `` variable `ALLOCA`. (This is done automatically for the systems on which we know it is needed.)

If you do define this macro, you should probably do it as follows:

```
#ifndef __GNUC__
#define USE_C_ALLOCA
#else
#define alloca __builtin_alloca
#endif
```

so that when the compiler is compiled with GNU CC it uses the more efficient built-in `alloca` function.

`FUNCTION_CONVERSION_BUG`

Define this macro to indicate that the host compiler does not properly handle converting a function value to a pointer-to-function when it is used in an expression.

`HAVE_VPRINTF`

Define this if the library function `vprintf` is available on your system.

`MULTIBYTE_CHARS`

Define this macro to enable support for multibyte characters in the input to GNU CC. This requires that the host system support the ANSI C library functions for converting multibyte characters to wide characters.

`HAVE_PUTENV`

Define this if the library function `putenv` is available on your system.

`POSIX`

Define this if your system is POSIX.1 compliant.

`NO_SYS_SIGLIST`

Define this if your system *does not* provide the variable `sys_siglist`.

`DONT_DECLARE_SYS_SIGLIST`

Define this if your system has the variable `sys_siglist`, and there is already a declaration in the system header files.

`USE_PROTOTYPES`

Define this to be 1 if you know that the host compiler supports prototypes, even if it doesn't define __STDC__, or define it to be 0 if you do not want any prototypes used in compiling GNU CC. If `USE_PROTOTYPES' is not defined, it will be determined automatically whether your compiler supports prototypes by checking if `__STDC__' is defined.

`NO_MD_PROTOTYPES`

Define this if you wish suppression of prototypes generated from the machine description file, but to use other prototypes within GNU CC. If `USE_PROTOTYPES' is defined to be 0, or the host compiler does not support prototypes, this macro has no effect.

`MD_CALL_PROTOTYPES`

Define this if you wish to generate prototypes for the `gen_call` or `gen_call_value` functions generated from the machine description file. If `USE_PROTOTYPES' is defined to be 0, or the host compiler does not support prototypes, or `NO_MD_PROTOTYPES' is defined, this macro has no effect. As soon as all of the machine descriptions are modified to have the appropriate number of arguments, this macro will be removed.

Some systems do provide this variable, but with a different name such as `_sys_siglist`. On these systems, you can define `sys_siglist` as a macro which expands into the name actually provided.

`NO_STAB_H`

Define this if your system does not have the include file `` `stab.h' ``. If `USG' is defined, `NO_STAB_H' is assumed.

`PATH_SEPARATOR`

Define this macro to be a C character constant representing the character used to separate components in paths. The default value is. the colon character

`DIR_SEPARATOR`

If your system uses some character other than slash to separate directory names within a file specification, define this macro to be a C character constant specifying that character. When GNU CC displays file names, the character you specify will be used. GNU CC will test for both slash and the character you specify when parsing filenames.

`OBJECT_SUFFIX`

Define this macro to be a C string representing the suffix for object files on your machine. If you do not define this macro, GNU CC will use `.o' as the suffix for object files.

`EXECUTABLE_SUFFIX`

Define this macro to be a C string representing the suffix for executable files on your machine. If you do not define this macro, GNU CC will use the null string as the suffix for object files.

`COLLECT_EXPORT_LIST`

If defined, `collect2` will scan the individual object files specified on its command line and create an export list for the linker. Define this macro for systems like AIX, where the linker

discards object files that are not referenced from `main` and uses export lists.

In addition, configuration files for system V define `bcopy`, `bzero` and `bcmp` as aliases. Some files define `alloca` as a macro when compiled with GNU CC, in order to take advantage of the benefit of GNU CC's built-in `alloca`.

Go to the previous, next section.

Go to the previous section.

Index

!

#

$

%

&

'

(

*

+

-

/

0

<

=

>

?

\

_

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

S

t

u

V

W

X

Z

Go to the previous section.

Using and Porting GNU CC

(1)

Prior to release 2 of the compiler, there was a separate g++ compiler. That version was based on GNU CC, but not integrated with it. Versions of g++ with a `1.xx' version number--for example, g++ version 1.37 or 1.42--are much less reliable than the versions integrated with GCC 2. Moreover, combining G++ `1.xx' with a version 2 GCC will simply not work.

(2)

The analogous feature in Fortran is called an assigned goto, but that name seems inappropriate in C, where one can do more than simply store label addresses in label variables.

(3)

A file's basename was the name stripped of all leading path information and of trailing suffixes, such as `.h' or `.C' or `.cc'.

Debugging with GDB

The GNU Source-Level Debugger

Seventh Edition, for GDB version 4.18

February 1999

Richard M. Stallman and Roland H. Pesch

This file describes GDB, the GNU symbolic debugger.

This is the Seventh Edition, February 1999, for GDB Version 4.18.

This document was generated on 20 May 1999 using the texi2html translator version 1.51a.

Go to the first, previous, next, last section, table of contents.

Summary of GDB

The purpose of a debugger such as GDB is to allow you to see what is going on "inside" another program while it executes--or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

You can use GDB to debug programs written in C or C++. For more information, see section C and C++.

Support for Modula-2 and Chill is partial. For information on Modula-2, see section Modula-2. There is no further documentation on Chill yet.

Debugging Pascal programs which use sets, subranges, file variables, or nested functions does not currently work. GDB does not support entering expressions, printing values, or similar features using Pascal syntax.

GDB can be used to debug programs written in Fortran, although it does not yet support entering expressions, printing values, or similar features using Fortran syntax. It may be necessary to refer to some variables with a trailing underscore.

Free software

GDB is **free software**, protected by the GNU General Public License (GPL). The GPL gives you the freedom to copy or adapt a licensed program--but every person getting a copy also gets with it the freedom to modify that copy (which means that they must get access to the source code), and the freedom to distribute further copies. Typical software companies use copyrights to limit your freedoms; the Free Software Foundation uses the GPL to preserve these freedoms.

Fundamentally, the General Public License is a license which says that you have these freedoms and that you cannot take these freedoms away from anyone else.

Contributors to GDB

Richard Stallman was the original author of GDB, and of many other GNU programs. Many others have contributed to its development. This section attempts to credit major contributors. One of the virtues of free software is that everyone is free to contribute to it; with regret, we cannot actually acknowledge everyone here. The file `ChangeLog' in the GDB distribution approximates a blow-by-blow account.

Changes much prior to version 2.0 are lost in the mists of time.

> *Plea:* Additions to this section are particularly welcome. If you or your friends (or enemies, to be evenhanded) have been unfairly omitted from this list, we would like to add your names!

So that they may not regard their many labors as thankless, we particularly thank those who shepherded GDB through major releases: Jim Blandy (release 4.18); Jason Molenda (release 4.17); Stan Shebs (release 4.14); Fred Fish (releases 4.16, 4.15, 4.13, 4.12, 4.11, 4.10, and 4.9); Stu Grossman and John Gilmore (releases 4.8, 4.7, 4.6, 4.5, and 4.4); John Gilmore (releases 4.3, 4.2, 4.1, 4.0, and 3.9); Jim Kingdon (releases 3.5, 3.4, and 3.3); and Randy Smith (releases 3.2, 3.1, and 3.0).

Richard Stallman, assisted at various times by Peter TerMaat, Chris Hanson, and Richard Mlynarik, handled releases through 2.8.

Michael Tiemann is the author of most of the GNU C++ support in GDB, with significant additional contributions from Per Bothner. James Clark wrote the GNU C++ demangler. Early work on C++ was by Peter TerMaat (who also did much general update work leading to release 3.0).

GDB 4 uses the BFD subroutine library to examine multiple object-file formats; BFD was a joint project of David V. Henkel-Wallace, Rich Pixley, Steve Chamberlain, and John Gilmore.

David Johnson wrote the original COFF support; Pace Willison did the original support for encapsulated COFF.

Brent Benson of Harris Computer Systems contributed DWARF 2 support.

Adam de Boor and Bradley Davis contributed the ISI Optimum V support. Per Bothner, Noboyuki Hikichi, and Alessandro Forin contributed MIPS support. Jean-Daniel Fekete contributed Sun 386i support. Chris Hanson improved the HP9000 support. Noboyuki Hikichi and Tomoyuki Hasei contributed Sony/News OS 3 support. David Johnson contributed Encore Umax support. Jyrki Kuoppala contributed Altos 3068 support. Jeff Law contributed HP PA and SOM support. Keith Packard contributed NS32K support. Doug Rabson contributed Acorn Risc Machine support. Bob Rusk contributed Harris Nighthawk CX-UX support. Chris Smith contributed Convex support (and Fortran debugging). Jonathan Stone contributed Pyramid support. Michael Tiemann contributed SPARC support. Tim Tucker contributed support for the Gould NP1 and Gould Powernode. Pace Willison contributed Intel 386 support. Jay Vosburgh contributed Symmetry support.

Andreas Schwab contributed M68K Linux support.

Rich Schaefer and Peter Schauer helped with support of SunOS shared libraries.

Jay Fenlason and Roland McGrath ensured that GDB and GAS agree about several machine instruction sets.

Patrick Duval, Ted Goldstein, Vikram Koka and Glenn Engel helped develop remote debugging. Intel Corporation, Wind River Systems, AMD, and ARM contributed remote debugging modules for the i960, VxWorks, A29K UDI, and RDI targets, respectively.

Brian Fox is the author of the readline libraries providing command-line editing and command history.

Andrew Beers of SUNY Buffalo wrote the language-switching code, the Modula-2 support, and contributed the Languages chapter of this manual.

Fred Fish wrote most of the support for Unix System Vr4. He also enhanced the command-completion support to cover C++ overloaded symbols.

Hitachi America, Ltd. sponsored the support for H8/300, H8/500, and Super-H processors.

NEC sponsored the support for the v850, Vr4xxx, and Vr5xxx processors.

Mitsubishi sponsored the support for D10V, D30V, and M32R/D processors.

Toshiba sponsored the support for the TX39 Mips processor.

Matsushita sponsored the support for the MN10200 and MN10300 processors.

Fujitsu sponsored the support for SPARClite and FR30 processors

Kung Hsu, Jeff Law, and Rick Sladkey added support for hardware watchpoints.

Michael Snyder added support for tracepoints.

Stu Grossman wrote gdbserver.

Jim Kingdon, Peter Schauer, Ian Taylor, and Stu Grossman made nearly innumerable bug fixes and cleanups throughout GDB.

The following people at the Hewlett-Packard Company contributed support for the PA-RISC 2.0 architecture, HP-UX 10.20, 10.30, and 11.0 (narrow mode), HP's implementation of kernel threads, HP's aC++ compiler, and the terminal user interface: Ben Krepp, Richard Title, John Bishop, Susan Macchia, Kathy Mann, Satish Pai, India Paul, Steve Rehrauer, and Elena Zannoni. Kim Haase provided HP-specific information in this manual.

Cygnus Solutions has sponsored GDB maintenance and much of its development since 1991. Cygnus engineers who have worked on GDB fulltime include Mark Alexander, Jim Blandy, Per Bothner, Edith Epstein, Chris Faylor, Fred Fish, Martin Hunt, Jim Ingham, John Gilmore, Stu Grossman, Kung Hsu, Jim Kingdon, John Metzler, Fernando Nasser, Geoffrey Noer, Dawn Perchik, Rich Pixley, Zdenek Radouch, Keith Seitz, Stan Shebs, David Taylor, and Elena Zannoni. In addition, Dave Brolley, Ian Carmichael, Steve Chamberlain, Nick Clifton, JT Conklin, Stan Cox, DJ Delorie, Ulrich Drepper, Frank Eigler, Doug Evans, Sean Fagan, David Henkel-Wallace, Richard Henderson, Jeff Holcomb, Jeff Law, Jim Lemke, Tom Lord, Bob Manson, Michael Meissner, Jason Merrill, Catherine Moore, Drew Moseley, Ken Raeburn, Gavin Romig-Koch, Rob Savoye, Jamie Smith, Mike Stump, Ian Taylor, Angela Thomas, Michael Tiemann, Tom Tromey, Ron Unrau, Jim Wilson, and David Zuhn have made contributions both

large and small.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

A Sample GDB Session

You can use this manual at your leisure to read all about GDB. However, a handful of commands are enough to get started using the debugger. This chapter illustrates those commands.

In this sample session, we emphasize user input like this: **input**, to make it easier to pick out from the surrounding output.

One of the preliminary versions of GNU `m4` (a generic macro processor) exhibits the following bug: sometimes, when we change its quote strings from the default, the commands used to capture one macro definition within another stop working. In the following short `m4` session, we define a macro `foo` which expands to `0000`; we then use the `m4` built-in `defn` to define `bar` as the same thing. However, when we change the open quote string to `<QUOTE>` and the close quote string to `<UNQUOTE>`, the same procedure fails to define a new synonym `baz`:

```
$ cd gnu/m4
$ ./m4
define(foo,0000)

foo
0000
define(bar,defn(`foo'))

bar
0000
changequote(<QUOTE>,<UNQUOTE>)

define(baz,defn(<QUOTE>foo<UNQUOTE>))
baz
C-d
m4: End of input: 0: fatal error: EOF in string
```

Let us use GDB to try to see what is going on.

```
$ gdb m4
GDB is free software and you are welcome to distribute copies
 of it under certain conditions; type "show copying" to see
 the conditions.
There is absolutely no warranty for GDB; type "show warranty"
 for details.
```


```
GDB 4.18, Copyright 1999 Free Software Foundation, Inc...
(gdb)
```

GDB reads only enough symbol data to know where to find the rest when needed; as a result, the first prompt comes up very quickly. We now tell GDB to use a narrower display width than usual, so that examples fit in this manual.

```
(gdb) set width 70
```

We need to see how the `m4` built-in `changequote` works. Having looked at the source, we know the relevant subroutine is `m4_changequote`, so we set a breakpoint there with the GDB `break` command.

```
(gdb) break m4_changequote
Breakpoint 1 at 0x62f4: file builtin.c, line 879.
```

Using the `run` command, we start `m4` running under GDB control; as long as control does not reach the `m4_changequote` subroutine, the program runs as usual:

```
(gdb) run
Starting program: /work/Editorial/gdb/gnu/m4/m4
define(foo,0000)

foo
0000
```

To trigger the breakpoint, we call `changequote`. GDB suspends execution of `m4`, displaying information about the context where it stops.

```
changequote(<QUOTE>,<UNQUOTE>)

Breakpoint 1, m4_changequote (argc=3, argv=0x33c70)
    at builtin.c:879
879         if (bad_argc(TOKEN_DATA_TEXT(argv[0]),argc,1,3))
```

Now we use the command `n` (`next`) to advance execution to the next line of the current function.

```
(gdb) n
882         set_quotes((argc >= 2) ? TOKEN_DATA_TEXT(argv[1])\
 : nil,
```

`set_quotes` looks like a promising subroutine. We can go into it by using the command `s` (`step`) instead of `next`. `step` goes to the next line to be executed in *any* subroutine, so it steps into `set_quotes`.

```
(gdb) s
```


```
set_quotes (lq=0x34c78 "<QUOTE>", rq=0x34c88 "<UNQUOTE>")
    at input.c:530
530         if (lquote != def_lquote)
```

The display that shows the subroutine where `m4` is now suspended (and its arguments) is called a stack frame display. It shows a summary of the stack. We can use the `backtrace` command (which can also be spelled `bt`), to see where we are in the stack as a whole: the `backtrace` command displays a stack frame for each active subroutine.

```
(gdb) bt
#0  set_quotes (lq=0x34c78 "<QUOTE>", rq=0x34c88 "<UNQUOTE>")
    at input.c:530
#1  0x6344 in m4_changequote (argc=3, argv=0x33c70)
    at builtin.c:882
#2  0x8174 in expand_macro (sym=0x33320) at macro.c:242
#3  0x7a88 in expand_token (obs=0x0, t=209696, td=0xf7fffa30)
    at macro.c:71
#4  0x79dc in expand_input () at macro.c:40
#5  0x2930 in main (argc=0, argv=0xf7fffb20) at m4.c:195
```

We step through a few more lines to see what happens. The first two times, we can use `s'; the next two times we use `n` to avoid falling into the `xstrdup` subroutine.

```
(gdb) s
0x3b5c  532         if (rquote != def_rquote)
(gdb) s
0x3b80  535         lquote = (lq == nil || *lq == '\0') ?  \
def_lquote : xstrdup(lq);
(gdb) n
536         rquote = (rq == nil || *rq == '\0') ? def_rquote\
 : xstrdup(rq);
(gdb) n
538         len_lquote = strlen(rquote);
```

The last line displayed looks a little odd; we can examine the variables `lquote` and `rquote` to see if they are in fact the new left and right quotes we specified. We use the command `p` (`print`) to see their values.

```
(gdb) p lquote
$1 = 0x35d40 "<QUOTE>"
(gdb) p rquote
$2 = 0x35d50 "<UNQUOTE>"
```

`lquote` and `rquote` are indeed the new left and right quotes. To look at some context, we can display ten lines of source surrounding the current line with the `l` (`list`) command.


```
(gdb) l
533             xfree(rquote);
534
535         lquote = (lq == nil || *lq == '\0') ? def_lquote\
 : xstrdup (lq);
536         rquote = (rq == nil || *rq == '\0') ? def_rquote\
 : xstrdup (rq);
537
538         len_lquote = strlen(rquote);
539         len_rquote = strlen(lquote);
540     }
541
542     void
```

Let us step past the two lines that set `len_lquote` and `len_rquote`, and then examine the values of those variables.

```
(gdb) n
539         len_rquote = strlen(lquote);
(gdb) n
540     }
(gdb) p len_lquote
$3 = 9
(gdb) p len_rquote
$4 = 7
```

That certainly looks wrong, assuming `len_lquote` and `len_rquote` are meant to be the lengths of `lquote` and `rquote` respectively. We can set them to better values using the `p` command, since it can print the value of any expression--and that expression can include subroutine calls and assignments.

```
(gdb) p len_lquote=strlen(lquote)
$5 = 7
(gdb) p len_rquote=strlen(rquote)
$6 = 9
```

Is that enough to fix the problem of using the new quotes with the `m4` built-in `defn`? We can allow `m4` to continue executing with the `c` (`continue`) command, and then try the example that caused trouble initially:

```
(gdb) c
Continuing.

define(baz,defn(<QUOTE>foo<UNQUOTE>))

baz
0000
```

Success! The new quotes now work just as well as the default ones. The problem seems to have been the two typos defining the wrong lengths. We allow `m4` exit by giving it an EOF as input:

```
C-d
Program exited normally.
```

The message `Program exited normally.' is from GDB; it indicates `m4` has finished executing. We can end our GDB session with the GDB `quit` command.

```
(gdb) quit
```

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Getting In and Out of GDB

This chapter discusses how to start GDB, and how to get out of it. The essentials are:

- type `gdb' to start GDB.
- type quit or C-d to exit.

Invoking GDB

Invoke GDB by running the program `gdb`. Once started, GDB reads commands from the terminal until you tell it to exit.

You can also run `gdb` with a variety of arguments and options, to specify more of your debugging environment at the outset.

The command-line options described here are designed to cover a variety of situations; in some environments, some of these options may effectively be unavailable.

The most usual way to start GDB is with one argument, specifying an executable program:

```
gdb program
```

You can also start with both an executable program and a core file specified:

```
gdb program core
```

You can, instead, specify a process ID as a second argument, if you want to debug a running process:

```
gdb program 1234
```

would attach GDB to process `1234` (unless you also have a file named `` `1234' ``; GDB does check for a core file first).

Taking advantage of the second command-line argument requires a fairly complete operating system; when you use GDB as a remote debugger attached to a bare board, there may not be any notion of "process", and there is often no way to get a core dump.

You can run `gdb` without printing the front material, which describes GDB's non-warranty, by specifying `-silent`:

```
gdb -silent
```

You can further control how GDB starts up by using command-line options. GDB itself can remind you

of the options available.

Type

```
gdb -help
```

to display all available options and briefly describe their use (`gdb -h' is a shorter equivalent).

All options and command line arguments you give are processed in sequential order. The order makes a difference when the `-x' option is used.

Choosing files

When GDB starts, it reads any arguments other than options as specifying an executable file and core file (or process ID). This is the same as if the arguments were specified by the `-se' and `-c' options respectively. (GDB reads the first argument that does not have an associated option flag as equivalent to the `-se' option followed by that argument; and the second argument that does not have an associated option flag, if any, as equivalent to the `-c' option followed by that argument.)

Many options have both long and short forms; both are shown in the following list. GDB also recognizes the long forms if you truncate them, so long as enough of the option is present to be unambiguous. (If you prefer, you can flag option arguments with `--' rather than `-', though we illustrate the more usual convention.)

`-symbols file`

`-s file`

Read symbol table from file file.

`-exec file`

`-e file`

Use file file as the executable file to execute when appropriate, and for examining pure data in conjunction with a core dump.

`-se file`

Read symbol table from file file and use it as the executable file.

`-core file`

`-c file`

Use file file as a core dump to examine.

`-c number`

Connect to process ID number, as with the `attach` command (unless there is a file in core-dump format named number, in which case `-c' specifies that file as a core dump to read).

`-command file`

`-x file`

Execute GDB commands from file file. See section Command files.

`-directory directory`

`-d directory`

Add directory to the path to search for source files.

`-m`

`-mapped`

Warning: this option depends on operating system facilities that are not supported on all systems. If memory-mapped files are available on your system through the `mmap` system call, you can use this option to have GDB write the symbols from your program into a reusable file in the current directory. If the program you are debugging is called `` `/tmp/fred' ``, the mapped symbol file is `` `./fred.syms' ``. Future GDB debugging sessions notice the presence of this file, and can quickly map in symbol information from it, rather than reading the symbol table from the executable program. The `` `.syms' `` file is specific to the host machine where GDB is run. It holds an exact image of the internal GDB symbol table. It cannot be shared across multiple host platforms.

`-r`

`-readnow`

Read each symbol file's entire symbol table immediately, rather than the default, which is to read it incrementally as it is needed. This makes startup slower, but makes future operations faster.

The `-mapped` and `-readnow` options are typically combined in order to build a `` `.syms' `` file that contains complete symbol information. (See section Commands to specify files, for information on `` `.syms' `` files.) A simple GDB invocation to do nothing but build a `` `.syms' `` file for future use is:

```
gdb -batch -nx -mapped -readnow programname
```

Choosing modes

You can run GDB in various alternative modes--for example, in batch mode or quiet mode.

`-nx`

`-n`

Do not execute commands from any initialization files (normally called `` `.gdbinit' ``, or `` `gdb.ini' `` on PCs). Normally, the commands in these files are executed after all the command options and arguments have been processed. See section Command files.

`-quiet`

`-q`

"Quiet". Do not print the introductory and copyright messages. These messages are also suppressed in batch mode.

`-batch`

Run in batch mode. Exit with status `0` after processing all the command files specified with `-x' (and all commands from initialization files, if not inhibited with `-n'). Exit with nonzero status if an error occurs in executing the GDB commands in the command files. Batch mode may be useful for running GDB as a filter, for example to download and run a program on another computer; in

order to make this more useful, the message

```
Program exited normally.
```

(which is ordinarily issued whenever a program running under GDB control terminates) is not issued when running in batch mode.

`-cd directory`

Run GDB using directory as its working directory, instead of the current directory.

`-fullname`

`-f`

GNU Emacs sets this option when it runs GDB as a subprocess. It tells GDB to output the full file name and line number in a standard, recognizable fashion each time a stack frame is displayed (which includes each time your program stops). This recognizable format looks like two `\032' characters, followed by the file name, line number and character position separated by colons, and a newline. The Emacs-to-GDB interface program uses the two `\032' characters as a signal to display the source code for the frame.

`-b bps`

Set the line speed (baud rate or bits per second) of any serial interface used by GDB for remote debugging.

`-tty device`

Run using device for your program's standard input and output.

Quitting GDB

`quit`

To exit GDB, use the `quit` command (abbreviated `q`), or type an end-of-file character (usually C-d). If you do not supply expression, GDB will terminate normally; otherwise it will terminate using the result of expression as the error code.

An interrupt (often C-c) does not exit from GDB, but rather terminates the action of any GDB command that is in progress and returns to GDB command level. It is safe to type the interrupt character at any time because GDB does not allow it to take effect until a time when it is safe.

If you have been using GDB to control an attached process or device, you can release it with the `detach` command (see section Debugging an already-running process).

Shell commands

If you need to execute occasional shell commands during your debugging session, there is no need to leave or suspend GDB; you can just use the `shell` command.

`shell command string`

Invoke a standard shell to execute command string. If it exists, the environment variable `SHELL`

determines which shell to run. Otherwise GDB uses `/bin/sh`.

The utility `make` is often needed in development environments. You do not have to use the `shell` command for this purpose in GDB:

`make make-args`

Execute the `make` program with the specified arguments. This is equivalent to `shell make make-args'.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

GDB Commands

You can abbreviate a GDB command to the first few letters of the command name, if that abbreviation is unambiguous; and you can repeat certain GDB commands by typing just RET. You can also use the TAB key to get GDB to fill out the rest of a word in a command (or to show you the alternatives available, if there is more than one possibility).

Command syntax

A GDB command is a single line of input. There is no limit on how long it can be. It starts with a command name, which is followed by arguments whose meaning depends on the command name. For example, the command `step` accepts an argument which is the number of times to step, as in `step 5'. You can also use the `step` command with no arguments. Some command names do not allow any arguments.

GDB command names may always be truncated if that abbreviation is unambiguous. Other possible command abbreviations are listed in the documentation for individual commands. In some cases, even ambiguous abbreviations are allowed; for example, `s` is specially defined as equivalent to `step` even though there are other commands whose names start with `s`. You can test abbreviations by using them as arguments to the `help` command.

A blank line as input to GDB (typing just RET) means to repeat the previous command. Certain commands (for example, `run`) will not repeat this way; these are commands whose unintentional repetition might cause trouble and which you are unlikely to want to repeat.

The `list` and `x` commands, when you repeat them with RET, construct new arguments rather than repeating exactly as typed. This permits easy scanning of source or memory.

GDB can also use RET in another way: to partition lengthy output, in a way similar to the common utility `more` (see section Screen size). Since it is easy to press one RET too many in this situation, GDB disables command repetition after any command that generates this sort of display.

Any text from a # to the end of the line is a comment; it does nothing. This is useful mainly in command files (see section Command files).

Command completion

GDB can fill in the rest of a word in a command for you, if there is only one possibility; it can also show you what the valid possibilities are for the next word in a command, at any time. This works for GDB commands, GDB subcommands, and the names of symbols in your program.

Press the TAB key whenever you want GDB to fill out the rest of a word. If there is only one possibi GDB fills in the word, and waits for you to finish the command (or press RET to enter it). For examp if you type

```
(gdb) info bre TAB
```

GDB fills in the rest of the word `breakpoints', since that is the only `info` subcommand beginning with `bre':

```
(gdb) info breakpoints
```

You can either press RET at this point, to run the `info breakpoints` command, or backspace and enter something else, if `breakpoints' does not look like the command you expected. (If you were sure you wanted `info breakpoints` in the first place, you might as well just type RET immediately after `info bre', to exploit command abbreviations rather than command completion).

If there is more than one possibility for the next word when you press TAB, GDB sounds a bell. You can either supply more characters and try again, or just press TAB a second time; GDB displays all the possible completions for that word. For example, you might want to set a breakpoint on a subroutine whose name begins with `make_', but when you type b make_TAB GDB just sounds the bell. Typing TAB again displays all the function names in your program that begin with those characters, for example:

```
(gdb) b make_ TAB
GDB sounds bell; press TAB again, to see:
make_a_section_from_file     make_environ
make_abs_section             make_function_type
make_blockvector             make_pointer_type
make_cleanup                 make_reference_type
make_command                 make_symbol_completion_list
(gdb) b make_
```

After displaying the available possibilities, GDB copies your partial input (`b make_' in the example) so you can finish the command.

If you just want to see the list of alternatives in the first place, you can press M-? rather than pressing TAB twice. M-? means META ?. You can type this either by holding down a key designated as the META shift on your keyboard (if there is one) while typing ?, or as ESC followed by ?.

Sometimes the string you need, while logically a "word", may contain parentheses or other characters that GDB normally excludes from its notion of a word. To permit word completion to work in this situation, you may enclose words in `'` (single quote marks) in GDB commands.

The most likely situation where you might need this is in typing the name of a C++ function. This is because C++ allows function overloading (multiple definitions of the same function, distinguished by argument type). For example, when you want to set a breakpoint you may need to distinguish whether you mean the version of `name` that takes an `int` parameter, `name(int)`, or the version that takes a

`float` parameter, `name(float)`. To use the word-completion facilities in this situation, type a single quote `'` at the beginning of the function name. This alerts GDB that it may need to consider more information than usual when you press TAB or M-? to request word completion:

```
(gdb) b 'bubble( M-?
bubble(double,double)    bubble(int,int)
(gdb) b 'bubble(
```

In some cases, GDB can tell that completing a name requires using quotes. When this happens, GDB inserts the quote for you (while completing as much as it can) if you do not type the quote in the first place:

```
(gdb) b bub TAB
GDB alters your input line to the following, and rings a bell:
(gdb) b 'bubble(
```

In general, GDB can tell that a quote is needed (and inserts it) if you have not yet started typing the argument list when you ask for completion on an overloaded symbol.

For more information about overloaded functions, see section C++ expressions. You can use the command `set overload-resolution off` to disable overload resolution; see section GDB features for C++.

Getting help

You can always ask GDB itself for information on its commands, using the command `help`.

`help`

`h`

> You can use `help` (abbreviated `h`) with no arguments to display a short list of named classes of commands:
>
> ```
> (gdb) help
> List of classes of commands:
>
> running -- Running the program
> stack -- Examining the stack
> data -- Examining data
> breakpoints -- Making program stop at certain points
> files -- Specifying and examining files
> status -- Status inquiries
> support -- Support facilities
> user-defined -- User-defined commands
> aliases -- Aliases of other commands
> obscure -- Obscure features
> ```


```
Type "help" followed by a class name for a list of
commands in that class.
Type "help" followed by command name for full
documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

`help class`

Using one of the general help classes as an argument, you can get a list of the individual commands in that class. For example, here is the help display for the class `status`:

```
(gdb) help status
Status inquiries.

List of commands:

show -- Generic command for showing things set
 with "set"
info -- Generic command for printing status

Type "help" followed by command name for full
documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

`help command`

With a command name as `help` argument, GDB displays a short paragraph on how to use that command.

`complete args`

The `complete args` command lists all the possible completions for the beginning of a command. Use args to specify the beginning of the command you want completed. For example:

```
complete i
```

results in:

```
info
inspect
ignore
```

This is intended for use by GNU Emacs.

In addition to `help`, you can use the GDB commands `info` and `show` to inquire about the state of your program, or the state of GDB itself. Each command supports many topics of inquiry; this manual introduces each of them in the appropriate context. The listings under `info` and under `show` in the Index point to all the sub-commands. See section Index.

`info`

This command (abbreviated `i`) is for describing the state of your program. For example, you can list the arguments given to your program with `info args`, list the registers currently in use with `info registers`, or list the breakpoints you have set with `info breakpoints`. You can get a complete list of the `info` sub-commands with `help info`.

`set`

You can assign the result of an expression to an environment variable with `set`. For example, you can set the GDB prompt to a $-sign with `set prompt $`.

`show`

In contrast to `info`, `show` is for describing the state of GDB itself. You can change most of the things you can `show`, by using the related command `set`; for example, you can control what number system is used for displays with `set radix`, or simply inquire which is currently in use with `show radix`. To display all the settable parameters and their current values, you can use `show` with no arguments; you may also use `info set`. Both commands produce the same display.

Here are three miscellaneous `show` subcommands, all of which are exceptional in lacking corresponding `set` commands:

`show version`

Show what version of GDB is running. You should include this information in GDB bug-reports. If multiple versions of GDB are in use at your site, you may occasionally want to determine which version of GDB you are running; as GDB evolves, new commands are introduced, and old ones may wither away. The version number is also announced when you start GDB.

`show copying`

Display information about permission for copying GDB.

`show warranty`

Display the GNU "NO WARRANTY" statement.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Running Programs Under GDB

When you run a program under GDB, you must first generate debugging information when you compile it. You may start GDB with its arguments, if any, in an environment of your choice. You may redirect your program's input and output, debug an already running process, or kill a child process.

Compiling for debugging

In order to debug a program effectively, you need to generate debugging information when you compile it. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

To request debugging information, specify the `-g' option when you run the compiler.

Many C compilers are unable to handle the `-g' and `-O' options together. Using those compilers, you cannot generate optimized executables containing debugging information.

GCC, the GNU C compiler, supports `-g' with or without `-O', making it possible to debug optimized code. We recommend that you *always* use `-g' whenever you compile a program. You may think your program is correct, but there is no sense in pushing your luck.

When you debug a program compiled with `-g -O', remember that the optimizer is rearranging your code; the debugger shows you what is really there. Do not be too surprised when the execution path does not exactly match your source file! An extreme example: if you define a variable, but never use it, GDB never sees that variable--because the compiler optimizes it out of existence.

Some things do not work as well with `-g -O' as with just `-g', particularly on machines with instruction scheduling. If in doubt, recompile with `-g' alone, and if this fixes the problem, please report it to us as a bug (including a test case!).

Older versions of the GNU C compiler permitted a variant option `-gg' for debugging information. GDB no longer supports this format; if your GNU C compiler has this option, do not use it.

Starting your program

`run`
`r`

Use the `run` command to start your program under GDB. You must first specify the program name (except on VxWorks) with an argument to GDB (see section Getting In and Out of GDB), or by using the `file` or `exec-file` command (see section Commands to specify files).

If you are running your program in an execution environment that supports processes, `run` creates an inferior process and makes that process run your program. (In environments without processes, `run` jumps to the start of your program.)

The execution of a program is affected by certain information it receives from its superior. GDB provides ways to specify this information, which you must do *before* starting your program. (You can change it after starting your program, but such changes only affect your program the next time you start it.) This information may be divided into four categories:

The *arguments.*

> Specify the arguments to give your program as the arguments of the `run` command. If a shell is available on your target, the shell is used to pass the arguments, so that you may use normal conventions (such as wildcard expansion or variable substitution) in describing the arguments. In Unix systems, you can control which shell is used with the `SHELL` environment variable. See section Your program's arguments.

The *environment.*

> Your program normally inherits its environment from GDB, but you can use the GDB commands `set environment` and `unset environment` to change parts of the environment that affect your program. See section Your program's environment.

The *working directory.*

> Your program inherits its working directory from GDB. You can set the GDB working directory with the `cd` command in GDB. See section Your program's working directory.

The *standard input and output.*

> Your program normally uses the same device for standard input and standard output as GDB is using. You can redirect input and output in the `run` command line, or you can use the `tty` command to set a different device for your program. See section Your program's input and output.
>
> *Warning:* While input and output redirection work, you cannot use pipes to pass the output of the program you are debugging to another program; if you attempt this, GDB is likely to wind up debugging the wrong program.

When you issue the `run` command, your program begins to execute immediately. See section Stopping and Continuing, for discussion of how to arrange for your program to stop. Once your program has stopped, you may call functions in your program, using the `print` or `call` commands. See section Examining Data.

If the modification time of your symbol file has changed since the last time GDB read its symbols, GDB discards its symbol table, and reads it again. When it does this, GDB tries to retain your current breakpoints.

Your program's arguments

The arguments to your program can be specified by the arguments of the `run` command. They are passed to a shell, which expands wildcard characters and performs redirection of I/O, and thence to your program. Your `SHELL` environment variable (if it exists) specifies what shell GDB uses. If you do not

define `SHELL`, GDB uses `/bin/sh`.

`run` with no arguments uses the same arguments used by the previous `run`, or those set by the `set args` command.

`set args`

Specify the arguments to be used the next time your program is run. If `set args` has no arguments, `run` executes your program with no arguments. Once you have run your program with arguments, using `set args` before the next `run` is the only way to run it again without arguments.

`show args`

Show the arguments to give your program when it is started.

Your program's environment

The **environment** consists of a set of environment variables and their values. Environment variables conventionally record such things as your user name, your home directory, your terminal type, and your search path for programs to run. Usually you set up environment variables with the shell and they are inherited by all the other programs you run. When debugging, it can be useful to try running your program with a modified environment without having to start GDB over again.

`path directory`

Add directory to the front of the `PATH` environment variable (the search path for executables), for both GDB and your program. You may specify several directory names, separated by `:' or whitespace. If directory is already in the path, it is moved to the front, so it is searched sooner. You can use the string `$cwd' to refer to whatever is the current working directory at the time GDB searches the path. If you use `.' instead, it refers to the directory where you executed the `path` command. GDB replaces `.' in the directory argument (with the current path) before adding directory to the search path.

`show paths`

Display the list of search paths for executables (the `PATH` environment variable).

`show environment [varname]`

Print the value of environment variable varname to be given to your program when it starts. If you do not supply varname, print the names and values of all environment variables to be given to your program. You can abbreviate `environment` as `env`.

`set environment varname [=] value`

Set environment variable varname to value. The value changes for your program only, not for GDB itself. value may be any string; the values of environment variables are just strings, and any interpretation is supplied by your program itself. The value parameter is optional; if it is eliminated, the variable is set to a null value. For example, this command:

```
set env USER = foo
```

tells a Unix program, when subsequently run, that its user is named `foo'. (The spaces around `=' are used for clarity here; they are not actually required.)

`unset environment varname`

Remove variable varname from the environment to be passed to your program. This is different from `set env varname ='; `unset environment` removes the variable from the environment, rather than assigning it an empty value.

Warning: GDB runs your program using the shell indicated by your `SHELL` environment variable if it exists (or `/bin/sh` if not). If your `SHELL` variable names a shell that runs an initialization file--such as `` `.cshrc' `` for C-shell, or `` `.bashrc' `` for BASH--any variables you set in that file affect your program. You may wish to move setting of environment variables to files that are only run when you sign on, such as `` `.login' `` or `` `.profile' ``.

Your program's working directory

Each time you start your program with `run`, it inherits its working directory from the current working directory of GDB. The GDB working directory is initially whatever it inherited from its parent process (typically the shell), but you can specify a new working directory in GDB with the `cd` command.

The GDB working directory also serves as a default for the commands that specify files for GDB to operate on. See section Commands to specify files.

`cd directory`

Set the GDB working directory to directory.

`pwd`

Print the GDB working directory.

Your program's input and output

By default, the program you run under GDB does input and output to the same terminal that GDB uses. GDB switches the terminal to its own terminal modes to interact with you, but it records the terminal modes your program was using and switches back to them when you continue running your program.

`info terminal`

Displays information recorded by GDB about the terminal modes your program is using.

You can redirect your program's input and/or output using shell redirection with the `run` command. For example,

```
run > outfile
```

starts your program, diverting its output to the file `` `outfile' ``.

Another way to specify where your program should do input and output is with the `tty` command. This command accepts a file name as argument, and causes this file to be the default for future `run` commands. It also resets the controlling terminal for the child process, for future `run` commands. For example,

`tty /dev/ttyb`

directs that processes started with subsequent `run` commands default to do input and output on the terminal `/dev/ttyb' and have that as their controlling terminal.

An explicit redirection in `run` overrides the `tty` command's effect on the input/output device, but not its effect on the controlling terminal.

When you use the `tty` command or redirect input in the `run` command, only the input *for your program* is affected. The input for GDB still comes from your terminal.

Debugging an already-running process

`attach process-id`

> This command attaches to a running process--one that was started outside GDB. (`info files` shows your active targets.) The command takes as argument a process ID. The usual way to find out the process-id of a Unix process is with the `ps` utility, or with the `jobs -l' shell command. `attach` does not repeat if you press RET a second time after executing the command.

To use `attach`, your program must be running in an environment which supports processes; for example, `attach` does not work for programs on bare-board targets that lack an operating system. You must also have permission to send the process a signal.

When you use `attach`, the debugger finds the program running in the process first by looking in the current working directory, then (if the program is not found) by using the source file search path (see section Specifying source directories). You can also use the `file` command to load the program. See section Commands to specify files.

The first thing GDB does after arranging to debug the specified process is to stop it. You can examine and modify an attached process with all the GDB commands that are ordinarily available when you start processes with `run`. You can insert breakpoints; you can step and continue; you can modify storage. If you would rather the process continue running, you may use the `continue` command after attaching GDB to the process.

`detach`

> When you have finished debugging the attached process, you can use the `detach` command to release it from GDB control. Detaching the process continues its execution. After the `detach` command, that process and GDB become completely independent once more, and you are ready to `attach` another process or start one with `run`. `detach` does not repeat if you press RET again after executing the command.

If you exit GDB or use the `run` command while you have an attached process, you kill that process. By default, GDB asks for confirmation if you try to do either of these things; you can control whether or not you need to confirm by using the `set confirm` command (see section Optional warnings and messages).

TOC

Killing the child process

`kill`

Kill the child process in which your program is running under GDB.

This command is useful if you wish to debug a core dump instead of a running process. GDB ignores any core dump file while your program is running.

On some operating systems, a program cannot be executed outside GDB while you have breakpoints set on it inside GDB. You can use the `kill` command in this situation to permit running your program outside the debugger.

The `kill` command is also useful if you wish to recompile and relink your program, since on many systems it is impossible to modify an executable file while it is running in a process. In this case, when you next type `run`, GDB notices that the file has changed, and reads the symbol table again (while trying to preserve your current breakpoint settings).

Additional process information

Some operating systems provide a facility called `/proc' that can be used to examine the image of a running process using file-system subroutines. If GDB is configured for an operating system with this facility, the command `info proc` is available to report on several kinds of information about the process running your program. `info proc` works only on SVR4 systems that support `procfs`.

`info proc`

Summarize available information about the process.

`info proc mappings`

Report on the address ranges accessible in the program, with information on whether your program may read, write, or execute each range.

`info proc times`

Starting time, user CPU time, and system CPU time for your program and its children.

`info proc id`

Report on the process IDs related to your program: its own process ID, the ID of its parent, the process group ID, and the session ID.

`info proc status`

General information on the state of the process. If the process is stopped, this report includes the reason for stopping, and any signal received.

`info proc all`

Show all the above information about the process.

Debugging programs with multiple threads

In some operating systems, such as HP-UX and Solaris, a single program may have more than one **thread** of execution. The precise semantics of threads differ from one operating system to another, but in general the threads of a single program are akin to multiple processes--except that they share one address space (that is, they can all examine and modify the same variables). On the other hand, each thread has its own registers and execution stack, and perhaps private memory.

GDB provides these facilities for debugging multi-thread programs:

- automatic notification of new threads
- `thread threadno', a command to switch among threads
- `info threads', a command to inquire about existing threads
- `thread apply [threadno] [all] args', a command to apply a command to a list of threads
- thread-specific breakpoints

  Warning: These facilities are not yet available on every GDB configuration where the operating system supports threads. If your GDB does not support threads, these commands have no effect. For example, a system without thread support shows no output from `info threads', and always rejects the `thread` command, like this:

```
(gdb) info threads
(gdb) thread 1
Thread ID 1 not known.  Use the "info threads" command to
see the IDs of currently known threads.
```

The GDB thread debugging facility allows you to observe all threads while your program runs--but whenever GDB takes control, one thread in particular is always the focus of debugging. This thread is called the **current thread**. Debugging commands show program information from the perspective of the current thread.

Whenever GDB detects a new thread in your program, it displays the target system's identification for the thread with a message in the form `[New systag]'. systag is a thread identifier whose form varies depending on the particular system. For example, on LynxOS, you might see

```
[New process 35 thread 27]
```

when GDB notices a new thread. In contrast, on an SGI system, the systag is simply something like `process 368', with no further qualifier.

For debugging purposes, GDB associates its own thread number--always a single integer--with each thread in your program.

`info threads`

Display a summary of all threads currently in your program. GDB displays for each thread (in this order):

1. the thread number assigned by GDB
2. the target system's thread identifier (systag)
3. the current stack frame summary for that thread

An asterisk `*' to the left of the GDB thread number indicates the current thread. For example,

```
(gdb) info threads
  3 process 35 thread 27  0x34e5 in sigpause ()
  2 process 35 thread 23  0x34e5 in sigpause ()
* 1 process 35 thread 13  main (argc=1, argv=0x7ffffff8)
    at threadtest.c:68
```

`thread threadno`

Make thread number threadno the current thread. The command argument threadno is the internal GDB thread number, as shown in the first field of the `info threads' display. GDB responds by displaying the system identifier of the thread you selected, and its current stack frame summary:

```
(gdb) thread 2
[Switching to process 35 thread 23]
0x34e5 in sigpause ()
```

As with the `[New ...]' message, the form of the text after `Switching to' depends on your system's conventions for identifying threads.

`thread apply [threadno] [all] args`

The `thread apply` command allows you to apply a command to one or more threads. Specify the numbers of the threads that you want affected with the command argument threadno. threadno is the internal GDB thread number, as shown in the first field of the `info threads' display. To apply a command to all threads, use `thread apply all` args.

Whenever GDB stops your program, due to a breakpoint or a signal, it automatically selects the thread where that breakpoint or signal happened. GDB alerts you to the context switch with a message of the form `[Switching to systag]' to identify the thread.

See section Stopping and starting multi-thread programs, for more information about how GDB behaves when you stop and start programs with multiple threads.

See section Setting watchpoints, for information about watchpoints in programs with multiple threads.

Debugging programs with multiple processes

GDB has no special support for debugging programs which create additional processes using the `fork` function. When a program forks, GDB will continue to debug the parent process and the child process will run unimpeded. If you have set a breakpoint in any code which the child then executes, the child will get a `SIGTRAP` signal which (unless it catches the signal) will cause it to terminate.

However, if you want to debug the child process there is a workaround which isn't too painful. Put a call

to `sleep` in the code which the child process executes after the fork. It may be useful to sleep only i certain environment variable is set, or a certain file exists, so that the delay need not occur when you don't want to run GDB on the child. While the child is sleeping, use the `ps` program to get its process ID. Then tell GDB (a new invocation of GDB if you are also debugging the parent process) to attach to the child process (see section Debugging an already-running process). From that point on you can debug the child process just like any other process which you attached to.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Stopping and Continuing

The principal purposes of using a debugger are so that you can stop your program before it terminates; or so that, if your program runs into trouble, you can investigate and find out why.

Inside GDB, your program may stop for any of several reasons, such as a signal, a breakpoint, or reaching a new line after a GDB command such as `step`. You may then examine and change variables, set new breakpoints or remove old ones, and then continue execution. Usually, the messages shown by GDB provide ample explanation of the status of your program--but you can also explicitly request this information at any time.

`info program`

Display information about the status of your program: whether it is running or not, what process it is, and why it stopped.

Breakpoints, watchpoints, and catchpoints

A **breakpoint** makes your program stop whenever a certain point in the program is reached. For each breakpoint, you can add conditions to control in finer detail whether your program stops. You can set breakpoints with the `break` command and its variants (see section Setting breakpoints), to specify the place where your program should stop by line number, function name or exact address in the program.

In HP-UX, SunOS 4.x, SVR4, and Alpha OSF/1 configurations, you can set breakpoints in shared libraries before the executable is run. There is a minor limitation on HP-UX systems: you must wait until the executable is run in order to set breakpoints in shared library routines that are not called directly by the program (for example, routines that are arguments in a `pthread_create` call).

A **watchpoint** is a special breakpoint that stops your program when the value of an expression changes. You must use a different command to set watchpoints (see section Setting watchpoints), but aside from that, you can manage a watchpoint like any other breakpoint: you enable, disable, and delete both breakpoints and watchpoints using the same commands.

You can arrange to have values from your program displayed automatically whenever GDB stops at a breakpoint. See section Automatic display.

A **catchpoint** is another special breakpoint that stops your program when a certain kind of event occurs, such as the throwing of a C++ exception or the loading of a library. As with watchpoints, you use a different command to set a catchpoint (see section Setting catchpoints), but aside from that, you can manage a catchpoint like any other breakpoint. (To stop when your program receives a signal, use the `handle` command; see section Signals.)

GDB assigns a number to each breakpoint, watchpoint, or catchpoint when you create it; these numbers

are successive integers starting with one. In many of the commands for controlling various features of breakpoints you use the breakpoint number to say which breakpoint you want to change. Each breakpoint may be **enabled** or **disabled**; if disabled, it has no effect on your program until you enable it again.

Setting breakpoints

Breakpoints are set with the `break` command (abbreviated `b`). The debugger convenience variable `$bpnum' records the number of the breakpoints you've set most recently; see section Convenience variables, for a discussion of what you can do with convenience variables.

You have several ways to say where the breakpoint should go.

`break function`

Set a breakpoint at entry to function function. When using source languages that permit overloading of symbols, such as C++, function may refer to more than one possible place to break. See section Breakpoint menus, for a discussion of that situation.

`break +offset`

`break -offset`

Set a breakpoint some number of lines forward or back from the position at which execution stopped in the currently selected frame.

`break linenum`

Set a breakpoint at line linenum in the current source file. That file is the last file whose source text was printed. This breakpoint stops your program just before it executes any of the code on that line.

`break filename:linenum`

Set a breakpoint at line linenum in source file filename.

`break filename:function`

Set a breakpoint at entry to function function found in file filename. Specifying a file name as well as a function name is superfluous except when multiple files contain similarly named functions.

`break *address`

Set a breakpoint at address address. You can use this to set breakpoints in parts of your program which do not have debugging information or source files.

`break`

When called without any arguments, `break` sets a breakpoint at the next instruction to be executed in the selected stack frame (see section Examining the Stack). In any selected frame but the innermost, this makes your program stop as soon as control returns to that frame. This is similar to the effect of a `finish` command in the frame inside the selected frame--except that `finish` does not leave an active breakpoint. If you use `break` without an argument in the innermost frame, GDB stops the next time it reaches the current location; this may be useful inside loops. GDB normally ignores breakpoints when it resumes execution, until at least one instruction has been executed. If it did not do this, you would be unable to proceed past a breakpoint without first disabling the breakpoint. This rule applies whether or not the breakpoint already existed when

your program stopped.

`break ... if cond`

Set a breakpoint with condition cond; evaluate the expression cond each time the breakpoint is reached, and stop only if the value is nonzero--that is, if cond evaluates as true. `...' stands for one of the possible arguments described above (or no argument) specifying where to break. See section Break conditions, for more information on breakpoint conditions.

`tbreak args`

Set a breakpoint enabled only for one stop. args are the same as for the `break` command, and the breakpoint is set in the same way, but the breakpoint is automatically deleted after the first time your program stops there. See section Disabling breakpoints.

`hbreak args`

Set a hardware-assisted breakpoint. args are the same as for the `break` command and the breakpoint is set in the same way, but the breakpoint requires hardware support and some target hardware may not have this support. The main purpose of this is EPROM/ROM code debugging, so you can set a breakpoint at an instruction without changing the instruction. This can be used with the new trap-generation provided by SPARClite DSU. DSU will generate traps when a program accesses some data or instruction address that is assigned to the debug registers. However the hardware breakpoint registers can only take two data breakpoints, and GDB will reject this command if more than two are used. Delete or disable unused hardware breakpoints before setting new ones. See section Break conditions.

`thbreak args`

Set a hardware-assisted breakpoint enabled only for one stop. args are the same as for the `hbreak` command and the breakpoint is set in the same way. However, like the `tbreak` command, the breakpoint is automatically deleted after the first time your program stops there. Also, like the `hbreak` command, the breakpoint requires hardware support and some target hardware may not have this support. See section Disabling breakpoints. Also See section Break conditions.

`rbreak regex`

Set breakpoints on all functions matching the regular expression regex. This command sets an unconditional breakpoint on all matches, printing a list of all breakpoints it set. Once these breakpoints are set, they are treated just like the breakpoints set with the `break` command. You can delete them, disable them, or make them conditional the same way as any other breakpoint. When debugging C++ programs, `rbreak` is useful for setting breakpoints on overloaded functions that are not members of any special classes.

`info breakpoints [n]`

`info break [n]`

`info watchpoints [n]`

Print a table of all breakpoints, watchpoints, and catchpoints set and not deleted, with the following columns for each breakpoint:

Breakpoint Numbers

Type

Breakpoint, watchpoint, or catchpoint.

Disposition

Whether the breakpoint is marked to be disabled or deleted when hit.

Enabled or Disabled

Enabled breakpoints are marked with `y'. `n' marks breakpoints that are not enabled.

Address

Where the breakpoint is in your program, as a memory address

What

Where the breakpoint is in the source for your program, as a file and line number.

If a breakpoint is conditional, `info break` shows the condition on the line following the affected breakpoint; breakpoint commands, if any, are listed after that. `info break` with a breakpoint number n as argument lists only that breakpoint. The convenience variable `$_` and the default examining-address for the `x` command are set to the address of the last breakpoint listed (see section Examining memory). `info break` displays a count of the number of times the breakpoint has been hit. This is especially useful in conjunction with the `ignore` command. You can ignore a large number of breakpoint hits, look at the breakpoint info to see how many times the breakpoint was hit, and then run again, ignoring one less than that number. This will get you quickly to the last hit of that breakpoint.

GDB allows you to set any number of breakpoints at the same place in your program. There is nothing silly or meaningless about this. When the breakpoints are conditional, this is even useful (see section Break conditions).

GDB itself sometimes sets breakpoints in your program for special purposes, such as proper handling of `longjmp` (in C programs). These internal breakpoints are assigned negative numbers, starting with `-1`; `info breakpoints' does not display them.

You can see these breakpoints with the GDB maintenance command `maint info breakpoints'.

`maint info breakpoints`

Using the same format as `info breakpoints', display both the breakpoints you've set explicitly, and those GDB is using for internal purposes. Internal breakpoints are shown with negative breakpoint numbers. The type column identifies what kind of breakpoint is shown:

`breakpoint`

Normal, explicitly set breakpoint.

`watchpoint`

Normal, explicitly set watchpoint.

`longjmp`

Internal breakpoint, used to handle correctly stepping through `longjmp` calls.

`longjmp resume`

Internal breakpoint at the target of a `longjmp`.

`until`

Temporary internal breakpoint used by the GDB `until` command.

`finish`

Temporary internal breakpoint used by the GDB `finish` command.

Setting watchpoints

You can use a watchpoint to stop execution whenever the value of an expression changes, without having to predict a particular place where this may happen.

Depending on your system, watchpoints may be implemented in software or hardware. GDB does software watchpointing by single-stepping your program and testing the variable's value each time, which is hundreds of times slower than normal execution. (But this may still be worth it, to catch errors where you have no clue what part of your program is the culprit.)

On some systems, such as HP-UX and Linux, GDB includes support for hardware watchpoints, which do not slow down the running of your program.

`watch expr`

Set a watchpoint for an expression. GDB will break when expr is written into by the program and its value changes.

`rwatch expr`

Set a watchpoint that will break when watch expr is read by the program. If you use both watchpoints, both must be set with the `rwatch` command.

`awatch expr`

Set a watchpoint that will break when args is read and written into by the program. If you use both watchpoints, both must be set with the `awatch` command.

`info watchpoints`

This command prints a list of watchpoints, breakpoints, and catchpoints; it is the same as `info break`.

GDB sets a **hardware watchpoint** if possible. Hardware watchpoints execute very quickly, and the debugger reports a change in value at the exact instruction where the change occurs. If GDB cannot set a hardware watchpoint, it sets a software watchpoint, which executes more slowly and reports the change in value at the next statement, not the instruction, after the change occurs.

When you issue the `watch` command, GDB reports

```
Hardware watchpoint num: expr
```

if it was able to set a hardware watchpoint.

The SPARClite DSU will generate traps when a program accesses some data or instruction address that is assigned to the debug registers. For the data addresses, DSU facilitates the `watch` command. However the hardware breakpoint registers can only take two data watchpoints, and both watchpoints must be the same kind. For example, you can set two watchpoints with `watch` commands, two with `rwatch` commands, **or** two with `awatch` commands, but you cannot set one watchpoint with one command and the other with a different command. GDB will reject the command if you try to mix watchpoints. Delete

or disable unused watchpoint commands before setting new ones.

If you call a function interactively using `print` or `call`, any watchpoints you have set will be inactive until GDB reaches another kind of breakpoint or the call completes.

> *Warning:* In multi-thread programs, watchpoints have only limited usefulness. With the current watchpoint implementation, GDB can only watch the value of an expression *in a single thread*. If you are confident that the expression can only change due to the current thread's activity (and if you are also confident that no other thread can become current), then you can use watchpoints as usual. However, GDB may not notice when a non-current thread's activity changes the expression.

Setting catchpoints

You can use **catchpoints** to cause the debugger to stop for certain kinds of program events, such as C++ exceptions or the loading of a shared library. Use the `catch` command to set a catchpoint.

`catch event`

Stop when event occurs. event can be any of the following:

`throw`

The throwing of a C++ exception.

`catch`

The catching of a C++ exception.

`exec`

A call to `exec`. This is currently only available for HP-UX.

`fork`

A call to `fork`. This is currently only available for HP-UX.

`vfork`

A call to `vfork`. This is currently only available for HP-UX.

`load`

`load libname`

The dynamic loading of any shared library, or the loading of the library libname. This is currently only available for HP-UX.

`unload`

`unload libname`

The unloading of any dynamically loaded shared library, or the unloading of the library libname. This is currently only available for HP-UX.

`tcatch event`

Set a catchpoint that is enabled only for one stop. The catchpoint is automatically deleted after the first time the event is caught.

Use the `info break` command to list the current catchpoints.

There are currently some limitations to C++ exception handling (`catch throw` and `catch cat` in GDB:

- If you call a function interactively, GDB normally returns control to you when the function has finished executing. If the call raises an exception, however, the call may bypass the mechanism that returns control to you and cause your program either to abort or to simply continue running until it hits a breakpoint, catches a signal that GDB is listening for, or exits. This is the case even if you set a catchpoint for the exception; catchpoints on exceptions are disabled within interactive calls.
- You cannot raise an exception interactively.
- You cannot install an exception handler interactively.

Sometimes `catch` is not the best way to debug exception handling: if you need to know exactly where an exception is raised, it is better to stop *before* the exception handler is called, since that way you can see the stack before any unwinding takes place. If you set a breakpoint in an exception handler instead, it may not be easy to find out where the exception was raised.

To stop just before an exception handler is called, you need some knowledge of the implementation. In the case of GNU C++, exceptions are raised by calling a library function named `__raise_exception` which has the following ANSI C interface:

```
/* addr is where the exception identifier is stored.
   ID is the exception identifier.  */
void __raise_exception (void **addr, void *id);
```

To make the debugger catch all exceptions before any stack unwinding takes place, set a breakpoint on `__raise_exception` (see section Breakpoints, watchpoints, and catchpoints).

With a conditional breakpoint (see section Break conditions) that depends on the value of id, you can stop your program when a specific exception is raised. You can use multiple conditional breakpoints to stop your program when any of a number of exceptions are raised.

Deleting breakpoints

It is often necessary to eliminate a breakpoint, watchpoint, or catchpoint once it has done its job and you no longer want your program to stop there. This is called **deleting** the breakpoint. A breakpoint that has been deleted no longer exists; it is forgotten.

With the `clear` command you can delete breakpoints according to where they are in your program. With the `delete` command you can delete individual breakpoints, watchpoints, or catchpoints by specifying their breakpoint numbers.

It is not necessary to delete a breakpoint to proceed past it. GDB automatically ignores breakpoints on the first instruction to be executed when you continue execution without changing the execution address.

`clear`

Delete any breakpoints at the next instruction to be executed in the selected stack frame (see section Selecting a frame). When the innermost frame is selected, this is a good way to delete a

breakpoint where your program just stopped.

`clear function`

`clear filename:function`

Delete any breakpoints set at entry to the function function.

`clear linenum`

`clear filename:linenum`

Delete any breakpoints set at or within the code of the specified line.

`delete [breakpoints] [bnums...]`

Delete the breakpoints, watchpoints, or catchpoints of the numbers specified as arguments. If no argument is specified, delete all breakpoints (GDB asks confirmation, unless you have `set confirm off`). You can abbreviate this command as `d`.

Disabling breakpoints

Rather than deleting a breakpoint, watchpoint, or catchpoint, you might prefer to **disable** it. This makes the breakpoint inoperative as if it had been deleted, but remembers the information on the breakpoint so that you can **enable** it again later.

You disable and enable breakpoints, watchpoints, and catchpoints with the `enable` and `disable` commands, optionally specifying one or more breakpoint numbers as arguments. Use `info break` or `info watch` to print a list of breakpoints, watchpoints, and catchpoints if you do not know which numbers to use.

A breakpoint, watchpoint, or catchpoint can have any of four different states of enablement:

- Enabled. The breakpoint stops your program. A breakpoint set with the `break` command starts out in this state.
- Disabled. The breakpoint has no effect on your program.
- Enabled once. The breakpoint stops your program, but then becomes disabled. A breakpoint set with the `tbreak` command starts out in this state.
- Enabled for deletion. The breakpoint stops your program, but immediately after it does so it is deleted permanently.

You can use the following commands to enable or disable breakpoints, watchpoints, and catchpoints:

`disable [breakpoints] [bnums...]`

Disable the specified breakpoints--or all breakpoints, if none are listed. A disabled breakpoint has no effect but is not forgotten. All options such as ignore-counts, conditions and commands are remembered in case the breakpoint is enabled again later. You may abbreviate `disable` as `dis`.

`enable [breakpoints] [bnums...]`

Enable the specified breakpoints (or all defined breakpoints). They become effective once again in stopping your program.

`enable [breakpoints] once bnums...`

Enable the specified breakpoints temporarily. GDB disables any of these breakpoints immediately

after stopping your program.

`enable [breakpoints] delete bnums...`

Enable the specified breakpoints to work once, then die. GDB deletes any of these breakpoints as soon as your program stops there.

Except for a breakpoint set with `tbreak` (see section Setting breakpoints), breakpoints that you set are initially enabled; subsequently, they become disabled or enabled only when you use one of the commands above. (The command `until` can set and delete a breakpoint of its own, but it does not change the state of your other breakpoints; see section Continuing and stepping.)

Break conditions

The simplest sort of breakpoint breaks every time your program reaches a specified place. You can also specify a **condition** for a breakpoint. A condition is just a Boolean expression in your programming language (see section Expressions). A breakpoint with a condition evaluates the expression each time your program reaches it, and your program stops only if the condition is *true*.

This is the converse of using assertions for program validation; in that situation, you want to stop when the assertion is violated--that is, when the condition is false. In C, if you want to test an assertion expressed by the condition assert, you should set the condition `! assert' on the appropriate breakpoint.

Conditions are also accepted for watchpoints; you may not need them, since a watchpoint is inspecting the value of an expression anyhow--but it might be simpler, say, to just set a watchpoint on a variable name, and specify a condition that tests whether the new value is an interesting one.

Break conditions can have side effects, and may even call functions in your program. This can be useful, for example, to activate functions that log program progress, or to use your own print functions to format special data structures. The effects are completely predictable unless there is another enabled breakpoint at the same address. (In that case, GDB might see the other breakpoint first and stop your program without checking the condition of this one.) Note that breakpoint commands are usually more convenient and flexible for the purpose of performing side effects when a breakpoint is reached (see section Breakpoint command lists).

Break conditions can be specified when a breakpoint is set, by using `if' in the arguments to the `break` command. See section Setting breakpoints. They can also be changed at any time with the `condition` command. The `watch` command does not recognize the `if` keyword; `condition` is the only way to impose a further condition on a watchpoint.

`condition bnum expression`

Specify expression as the break condition for breakpoint, watchpoint, or catchpoint number bnum. After you set a condition, breakpoint bnum stops your program only if the value of expression is true (nonzero, in C). When you use `condition`, GDB checks expression immediately for syntactic correctness, and to determine whether symbols in it have referents in the context of your breakpoint. GDB does not actually evaluate expression at the time the `condition` command is given, however. See section Expressions.

`condition bnum`

Remove the condition from breakpoint number bnum. It becomes an ordinary unconditional breakpoint.

A special case of a breakpoint condition is to stop only when the breakpoint has been reached a certain number of times. This is so useful that there is a special way to do it, using the **ignore count** of the breakpoint. Every breakpoint has an ignore count, which is an integer. Most of the time, the ignore count is zero, and therefore has no effect. But if your program reaches a breakpoint whose ignore count is positive, then instead of stopping, it just decrements the ignore count by one and continues. As a result, if the ignore count value is n, the breakpoint does not stop the next n times your program reaches it.

`ignore bnum count`

Set the ignore count of breakpoint number bnum to count. The next count times the breakpoint is reached, your program's execution does not stop; other than to decrement the ignore count, GDB takes no action. To make the breakpoint stop the next time it is reached, specify a count of zero. When you use `continue` to resume execution of your program from a breakpoint, you can specify an ignore count directly as an argument to `continue`, rather than using `ignore`. See section Continuing and stepping. If a breakpoint has a positive ignore count and a condition, the condition is not checked. Once the ignore count reaches zero, GDB resumes checking the condition. You could achieve the effect of the ignore count with a condition such as `$foo-- <= 0' using a debugger convenience variable that is decremented each time. See section Convenience variables.

Ignore counts apply to breakpoints, watchpoints, and catchpoints.

Breakpoint command lists

You can give any breakpoint (or watchpoint or catchpoint) a series of commands to execute when your program stops due to that breakpoint. For example, you might want to print the values of certain expressions, or enable other breakpoints.

```
commands [bnum]
... command-list ...
end
```

Specify a list of commands for breakpoint number bnum. The commands themselves appear on the following lines. Type a line containing just `end` to terminate the commands. To remove all commands from a breakpoint, type `commands` and follow it immediately with `end`; that is, give no commands. With no bnum argument, `commands` refers to the last breakpoint, watchpoint, or catchpoint set (not to the breakpoint most recently encountered).

Pressing RET as a means of repeating the last GDB command is disabled within a command-list.

You can use breakpoint commands to start your program up again. Simply use the `continue` command, or `step`, or any other command that resumes execution.

Any other commands in the command list, after a command that resumes execution, are ignored. This is because any time you resume execution (even with a simple `next` or `step`), you may encounter another breakpoint--which could have its own command list, leading to ambiguities about which list to execute.

If the first command you specify in a command list is `silent`, the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If none of the remaining commands print anything, you see no sign that the breakpoint was reached. `silent` is meaningful only at the beginning of a breakpoint command list.

The commands `echo`, `output`, and `printf` allow you to print precisely controlled output, and are often useful in silent breakpoints. See section Commands for controlled output.

For example, here is how you could use breakpoint commands to print the value of `x` at entry to `foo` whenever `x` is positive.

```
break foo if x>0
commands
silent
printf "x is %d\n",x
cont
end
```

One application for breakpoint commands is to compensate for one bug so you can test for another. Put a breakpoint just after the erroneous line of code, give it a condition to detect the case in which something erroneous has been done, and give it commands to assign correct values to any variables that need them. End with the `continue` command so that your program does not stop, and start with the `silent` command so that no output is produced. Here is an example:

```
break 403
commands
silent
set x = y + 4
cont
end
```

Breakpoint menus

Some programming languages (notably C++) permit a single function name to be defined several times, for application in different contexts. This is called **overloading**. When a function name is overloaded, `break function' is not enough to tell GDB where you want a breakpoint. If you realize this is a problem, you can use something like `break function(types)' to specify which particular version of the function you want. Otherwise, GDB offers you a menu of numbered choices for different possible breakpoints, and waits for your selection with the prompt `>'. The first two options are always `[0] cancel' and `[1] all'. Typing 1 sets a breakpoint at each definition of function, and typing 0 aborts the `break` command without setting any new breakpoints.

For example, the following session excerpt shows an attempt to set a breakpoint at the overloaded symbol `String::after`. We choose three particular definitions of that function name:

```
(gdb) b String::after
```


```
[0] cancel
[1] all
[2] file:String.cc; line number:867
[3] file:String.cc; line number:860
[4] file:String.cc; line number:875
[5] file:String.cc; line number:853
[6] file:String.cc; line number:846
[7] file:String.cc; line number:735
> 2 4 6
Breakpoint 1 at 0xb26c: file String.cc, line 867.
Breakpoint 2 at 0xb344: file String.cc, line 875.
Breakpoint 3 at 0xafcc: file String.cc, line 846.
Multiple breakpoints were set.
Use the "delete" command to delete unwanted
 breakpoints.
(gdb)
```

Continuing and stepping

Continuing means resuming program execution until your program completes normally. In contrast, **stepping** means executing just one more "step" of your program, where "step" may mean either one line of source code, or one machine instruction (depending on what particular command you use). Either when continuing or when stepping, your program may stop even sooner, due to a breakpoint or a signal. (If due to a signal, you may want to use `handle`, or use `signal 0' to resume execution. See section Signals.)

`continue [ignore-count]`

`c [ignore-count]`

`fg [ignore-count]`

> Resume program execution, at the address where your program last stopped; any breakpoints set at that address are bypassed. The optional argument ignore-count allows you to specify a further number of times to ignore a breakpoint at this location; its effect is like that of `ignore` (see section Break conditions). The argument ignore-count is meaningful only when your program stopped due to a breakpoint. At other times, the argument to `continue` is ignored. The synonyms `c` and `fg` are provided purely for convenience, and have exactly the same behavior as `continue`.

To resume execution at a different place, you can use `return` (see section Returning from a function) to go back to the calling function; or `jump` (see section Continuing at a different address) to go to an arbitrary location in your program.

A typical technique for using stepping is to set a breakpoint (see section Breakpoints, watchpoints, and catchpoints) at the beginning of the function or the section of your program where a problem is believed to lie, run your program until it stops at that breakpoint, and then step through the suspect area, examining the variables that are interesting, until you see the problem happen.

`step`

Continue running your program until control reaches a different source line, then stop it and return control to GDB. This command is abbreviated `s`.

> *Warning:* If you use the `step` command while control is within a function that was compiled without debugging information, execution proceeds until control reaches a function that does have debugging information. Likewise, it will not step into a function which is compiled without debugging information. To step through functions without debugging information, use the `stepi` command, described below.

The `step` command now only stops at the first instruction of a source line. This prevents the multiple stops that used to occur in switch statements, for loops, etc. `step` continues to stop if a function that has debugging information is called within the line. Also, the `step` command now only enters a subroutine if there is line number information for the subroutine. Otherwise it acts like the `next` command. This avoids problems when using `cc -gl` on MIPS machines. Previously, `step` entered subroutines if there was any debugging information about the routine.

`step count`

Continue running as in `step`, but do so count times. If a breakpoint is reached, or a signal not related to stepping occurs before count steps, stepping stops right away.

`next [count]`

Continue to the next source line in the current (innermost) stack frame. This is similar to `step`, but function calls that appear within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the original stack level that was executing when you gave the `next` command. This command is abbreviated `n`. An argument count is a repeat count, as for `step`. The `next` command now only stops at the first instruction of a source line. This prevents the multiple stops that used to occur in switch statements, for loops, etc.

`finish`

Continue running until just after function in the selected stack frame returns. Print the returned value (if any). Contrast this with the `return` command (see section Returning from a function).

`until`

`u`

Continue running until a source line past the current line, in the current stack frame, is reached. This command is used to avoid single stepping through a loop more than once. It is like the `next` command, except that when `until` encounters a jump, it automatically continues execution until the program counter is greater than the address of the jump. This means that when you reach the end of a loop after single stepping though it, `until` makes your program continue execution until it exits the loop. In contrast, a `next` command at the end of a loop simply steps back to the beginning of the loop, which forces you to step through the next iteration. `until` always stops your program if it attempts to exit the current stack frame. `until` may produce somewhat counterintuitive results if the order of machine code does not match the order of the source lines. For example, in the following excerpt from a debugging session, the `f` (`frame`) command shows that execution is stopped at line `206`; yet when we use `until`, we get to line `195`:

```
(gdb) f
```


```
#0  main (argc=4, argv=0xf7fffae8) at m4.c:206
206                 expand_input();
(gdb) until
195             for ( ; argc > 0; NEXTARG) {
```

This happened because, for execution efficiency, the compiler had generated code for the loop closure test at the end, rather than the start, of the loop--even though the test in a C `for`-loop is written before the body of the loop. The `until` command appeared to step back to the beginning of the loop when it advanced to this expression; however, it has not really gone to an earlier statement--not in terms of the actual machine code. `until` with no argument works by means of single instruction stepping, and hence is slower than `until` with an argument.

`until location`

`u location`

Continue running your program until either the specified location is reached, or the current stack frame returns. location is any of the forms of argument acceptable to `break` (see section Setting breakpoints). This form of the command uses breakpoints, and hence is quicker than `until` without an argument.

`stepi`

`si`

Execute one machine instruction, then stop and return to the debugger. It is often useful to do `display/i $pc' when stepping by machine instructions. This makes GDB automatically display the next instruction to be executed, each time your program stops. See section Automatic display. An argument is a repeat count, as in `step`.

`nexti`

`ni`

Execute one machine instruction, but if it is a function call, proceed until the function returns. An argument is a repeat count, as in `next`.

Signals

A signal is an asynchronous event that can happen in a program. The operating system defines the possible kinds of signals, and gives each kind a name and a number. For example, in Unix `SIGINT` is the signal a program gets when you type an interrupt (often C-c); `SIGSEGV` is the signal a program gets from referencing a place in memory far away from all the areas in use; `SIGALRM` occurs when the alarm clock timer goes off (which happens only if your program has requested an alarm).

Some signals, including `SIGALRM`, are a normal part of the functioning of your program. Others, such as `SIGSEGV`, indicate errors; these signals are **fatal** (kill your program immediately) if the program has not specified in advance some other way to handle the signal. `SIGINT` does not indicate an error in your program, but it is normally fatal so it can carry out the purpose of the interrupt: to kill the program.

GDB has the ability to detect any occurrence of a signal in your program. You can tell GDB in advance what to do for each kind of signal.

Normally, GDB is set up to ignore non-erroneous signals like `SIGALRM` (so as not to interfere with t role in the functioning of your program) but to stop your program immediately whenever an error sig happens. You can change these settings with the `handle` command.

`info signals`

Print a table of all the kinds of signals and how GDB has been told to handle each one. You can use this to see the signal numbers of all the defined types of signals. `info handle` is the new alias for `info signals`.

`handle signal keywords...`

Change the way GDB handles signal signal. signal can be the number of a signal or its name (with or without the `SIG' at the beginning). The keywords say what change to make.

The keywords allowed by the `handle` command can be abbreviated. Their full names are:

`nostop`

GDB should not stop your program when this signal happens. It may still print a message telling you that the signal has come in.

`stop`

GDB should stop your program when this signal happens. This implies the `print` keyword as well.

`print`

GDB should print a message when this signal happens.

`noprint`

GDB should not mention the occurrence of the signal at all. This implies the `nostop` keyword as well.

`pass`

GDB should allow your program to see this signal; your program can handle the signal, or else it may terminate if the signal is fatal and not handled.

`nopass`

GDB should not allow your program to see this signal.

When a signal stops your program, the signal is not visible until you continue. Your program sees the signal then, if `pass` is in effect for the signal in question *at that time*. In other words, after GDB reports a signal, you can use the `handle` command with `pass` or `nopass` to control whether your program sees that signal when you continue.

You can also use the `signal` command to prevent your program from seeing a signal, or cause it to see a signal it normally would not see, or to give it any signal at any time. For example, if your program stopped due to some sort of memory reference error, you might store correct values into the erroneous variables and continue, hoping to see more execution; but your program would probably terminate immediately as a result of the fatal signal once it saw the signal. To prevent this, you can continue with `signal 0'. See section Giving your program a signal.

Stopping and starting multi-thread programs

When your program has multiple threads (see section Debugging programs with multiple threads), you can choose whether to set breakpoints on all threads, or on a particular thread.

`break linespec thread threadno`

`break linespec thread threadno if ...`

linespec specifies source lines; there are several ways of writing them, but the effect is always to specify some source line. Use the qualifier `thread threadno' with a breakpoint command to specify that you only want GDB to stop the program when a particular thread reaches this breakpoint. threadno is one of the numeric thread identifiers assigned by GDB, shown in the first column of the `info threads' display. If you do not specify `thread threadno' when you set a breakpoint, the breakpoint applies to *all* threads of your program. You can use the `thread` qualifier on conditional breakpoints as well; in this case, place `thread threadno' before the breakpoint condition, like this:

```
(gdb) break frik.c:13 thread 28 if bartab > lim
```

Whenever your program stops under GDB for any reason, *all* threads of execution stop, not just the current thread. This allows you to examine the overall state of the program, including switching between threads, without worrying that things may change underfoot.

Conversely, whenever you restart the program, *all* threads start executing. *This is true even when single-stepping* with commands like `step` or `next`.

In particular, GDB cannot single-step all threads in lockstep. Since thread scheduling is up to your debugging target's operating system (not controlled by GDB), other threads may execute more than one statement while the current thread completes a single step. Moreover, in general other threads stop in the middle of a statement, rather than at a clean statement boundary, when the program stops.

You might even find your program stopped in another thread after continuing or even single-stepping. This happens whenever some other thread runs into a breakpoint, a signal, or an exception before the first thread completes whatever you requested.

On some OSes, you can lock the OS scheduler and thus allow only a single thread to run.

`set scheduler-locking mode`

Set the scheduler locking mode. If it is `off`, then there is no locking and any thread may run at any time. If `on`, then only the current thread may run when the inferior is resumed. The `step` mode optimizes for single-stepping. It stops other threads from "seizing the prompt" by preempting the current thread while you are stepping. Other threads will only rarely (or never) get a chance to run when you step. They are more likely to run when you "next" over a function call, and they are completely free to run when you use commands like "continue", "until", or "finish". However, unless another thread hits a breakpoint during its timeslice, they will never steal the GDB prompt away from the thread that you are debugging.

`show scheduler-locking`

Display the current scheduler locking mode.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Examining the Stack

When your program has stopped, the first thing you need to know is where it stopped and how it got there.

Each time your program performs a function call, information about the call is generated. That information includes the location of the call in your program, the arguments of the call, and the local variables of the function being called. The information is saved in a block of data called a **stack frame**. The stack frames are allocated in a region of memory called the **call stack**.

When your program stops, the GDB commands for examining the stack allow you to see all of this information.

One of the stack frames is **selected** by GDB and many GDB commands refer implicitly to the selected frame. In particular, whenever you ask GDB for the value of a variable in your program, the value is found in the selected frame. There are special GDB commands to select whichever frame you are interested in. See section Selecting a frame.

When your program stops, GDB automatically selects the currently executing frame and describes it briefly, similar to the `frame` command (see section Information about a frame).

Stack frames

The call stack is divided up into contiguous pieces called **stack frames**, or **frames** for short; each frame is the data associated with one call to one function. The frame contains the arguments given to the function, the function's local variables, and the address at which the function is executing.

When your program is started, the stack has only one frame, that of the function `main`. This is called the **initial** frame or the **outermost** frame. Each time a function is called, a new frame is made. Each time a function returns, the frame for that function invocation is eliminated. If a function is recursive, there can be many frames for the same function. The frame for the function in which execution is actually occurring is called the **innermost** frame. This is the most recently created of all the stack frames that still exist.

Inside your program, stack frames are identified by their addresses. A stack frame consists of many bytes, each of which has its own address; each kind of computer has a convention for choosing one byte whose address serves as the address of the frame. Usually this address is kept in a register called the **frame pointer register** while execution is going on in that frame.

GDB assigns numbers to all existing stack frames, starting with zero for the innermost frame, one for the frame that called it, and so on upward. These numbers do not really exist in your program; they are assigned by GDB to give you a way of designating stack frames in GDB commands.

Some compilers provide a way to compile functions so that they operate without stack frames. (For example, the `gcc` option `-fomit-frame-pointer' generates functions without a frame.) This is occasionally done with heavily used library functions to save the frame setup time. GDB has limited facilities for dealing with these function invocations. If the innermost function invocation has no stack frame, GDB nevertheless regards it as though it had a separate frame, which is numbered zero as usual, allowing correct tracing of the function call chain. However, GDB has no provision for frameless functions elsewhere in the stack.

`frame args`

The `frame` command allows you to move from one stack frame to another, and to print the stack frame you select. args may be either the address of the frame or the stack frame number. Without an argument, `frame` prints the current stack frame.

`select-frame`

The `select-frame` command allows you to move from one stack frame to another without printing the frame. This is the silent version of `frame`.

Backtraces

A backtrace is a summary of how your program got where it is. It shows one line per frame, for many frames, starting with the currently executing frame (frame zero), followed by its caller (frame one), and on up the stack.

`backtrace`

`bt`

Print a backtrace of the entire stack: one line per frame for all frames in the stack. You can stop the backtrace at any time by typing the system interrupt character, normally C-c.

`backtrace n`

`bt n`

Similar, but print only the innermost n frames.

`backtrace -n`

`bt -n`

Similar, but print only the outermost n frames.

The names `where` and `info stack` (abbreviated `info s`) are additional aliases for `backtrace`.

Each line in the backtrace shows the frame number and the function name. The program counter value is also shown--unless you use `set print address off`. The backtrace also shows the source file name and line number, as well as the arguments to the function. The program counter value is omitted if it is at the beginning of the code for that line number.

Here is an example of a backtrace. It was made with the command `bt 3', so it shows the innermost three frames.

```
#0  m4_traceon (obs=0x24eb0, argc=1, argv=0x2b8c8)
```


```
    at builtin.c:993
#1  0x6e38 in expand_macro (sym=0x2b600) at macro.c:242
#2  0x6840 in expand_token (obs=0x0, t=177664, td=0xf7fffb08)
    at macro.c:71
(More stack frames follow...)
```

The display for frame zero does not begin with a program counter value, indicating that your program has stopped at the beginning of the code for line `993` of `builtin.c`.

Selecting a frame

Most commands for examining the stack and other data in your program work on whichever stack frame is selected at the moment. Here are the commands for selecting a stack frame; all of them finish by printing a brief description of the stack frame just selected.

`frame n`

`f n`

Select frame number n. Recall that frame zero is the innermost (currently executing) frame, frame one is the frame that called the innermost one, and so on. The highest-numbered frame is the one for `main`.

`frame addr`

`f addr`

Select the frame at address addr. This is useful mainly if the chaining of stack frames has been damaged by a bug, making it impossible for GDB to assign numbers properly to all frames. In addition, this can be useful when your program has multiple stacks and switches between them. On the SPARC architecture, `frame` needs two addresses to select an arbitrary frame: a frame pointer and a stack pointer. On the MIPS and Alpha architecture, it needs two addresses: a stack pointer and a program counter. On the 29k architecture, it needs three addresses: a register stack pointer, a program counter, and a memory stack pointer.

`up n`

Move n frames up the stack. For positive numbers n, this advances toward the outermost frame, to higher frame numbers, to frames that have existed longer. n defaults to one.

`down n`

Move n frames down the stack. For positive numbers n, this advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. n defaults to one. You may abbreviate `down` as `do`.

All of these commands end by printing two lines of output describing the frame. The first line shows the frame number, the function name, the arguments, and the source file and line number of execution in that frame. The second line shows the text of that source line.

For example:

```
(gdb) up
```


```
#1  0x22f0 in main (argc=1, argv=0xf7fffbf4, env=0xf7fffbfc)
    at env.c:10
10              read_input_file (argv[i]);
```

After such a printout, the `list` command with no arguments prints ten lines centered on the point of execution in the frame. See section Printing source lines.

`up-silently n`

`down-silently n`

These two commands are variants of `up` and `down`, respectively; they differ in that they do their work silently, without causing display of the new frame. They are intended primarily for use in GDB command scripts, where the output might be unnecessary and distracting.

Information about a frame

There are several other commands to print information about the selected stack frame.

`frame`

`f`

When used without any argument, this command does not change which frame is selected, but prints a brief description of the currently selected stack frame. It can be abbreviated `f`. With an argument, this command is used to select a stack frame. See section Selecting a frame.

`info frame`

`info f`

This command prints a verbose description of the selected stack frame, including:

- the address of the frame
- the address of the next frame down (called by this frame)
- the address of the next frame up (caller of this frame)
- the language in which the source code corresponding to this frame is written
- the address of the frame's arguments
- the program counter saved in it (the address of execution in the caller frame)
- which registers were saved in the frame

The verbose description is useful when something has gone wrong that has made the stack format fail to fit the usual conventions.

`info frame addr`

`info f addr`

Print a verbose description of the frame at address addr, without selecting that frame. The selected frame remains unchanged by this command. This requires the same kind of address (more than one for some architectures) that you specify in the `frame` command. See section Selecting a frame.

`info args`

Print the arguments of the selected frame, each on a separate line.

`info locals`

Print the local variables of the selected frame, each on a separate line. These are all variables (declared either static or automatic) accessible at the point of execution of the selected frame.

`info catch`

Print a list of all the exception handlers that are active in the current stack frame at the current point of execution. To see other exception handlers, visit the associated frame (using the `up`, `down`, or `frame` commands); then type `info catch`. See section Setting catchpoints.

MIPS/Alpha machines and the function stack

Alpha- and MIPS-based computers use an unusual stack frame, which sometimes requires GDB to search backward in the object code to find the beginning of a function.

To improve response time (especially for embedded applications, where GDB may be restricted to a slow serial line for this search) you may want to limit the size of this search, using one of these commands:

`set heuristic-fence-post limit`

Restrict GDB to examining at most limit bytes in its search for the beginning of a function. A value of 0 (the default) means there is no limit. However, except for 0, the larger the limit the more bytes `heuristic-fence-post` must search and therefore the longer it takes to run.

`show heuristic-fence-post`

Display the current limit.

These commands are available *only* when GDB is configured for debugging programs on Alpha or MIPS processors.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Examining Source Files

GDB can print parts of your program's source, since the debugging information recorded in the program tells GDB what source files were used to build it. When your program stops, GDB spontaneously prints the line where it stopped. Likewise, when you select a stack frame (see section Selecting a frame), GDB prints the line where execution in that frame has stopped. You can print other portions of source files by explicit command.

If you use GDB through its GNU Emacs interface, you may prefer to use Emacs facilities to view source; see section Using GDB under GNU Emacs.

Printing source lines

To print lines from a source file, use the `list` command (abbreviated `l`). By default, ten lines are printed. There are several ways to specify what part of the file you want to print.

Here are the forms of the `list` command most commonly used:

`list linenum`

Print lines centered around line number linenum in the current source file.

`list function`

Print lines centered around the beginning of function function.

`list`

Print more lines. If the last lines printed were printed with a `list` command, this prints lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame (see section Examining the Stack), this prints lines centered around that line.

`list -`

Print lines just before the lines last printed.

By default, GDB prints ten source lines with any of these forms of the `list` command. You can change this using `set listsize`:

`set listsize count`

Make the `list` command display count source lines (unless the `list` argument explicitly specifies some other number).

`show listsize`

Display the number of lines that `list` prints.

Repeating a `list` command with RET discards the argument, so it is equivalent to typing just `list`.

This is more useful than listing the same lines again. An exception is made for an argument of `-'; that argument is preserved in repetition so that each repetition moves up in the source file.

In general, the `list` command expects you to supply zero, one or two **linespecs**. Linespecs specify source lines; there are several ways of writing them but the effect is always to specify some source line. Here is a complete description of the possible arguments for `list`:

`list linespec`

Print lines centered around the line specified by linespec.

`list first,last`

Print lines from first to last. Both arguments are linespecs.

`list ,last`

Print lines ending with last.

`list first,`

Print lines starting with first.

`list +`

Print lines just after the lines last printed.

`list -`

Print lines just before the lines last printed.

`list`

As described in the preceding table.

Here are the ways of specifying a single source line--all the kinds of linespec.

`number`

Specifies line number of the current source file. When a `list` command has two linespecs, this refers to the same source file as the first linespec.

`+offset`

Specifies the line offset lines after the last line printed. When used as the second linespec in a `list` command that has two, this specifies the line offset lines down from the first linespec.

`-offset`

Specifies the line offset lines before the last line printed.

`filename:number`

Specifies line number in the source file filename.

`function`

Specifies the line that begins the body of the function function. For example: in C, this is the line with the open brace.

`filename:function`

Specifies the line of the open-brace that begins the body of the function function in the file filename. You only need the file name with a function name to avoid ambiguity when there are identically named functions in different source files.

`*address`

Specifies the line containing the program address address. address may be any expression.

Searching source files

There are two commands for searching through the current source file for a regular expression.

`forward-search regexp`

`search regexp`

The command `forward-search regexp' checks each line, starting with the one following the last line listed, for a match for regexp. It lists the line that is found. You can use the synonym `search regexp' or abbreviate the command name as `fo`.

`reverse-search regexp`

The command `reverse-search regexp' checks each line, starting with the one before the last line listed and going backward, for a match for regexp. It lists the line that is found. You can abbreviate this command as `rev`.

Specifying source directories

Executable programs sometimes do not record the directories of the source files from which they were compiled, just the names. Even when they do, the directories could be moved between the compilation and your debugging session. GDB has a list of directories to search for source files; this is called the **source path**. Each time GDB wants a source file, it tries all the directories in the list, in the order they are present in the list, until it finds a file with the desired name. Note that the executable search path is *not* used for this purpose. Neither is the current working directory, unless it happens to be in the source path.

If GDB cannot find a source file in the source path, and the object program records a directory, GDB tries that directory too. If the source path is empty, and there is no record of the compilation directory, GDB looks in the current directory as a last resort.

Whenever you reset or rearrange the source path, GDB clears out any information it has cached about where source files are found and where each line is in the file.

When you start GDB, its source path is empty. To add other directories, use the `directory` command.

`directory dirname ...`

`dir dirname ...`

Add directory dirname to the front of the source path. Several directory names may be given to this command, separated by `:' or whitespace. You may specify a directory that is already in the source path; this moves it forward, so GDB searches it sooner. You can use the string `$cdir' to refer to the compilation directory (if one is recorded), and `$cwd' to refer to the current working directory. `$cwd' is not the same as `.'---the former tracks the current working directory as it changes during your GDB session, while the latter is immediately expanded to the current directory at the time you add an entry to the source path.

`directory`

Reset the source path to empty again. This requires confirmation.

`show directories`

Print the source path: show which directories it contains.

If your source path is cluttered with directories that are no longer of interest, GDB may sometimes cause confusion by finding the wrong versions of source. You can correct the situation as follows:

1. Use `directory` with no argument to reset the source path to empty.
2. Use `directory` with suitable arguments to reinstall the directories you want in the source path. You can add all the directories in one command.

Source and machine code

You can use the command `info line` to map source lines to program addresses (and vice versa), and the command `disassemble` to display a range of addresses as machine instructions. When run under GNU Emacs mode, the `info line` command now causes the arrow to point to the line specified. Also, `info line` prints addresses in symbolic form as well as hex.

`info line linespec`

Print the starting and ending addresses of the compiled code for source line linespec. You can specify source lines in any of the ways understood by the `list` command (see section Printing source lines).

For example, we can use `info line` to discover the location of the object code for the first line of function `m4_changequote`:

```
(gdb) info line m4_changecom
Line 895 of "builtin.c" starts at pc 0x634c and ends at 0x6350.
```

We can also inquire (using `*addr` as the form for linespec) what source line covers a particular address:

```
(gdb) info line *0x63ff
Line 926 of "builtin.c" starts at pc 0x63e4 and ends at 0x6404.
```

After `info line`, the default address for the `x` command is changed to the starting address of the line, so that `x/i' is sufficient to begin examining the machine code (see section Examining memory). Also, this address is saved as the value of the convenience variable `$_` (see section Convenience variables).

`disassemble`

This specialized command dumps a range of memory as machine instructions. The default memory range is the function surrounding the program counter of the selected frame. A single argument to this command is a program counter value; GDB dumps the function surrounding this value. Two arguments specify a range of addresses (first inclusive, second exclusive) to dump.

The following example shows the disassembly of a range of addresses of HP PA-RISC 2.0 code:


```
(gdb) disas 0x32c4 0x32e4
Dump of assembler code from 0x32c4 to 0x32e4:
0x32c4 <main+204>:      addil 0,dp
0x32c8 <main+208>:      ldw 0x22c(sr0,r1),r26
0x32cc <main+212>:      ldil 0x3000,r31
0x32d0 <main+216>:      ble 0x3f8(sr4,r31)
0x32d4 <main+220>:      ldo 0(r31),rp
0x32d8 <main+224>:      addil -0x800,dp
0x32dc <main+228>:      ldo 0x588(r1),r26
0x32e0 <main+232>:      ldil 0x3000,r31
End of assembler dump.
```

Some architectures have more than one commonly-used set of instruction mnemonics or other syntax.

`set assembly-language instruction-set`

> Select the instruction set to use when disassembling the program via the `disassemble` or `x/i` commands. Currently this command is only defined for the Intel x86 family. You can set instruction-set to either `i386` or `i8086`. The default is `i386`.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Examining Data

The usual way to examine data in your program is with the `print` command (abbreviated `p`), or its synonym `inspect`. It evaluates and prints the value of an expression of the language your program is written in (see section Using GDB with Different Languages).

`print exp`

`print /f exp`

> exp is an expression (in the source language). By default the value of exp is printed in a format appropriate to its data type; you can choose a different format by specifying `/f', where f is a letter specifying the format; see section Output formats.

`print`

`print /f`

> If you omit exp, GDB displays the last value again (from the **value history**; see section Value history). This allows you to conveniently inspect the same value in an alternative format.

A more low-level way of examining data is with the `x` command. It examines data in memory at a specified address and prints it in a specified format. See section Examining memory.

If you are interested in information about types, or about how the fields of a struct or class are declared, use the `ptype exp` command rather than `print`. See section Examining the Symbol Table.

Expressions

`print` and many other GDB commands accept an expression and compute its value. Any kind of constant, variable or operator defined by the programming language you are using is valid in an expression in GDB. This includes conditional expressions, function calls, casts and string constants. It unfortunately does not include symbols defined by preprocessor `#define` commands.

GDB now supports array constants in expressions input by the user. The syntax is {element, element...}. For example, you can now use the command `print {1, 2, 3}` to build up an array in memory that is malloc'd in the target program.

Because C is so widespread, most of the expressions shown in examples in this manual are in C. See section Using GDB with Different Languages, for information on how to use expressions in other languages.

In this section, we discuss operators that you can use in GDB expressions regardless of your programming language.

Casts are supported in all languages, not just in C, because it is so useful to cast a number into a pointer

in order to examine a structure at that address in memory.

GDB supports these operators, in addition to those common to programming languages:

`@`

`@' is a binary operator for treating parts of memory as arrays. See section Artificial arrays, for more information.

`::`

`::' allows you to specify a variable in terms of the file or function where it is defined. See section Program variables.

`{type} addr`

Refers to an object of type type stored at address addr in memory. addr may be any expression whose value is an integer or pointer (but parentheses are required around binary operators, just as in a cast). This construct is allowed regardless of what kind of data is normally supposed to reside at addr.

Program variables

The most common kind of expression to use is the name of a variable in your program.

Variables in expressions are understood in the selected stack frame (see section Selecting a frame); they must be either:

- global (or file-static)

or

- visible according to the scope rules of the programming language from the point of execution in that frame

This means that in the function

```
foo (a)
     int a;
{
  bar (a);
  {
    int b = test ();
    bar (b);
  }
}
```

you can examine and use the variable `a` whenever your program is executing within the function `foo`, but you can only use or examine the variable `b` while your program is executing inside the block where `b` is declared.

There is an exception: you can refer to a variable or function whose scope is a single source file even if

the current execution point is not in this file. But it is possible to have more than one such variable or function with the same name (in different source files). If that happens, referring to that name has unpredictable effects. If you wish, you can specify a static variable in a particular function or file, using the colon-colon notation:

```
file::variable
function::variable
```

Here file or function is the name of the context for the static variable. In the case of file names, you can use quotes to make sure GDB parses the file name as a single word--for example, to print a global value of `x` defined in `` `f2.c' ``:

```
(gdb) p 'f2.c'::x
```

This use of `::' is very rarely in conflict with the very similar use of the same notation in C++. GDB also supports use of the C++ scope resolution operator in GDB expressions.

> *Warning:* Occasionally, a local variable may appear to have the wrong value at certain points in a function--just after entry to a new scope, and just before exit.

You may see this problem when you are stepping by machine instructions. This is because, on most machines, it takes more than one instruction to set up a stack frame (including local variable definitions); if you are stepping by machine instructions, variables may appear to have the wrong values until the stack frame is completely built. On exit, it usually also takes more than one machine instruction to destroy a stack frame; after you begin stepping through that group of instructions, local variable definitions may be gone.

This may also happen when the compiler does significant optimizations. To be sure of always seeing accurate values, turn off all optimization when compiling.

Artificial arrays

It is often useful to print out several successive objects of the same type in memory; a section of an array, or an array of dynamically determined size for which only a pointer exists in the program.

You can do this by referring to a contiguous span of memory as an **artificial array**, using the binary operator `@'. The left operand of `@' should be the first element of the desired array and be an individual object. The right operand should be the desired length of the array. The result is an array value whose elements are all of the type of the left argument. The first element is actually the left argument; the second element comes from bytes of memory immediately following those that hold the first element, and so on. Here is an example. If a program says

```
int *array = (int *) malloc (len * sizeof (int));
```

you can print the contents of `array` with


```
p *array@len
```

The left operand of `@' must reside in memory. Array values made with `@' in this way behave just like other arrays in terms of subscripting, and are coerced to pointers when used in expressions. Artificial arrays most often appear in expressions via the value history (see section Value history), after printing one out.

Another way to create an artificial array is to use a cast. This re-interprets a value as if it were an array. The value need not be in memory:

```
(gdb) p/x (short[2])0x12345678
$1 = {0x1234, 0x5678}
```

As a convenience, if you leave the array length out (as in `(type)[])value') gdb calculates the size to fill the value (as `sizeof(value)/sizeof(type)':

```
(gdb) p/x (short[])0x12345678
$2 = {0x1234, 0x5678}
```

Sometimes the artificial array mechanism is not quite enough; in moderately complex data structures, the elements of interest may not actually be adjacent--for example, if you are interested in the values of pointers in an array. One useful work-around in this situation is to use a convenience variable (see section Convenience variables) as a counter in an expression that prints the first interesting value, and then repeat that expression via RET. For instance, suppose you have an array `dtab` of pointers to structures, and you are interested in the values of a field `fv` in each structure. Here is an example of what you might type:

```
set $i = 0
p dtab[$i++]->fv
RET
RET
...
```

Output formats

By default, GDB prints a value according to its data type. Sometimes this is not what you want. For example, you might want to print a number in hex, or a pointer in decimal. Or you might want to view data in memory at a certain address as a character string or as an instruction. To do these things, specify an **output format** when you print a value.

The simplest use of output formats is to say how to print a value already computed. This is done by starting the arguments of the `print` command with a slash and a format letter. The format letters supported are:

`x`

Regard the bits of the value as an integer, and print the integer in hexadecimal.

`d`

Print as integer in signed decimal.

`u`

Print as integer in unsigned decimal.

`o`

Print as integer in octal.

`t`

Print as integer in binary. The letter `t' stands for "two". (1)

`a`

Print as an address, both absolute in hexadecimal and as an offset from the nearest preceding symbol. You can use this format used to discover where (in what function) an unknown address is located:

```
(gdb) p/a 0x54320
$3 = 0x54320 <_initialize_vx+396>
```

`c`

Regard as an integer and print it as a character constant.

`f`

Regard the bits of the value as a floating point number and print using typical floating point syntax.

For example, to print the program counter in hex (see section Registers), type

```
p/x $pc
```

Note that no space is required before the slash; this is because command names in GDB cannot contain a slash.

To reprint the last value in the value history with a different format, you can use the `print` command with just a format and no expression. For example, `p/x' reprints the last value in hex.

Examining memory

You can use the command `x` (for "examine") to examine memory in any of several formats, independently of your program's data types.

`x/nfu addr`

`x addr`

`x`

Use the `x` command to examine memory.

n, f, and u are all optional parameters that specify how much memory to display and how to format it;

addr is an expression giving the address where you want to start displaying memory. If you use defau for nfu, you need not type the slash `/'. Several commands set convenient defaults for addr.

n, the repeat count

The repeat count is a decimal integer; the default is 1. It specifies how much memory (counting by units u) to display.

f, the display format

The display format is one of the formats used by `print`, `s' (null-terminated string), or `i' (machine instruction). The default is `x' (hexadecimal) initially. The default changes each time you use either `x` or `print`.

u, the unit size

The unit size is any of

`b`

Bytes.

`h`

Halfwords (two bytes).

`w`

Words (four bytes). This is the initial default.

`g`

Giant words (eight bytes).

Each time you specify a unit size with `x`, that size becomes the default unit the next time you use `x`. (For the `s' and `i' formats, the unit size is ignored and is normally not written.)

addr, starting display address

addr is the address where you want GDB to begin displaying memory. The expression need not have a pointer value (though it may); it is always interpreted as an integer address of a byte of memory. See section Expressions, for more information on expressions. The default for addr is usually just after the last address examined--but several other commands also set the default address: `info breakpoints` (to the address of the last breakpoint listed), `info line` (to the starting address of a line), and `print` (if you use it to display a value from memory).

For example, `x/3uh 0x54320' is a request to display three halfwords (`h`) of memory, formatted as unsigned decimal integers (`u'), starting at address `0x54320`. `x/4xw $sp' prints the four words (`w') of memory above the stack pointer (here, `$sp'; see section Registers) in hexadecimal (`x').

Since the letters indicating unit sizes are all distinct from the letters specifying output formats, you do not have to remember whether unit size or format comes first; either order works. The output specifications `4xw' and `4wx' mean exactly the same thing. (However, the count n must come first; `wx4' does not work.)

Even though the unit size u is ignored for the formats `s' and `i', you might still want to use a count n; for example, `3i' specifies that you want to see three machine instructions, including any operands. The command `disassemble` gives an alternative way of inspecting machine instructions; see section Source and machine code.

All the defaults for the arguments to `x` are designed to make it easy to continue scanning memory wi minimal specifications each time you use `x`. For example, after you have inspected three machine instructions with `x/3i addr', you can inspect the next seven with just `x/7'. If you use RET to repeat the x command, the repeat count n is used again; the other arguments default as for successive uses of `x`.

The addresses and contents printed by the `x` command are not saved in the value history because there is often too much of them and they would get in the way. Instead, GDB makes these values available for subsequent use in expressions as values of the convenience variables `$_` and `$__`. After an `x` command, the last address examined is available for use in expressions in the convenience variable `$_`. The contents of that address, as examined, are available in the convenience variable `$__`.

If the `x` command has a repeat count, the address and contents saved are from the last memory unit printed; this is not the same as the last address printed if several units were printed on the last line of output.

Automatic display

If you find that you want to print the value of an expression frequently (to see how it changes), you might want to add it to the **automatic display list** so that GDB prints its value each time your program stops. Each expression added to the list is given a number to identify it; to remove an expression from the list, you specify that number. The automatic display looks like this:

```
2: foo = 38
3: bar[5] = (struct hack *) 0x3804
```

This display shows item numbers, expressions and their current values. As with displays you request manually using `x` or `print`, you can specify the output format you prefer; in fact, `display` decides whether to use `print` or `x` depending on how elaborate your format specification is--it uses `x` if you specify a unit size, or one of the two formats (`i' and `s') that are only supported by `x`; otherwise it uses `print`.

`display exp`

Add the expression exp to the list of expressions to display each time your program stops. See section Expressions. `display` does not repeat if you press RET again after using it.

`display/fmt exp`

For fmt specifying only a display format and not a size or count, add the expression exp to the auto-display list but arrange to display it each time in the specified format fmt. See section Output formats.

`display/fmt addr`

For fmt `i' or `s', or including a unit-size or a number of units, add the expression addr as a memory address to be examined each time your program stops. Examining means in effect doing `x/fmt addr'. See section Examining memory.

For example, `display/i $pc' can be helpful, to see the machine instruction about to be executed each time execution stops (`$pc' is a common name for the program counter; see section Registers).

`undisplay dnums...`
`delete display dnums...`

Remove item numbers dnums from the list of expressions to display. `undisplay` does not repeat if you press RET after using it. (Otherwise you would just get the error `No display number ...'.)

`disable display dnums...`

Disable the display of item numbers dnums. A disabled display item is not printed automatically, but is not forgotten. It may be enabled again later.

`enable display dnums...`

Enable display of item numbers dnums. It becomes effective once again in auto display of its expression, until you specify otherwise.

`display`

Display the current values of the expressions on the list, just as is done when your program stops.

`info display`

Print the list of expressions previously set up to display automatically, each one with its item number, but without showing the values. This includes disabled expressions, which are marked as such. It also includes expressions which would not be displayed right now because they refer to automatic variables not currently available.

If a display expression refers to local variables, then it does not make sense outside the lexical context for which it was set up. Such an expression is disabled when execution enters a context where one of its variables is not defined. For example, if you give the command `display last_char` while inside a function with an argument `last_char`, GDB displays this argument while your program continues to stop inside that function. When it stops elsewhere--where there is no variable `last_char`---the display is disabled automatically. The next time your program stops where `last_char` is meaningful, you can enable the display expression once again.

Print settings

GDB provides the following ways to control how arrays, structures, and symbols are printed.

These settings are useful for debugging programs in any language:

`set print address`
`set print address on`

GDB prints memory addresses showing the location of stack traces, structure values, pointer values, breakpoints, and so forth, even when it also displays the contents of those addresses. The default is `on`. For example, this is what a stack frame display looks like with `set print address on`:

```
(gdb) f
#0  set_quotes (lq=0x34c78 "<<", rq=0x34c88 ">>")
    at input.c:530
530         if (lquote != def_lquote)
```

`set print address off`

Do not print addresses when displaying their contents. For example, this is the same stack frame displayed with `set print address off`:

```
(gdb) set print addr off
(gdb) f
#0  set_quotes (lq="<<", rq=">>") at input.c:530
530         if (lquote != def_lquote)
```

You can use `set print address off' to eliminate all machine dependent displays from the GDB interface. For example, with `print address off`, you should get the same text for backtraces on all machines--whether or not they involve pointer arguments.

`show print address`

Show whether or not addresses are to be printed.

When GDB prints a symbolic address, it normally prints the closest earlier symbol plus an offset. If that symbol does not uniquely identify the address (for example, it is a name whose scope is a single source file), you may need to clarify. One way to do this is with `info line`, for example `info line *0x4537'. Alternately, you can set GDB to print the source file and line number when it prints a symbolic address:

`set print symbol-filename on`

Tell GDB to print the source file name and line number of a symbol in the symbolic form of an address.

`set print symbol-filename off`

Do not print source file name and line number of a symbol. This is the default.

`show print symbol-filename`

Show whether or not GDB will print the source file name and line number of a symbol in the symbolic form of an address.

Another situation where it is helpful to show symbol filenames and line numbers is when disassembling code; GDB shows you the line number and source file that corresponds to each instruction.

Also, you may wish to see the symbolic form only if the address being printed is reasonably close to the closest earlier symbol:

`set print max-symbolic-offset max-offset`

Tell GDB to only display the symbolic form of an address if the offset between the closest earlier symbol and the address is less than max-offset. The default is 0, which tells GDB to always print the symbolic form of an address if any symbol precedes it.

`show print max-symbolic-offset`

Ask how large the maximum offset is that GDB prints in a symbolic address.

If you have a pointer and you are not sure where it points, try `set print symbol-filename on'. Then you can determine the name and source file location of the variable where it points, using `p/a pointer'. This interprets the address in symbolic form. For example, here GDB shows that a variable `ptt` points at another variable `t`, defined in `hi2.c':


```
(gdb) set print symbol-filename on
(gdb) p/a ptt
$4 = 0xe008 <t in hi2.c>
```

Warning: For pointers that point to a local variable, `p/a' does not show the symbol name and filename of the referent, even with the appropriate `set print` options turned on.

Other settings control how different kinds of objects are printed:

`set print array`

`set print array on`

Pretty print arrays. This format is more convenient to read, but uses more space. The default is off.

`set print array off`

Return to compressed format for arrays.

`show print array`

Show whether compressed or pretty format is selected for displaying arrays.

`set print elements number-of-elements`

Set a limit on how many elements of an array GDB will print. If GDB is printing a large array, it stops printing after it has printed the number of elements set by the `set print elements` command. This limit also applies to the display of strings. Setting number-of-elements to zero means that the printing is unlimited.

`show print elements`

Display the number of elements of a large array that GDB will print. If the number is 0, then the printing is unlimited.

`set print null-stop`

Cause GDB to stop printing the characters of an array when the first NULL is encountered. This is useful when large arrays actually contain only short strings.

`set print pretty on`

Cause GDB to print structures in an indented format with one member per line, like this:

```
$1 = {
  next = 0x0,
  flags = {
    sweet = 1,
    sour = 1
  },
  meat = 0x54 "Pork"
}
```

`set print pretty off`

Cause GDB to print structures in a compact format, like this:


```
$1 = {next = 0x0, flags = {sweet = 1, sour = 1}, \
meat = 0x54 "Pork"}
```

This is the default format.

`show print pretty`

Show which format GDB is using to print structures.

`set print sevenbit-strings on`

Print using only seven-bit characters; if this option is set, GDB displays any eight-bit characters (in strings or character values) using the notation \nnn. This setting is best if you are working in English (ASCII) and you use the high-order bit of characters as a marker or "meta" bit.

`set print sevenbit-strings off`

Print full eight-bit characters. This allows the use of more international character sets, and is the default.

`show print sevenbit-strings`

Show whether or not GDB is printing only seven-bit characters.

`set print union on`

Tell GDB to print unions which are contained in structures. This is the default setting.

`set print union off`

Tell GDB not to print unions which are contained in structures.

`show print union`

Ask GDB whether or not it will print unions which are contained in structures. For example, given the declarations

```
typedef enum {Tree, Bug} Species;
typedef enum {Big_tree, Acorn, Seedling} Tree_forms;
typedef enum {Caterpillar, Cocoon, Butterfly}
              Bug_forms;

struct thing {
  Species it;
  union {
    Tree_forms tree;
    Bug_forms bug;
  } form;
};

struct thing foo = {Tree, {Acorn}};
```

with `set print union on` in effect `p foo' would print

```
$1 = {it = Tree, form = {tree = Acorn, bug = Cocoon}}
```

and with `set print union off` in effect it would print


```
$1 = {it = Tree, form = {...}}
```

These settings are of interest when debugging C++ programs:

`set print demangle`

`set print demangle on`

Print C++ names in their source form rather than in the encoded ("mangled") form passed to the assembler and linker for type-safe linkage. The default is `on'.

`show print demangle`

Show whether C++ names are printed in mangled or demangled form.

`set print asm-demangle`

`set print asm-demangle on`

Print C++ names in their source form rather than their mangled form, even in assembler code printouts such as instruction disassemblies. The default is off.

`show print asm-demangle`

Show whether C++ names in assembly listings are printed in mangled or demangled form.

`set demangle-style style`

Choose among several encoding schemes used by different compilers to represent C++ names. The choices for style are currently:

`auto`

Allow GDB to choose a decoding style by inspecting your program.

`gnu`

Decode based on the GNU C++ compiler (`g++`) encoding algorithm. This is the default.

`hp`

Decode based on the HP ANSI C++ (`aCC`) encoding algorithm.

`lucid`

Decode based on the Lucid C++ compiler (`lcc`) encoding algorithm.

`arm`

Decode using the algorithm in the C++ Annotated Reference Manual. **Warning:** this setting alone is not sufficient to allow debugging `cfront`-generated executables. GDB would require further enhancement to permit that.

If you omit style, you will see a list of possible formats.

`show demangle-style`

Display the encoding style currently in use for decoding C++ symbols.

`set print object`

`set print object on`

When displaying a pointer to an object, identify the *actual* (derived) type of the object rather than the *declared* type, using the virtual function table.

`set print object off`

Display only the declared type of objects, without reference to the virtual function table. This is the default setting.

`show print object`

Show whether actual, or declared, object types are displayed.

`set print static-members`

`set print static-members on`

Print static members when displaying a C++ object. The default is on.

`set print static-members off`

Do not print static members when displaying a C++ object.

`show print static-members`

Show whether C++ static members are printed, or not.

`set print vtbl`

`set print vtbl on`

Pretty print C++ virtual function tables. The default is off.

`set print vtbl off`

Do not pretty print C++ virtual function tables.

`show print vtbl`

Show whether C++ virtual function tables are pretty printed, or not.

Value history

Values printed by the `print` command are saved in the GDB **value history**. This allows you to refer to them in other expressions. Values are kept until the symbol table is re-read or discarded (for example with the `file` or `symbol-file` commands). When the symbol table changes, the value history is discarded, since the values may contain pointers back to the types defined in the symbol table.

The values printed are given **history numbers** by which you can refer to them. These are successive integers starting with one. `print` shows you the history number assigned to a value by printing `$num = ' before the value; here num is the history number.

To refer to any previous value, use `$' followed by the value's history number. The way `print` labels its output is designed to remind you of this. Just `$` refers to the most recent value in the history, and `$$` refers to the value before that. `$$n` refers to the nth value from the end; `$$2` is the value just prior to `$$`, `$$1` is equivalent to `$$`, and `$$0` is equivalent to `$`.

For example, suppose you have just printed a pointer to a structure and want to see the contents of the structure. It suffices to type

```
p *$
```

If you have a chain of structures where the component `next` points to the next one, you can print the contents of the next one with this:


```
p *$.next
```

You can print successive links in the chain by repeating this command--which you can do by just typing RET.

Note that the history records values, not expressions. If the value of `x` is 4 and you type these commands:

```
print x
set x=5
```

then the value recorded in the value history by the `print` command remains 4 even though the value of `x` has changed.

`show values`

Print the last ten values in the value history, with their item numbers. This is like `p $$9' repeated ten times, except that `show values` does not change the history.

`show values n`

Print ten history values centered on history item number n.

`show values +`

Print ten history values just after the values last printed. If no more values are available, `show values +` produces no display.

Pressing RET to repeat `show values n` has exactly the same effect as `show values +'.

Convenience variables

GDB provides **convenience variables** that you can use within GDB to hold on to a value and refer to it later. These variables exist entirely within GDB; they are not part of your program, and setting a convenience variable has no direct effect on further execution of your program. That is why you can use them freely.

Convenience variables are prefixed with `$'. Any name preceded by `$' can be used for a convenience variable, unless it is one of the predefined machine-specific register names (see section Registers). (Value history references, in contrast, are *numbers* preceded by `$'. See section Value history.)

You can save a value in a convenience variable with an assignment expression, just as you would set a variable in your program. For example:

```
set $foo = *object_ptr
```

would save in `$foo` the value contained in the object pointed to by `object_ptr`.

Using a convenience variable for the first time creates it, but its value is `void` until you assign a new value. You can alter the value with another assignment at any time.

Convenience variables have no fixed types. You can assign a convenience variable any type of value,

including structures and arrays, even if that variable already has a value of a different type. The convenience variable, when used as an expression, has the type of its current value.

`show convenience`

Print a list of convenience variables used so far, and their values. Abbreviated `show con`.

One of the ways to use a convenience variable is as a counter to be incremented or a pointer to be advanced. For example, to print a field from successive elements of an array of structures:

```
set $i = 0
print bar[$i++]->contents
```

Repeat that command by typing RET.

Some convenience variables are created automatically by GDB and given values likely to be useful.

`$_`

The variable `$_` is automatically set by the `x` command to the last address examined (see section Examining memory). Other commands which provide a default address for `x` to examine also set `$_` to that address; these commands include `info line` and `info breakpoint`. The type of `$_` is `void *` except when set by the `x` command, in which case it is a pointer to the type of `$__`.

`$__`

The variable `$__` is automatically set by the `x` command to the value found in the last address examined. Its type is chosen to match the format in which the data was printed.

`$_exitcode`

The variable `$_exitcode` is automatically set to the exit code when the program being debugged terminates.

Registers

You can refer to machine register contents, in expressions, as variables with names starting with `$'. The names of registers are different for each machine; use `info registers` to see the names used on your machine.

`info registers`

Print the names and values of all registers except floating-point registers (in the selected stack frame).

`info all-registers`

Print the names and values of all registers, including floating-point registers.

`info registers regname ...`

Print the **relativized** value of each specified register regname. As discussed in detail below, register values are normally relative to the selected stack frame. regname may be any register name valid on the machine you are using, with or without the initial `$'.

GDB has four "standard" register names that are available (in expressions) on most machines--whenever

they do not conflict with an architecture's canonical mnemonics for registers. The register names `$pc` and `$sp` are used for the program counter register and the stack pointer. `$fp` is used for a register that contains a pointer to the current stack frame, and `$ps` is used for a register that contains the processor status. For example, you could print the program counter in hex with

```
p/x $pc
```

or print the instruction to be executed next with

```
x/i $pc
```

or add four to the stack pointer(2) with

```
set $sp += 4
```

Whenever possible, these four standard register names are available on your machine even though the machine has different canonical mnemonics, so long as there is no conflict. The `info registers` command shows the canonical names. For example, on the SPARC, `info registers` displays the processor status register as `$psr` but you can also refer to it as `$ps`.

GDB always considers the contents of an ordinary register as an integer when the register is examined in this way. Some machines have special registers which can hold nothing but floating point; these registers are considered to have floating point values. There is no way to refer to the contents of an ordinary register as floating point value (although you can *print* it as a floating point value with `print/f $regname').

Some registers have distinct "raw" and "virtual" data formats. This means that the data format in which the register contents are saved by the operating system is not the same one that your program normally sees. For example, the registers of the 68881 floating point coprocessor are always saved in "extended" (raw) format, but all C programs expect to work with "double" (virtual) format. In such cases, GDB normally works with the virtual format only (the format that makes sense for your program), but the `info registers` command prints the data in both formats.

Normally, register values are relative to the selected stack frame (see section Selecting a frame). This means that you get the value that the register would contain if all stack frames farther in were exited and their saved registers restored. In order to see the true contents of hardware registers, you must select the innermost frame (with `frame 0').

However, GDB must deduce where registers are saved, from the machine code generated by your compiler. If some registers are not saved, or if GDB is unable to locate the saved registers, the selected stack frame makes no difference.

`set rstack_high_address address`

> On AMD 29000 family processors, registers are saved in a separate "register stack". There is no way for GDB to determine the extent of this stack. Normally, GDB just assumes that the stack is "large enough". This may result in GDB referencing memory locations that do not exist. If necessary, you can get around this problem by specifying the ending address of the register stack with the `set rstack_high_address` command. The argument should be an address, which

TOC

you probably want to precede with `0x' to specify in hexadecimal.

`show rstack_high_address`

Display the current limit of the register stack, on AMD 29000 family processors.

Floating point hardware

Depending on the configuration, GDB may be able to give you more information about the status of the floating point hardware.

`info float`

Display hardware-dependent information about the floating point unit. The exact contents and layout vary depending on the floating point chip. Currently, `info float' is supported on the ARM and x86 machines.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Using GDB with Different Languages

Although programming languages generally have common aspects, they are rarely expressed in the same manner. For instance, in ANSI C, dereferencing a pointer `p` is accomplished by `*p`, but in Modula-2, it is accomplished by `p^`. Values can also be represented (and displayed) differently. Hex numbers in C appear as `0x1ae', while in Modula-2 they appear as `1AEH'.

Language-specific information is built into GDB for some languages, allowing you to express operations like the above in your program's native language, and allowing GDB to output values in a manner consistent with the syntax of your program's native language. The language you use to build expressions is called the **working language**.

Switching between source languages

There are two ways to control the working language--either have GDB set it automatically, or select it manually yourself. You can use the `set language` command for either purpose. On startup, GDB defaults to setting the language automatically. The working language is used to determine how expressions you type are interpreted, how values are printed, etc.

In addition to the working language, every source file that GDB knows about has its own working language. For some object file formats, the compiler might indicate which language a particular source file is in. However, most of the time GDB infers the language from the name of the file. The language of a source file controls whether C++ names are demangled--this way `backtrace` can show each frame appropriately for its own language. There is no way to set the language of a source file from within GDB.

This is most commonly a problem when you use a program, such as `cfront` or `f2c`, that generates C but is written in another language. In that case, make the program use `#line` directives in its C output; that way GDB will know the correct language of the source code of the original program, and will display that source code, not the generated C code.

List of filename extensions and languages

If a source file name ends in one of the following extensions, then GDB infers that its language is the one indicated.

`.c'`
: C source file

`.C'`

`.cc'`

`.cp'`

`.cpp'
`.cxx'
`.c++'
: C++ source file

`.f'
`.F'
: Fortran source file

`.ch'
`.c186'
`.c286'
: CHILL source file.

`.mod'
: Modula-2 source file

`.s'
`.S'
: Assembler source file. This actually behaves almost like C, but GDB does not skip over function prologues when stepping.

In addition, you may set the language associated with a filename extension. See section Displaying the language.

Setting the working language

If you allow GDB to set the language automatically, expressions are interpreted the same way in your debugging session and your program.

If you wish, you may set the language manually. To do this, issue the command `set language lang', where lang is the name of a language, such as `c` or `modula-2`. For a list of the supported languages, type `set language'.

Setting the language manually prevents GDB from updating the working language automatically. This can lead to confusion if you try to debug a program when the working language is not the same as the source language, when an expression is acceptable to both languages--but means different things. For instance, if the current source file were written in C, and GDB was parsing Modula-2, a command such as:

```
print a = b + c
```

might not have the effect you intended. In C, this means to add `b` and `c` and place the result in `a`. The result printed would be the value of `a`. In Modula-2, this means to compare `a` to the result of `b+c`, yielding a `BOOLEAN` value.

Having GDB infer the source language

To have GDB set the working language automatically, use `set language local' or `set language auto'. GDB then infers the working language. That is, when your program stops in a frame (usually by encountering a breakpoint), GDB sets the working language to the language recorded for the function in that frame. If the language for a frame is unknown (that is, if the function or block corresponding to the frame was defined in a source file that does not have a recognized extension), the current working language is not changed, and GDB issues a warning.

This may not seem necessary for most programs, which are written entirely in one source language. However, program modules and libraries written in one source language can be used by a main program written in a different source language. Using `set language auto' in this case frees you from having to set the working language manually.

Displaying the language

The following commands help you find out which language is the working language, and also what language source files were written in.

`show language`

Display the current working language. This is the language you can use with commands such as `print` to build and compute expressions that may involve variables in your program.

`info frame`

Display the source language for this frame. This language becomes the working language if you use an identifier from this frame. See section Information about a frame, to identify the other information listed here.

`info source`

Display the source language of this source file. See section Examining the Symbol Table, to identify the other information listed here.

In unusual circumstances, you may have source files with extensions not in the standard list. You can then set the extension associated with a language explicitly:

`set extension-language .ext language`

Set source files with extension .ext to be assumed to be in the source language language.

`info extensions`

List all the filename extensions and the associated languages.

Type and range checking

Warning: In this release, the GDB commands for type and range checking are included, but they do not yet have any effect. This section documents the intended facilities.

Some languages are designed to guard you against making seemingly common errors through a series of

compile- and run-time checks. These include checking the type of arguments to functions and operat and making sure mathematical overflows are caught at run time. Checks such as these help to ensure program's correctness once it has been compiled by eliminating type mismatches, and providing active checks for range errors when your program is running.

GDB can check for conditions like the above if you wish. Although GDB does not check the statements in your program, it can check expressions entered directly into GDB for evaluation via the `print` command, for example. As with the working language, GDB can also decide whether or not to check automatically based on your program's source language. See section Supported languages, for the default settings of supported languages.

An overview of type checking

Some languages, such as Modula-2, are strongly typed, meaning that the arguments to operators and functions have to be of the correct type, otherwise an error occurs. These checks prevent type mismatch errors from ever causing any run-time problems. For example,

```
1 + 2 => 3
but
error--> 1 + 2.3
```

The second example fails because the `CARDINAL` 1 is not type-compatible with the `REAL` 2.3.

For the expressions you use in GDB commands, you can tell the GDB type checker to skip checking; to treat any mismatches as errors and abandon the expression; or to only issue warnings when type mismatches occur, but evaluate the expression anyway. When you choose the last of these, GDB evaluates expressions like the second example above, but also issues a warning.

Even if you turn type checking off, there may be other reasons related to type that prevent GDB from evaluating an expression. For instance, GDB does not know how to add an `int` and a `struct foo`. These particular type errors have nothing to do with the language in use, and usually arise from expressions, such as the one described above, which make little sense to evaluate anyway.

Each language defines to what degree it is strict about type. For instance, both Modula-2 and C require the arguments to arithmetical operators to be numbers. In C, enumerated types and pointers can be represented as numbers, so that they are valid arguments to mathematical operators. See section Supported languages, for further details on specific languages.

GDB provides some additional commands for controlling the type checker:

`set check type auto`

Set type checking on or off based on the current working language. See section Supported languages, for the default settings for each language.

`set check type on`

`set check type off`

Set type checking on or off, overriding the default setting for the current working language. Issue a warning if the setting does not match the language default. If any type mismatches occur in

evaluating an expression while typechecking is on, GDB prints a message and aborts evaluatic the expression.

`set check type warn`

Cause the type checker to issue warnings, but to always attempt to evaluate the expression. Evaluating the expression may still be impossible for other reasons. For example, GDB cannot add numbers and structures.

`show type`

Show the current setting of the type checker, and whether or not GDB is setting it automatically.

An overview of range checking

In some languages (such as Modula-2), it is an error to exceed the bounds of a type; this is enforced with run-time checks. Such range checking is meant to ensure program correctness by making sure computations do not overflow, or indices on an array element access do not exceed the bounds of the array.

For expressions you use in GDB commands, you can tell GDB to treat range errors in one of three ways: ignore them, always treat them as errors and abandon the expression, or issue warnings but evaluate the expression anyway.

A range error can result from numerical overflow, from exceeding an array index bound, or when you type a constant that is not a member of any type. Some languages, however, do not treat overflows as an error. In many implementations of C, mathematical overflow causes the result to "wrap around" to lower values--for example, if m is the largest integer value, and s is the smallest, then

```
m + 1 => s
```

This, too, is specific to individual languages, and in some cases specific to individual compilers or machines. See section Supported languages, for further details on specific languages.

GDB provides some additional commands for controlling the range checker:

`set check range auto`

Set range checking on or off based on the current working language. See section Supported languages, for the default settings for each language.

`set check range on`

`set check range off`

Set range checking on or off, overriding the default setting for the current working language. A warning is issued if the setting does not match the language default. If a range error occurs, then a message is printed and evaluation of the expression is aborted.

`set check range warn`

Output messages when the GDB range checker detects a range error, but attempt to evaluate the expression anyway. Evaluating the expression may still be impossible for other reasons, such as accessing memory that the process does not own (a typical example from many Unix systems).

`show range`

Show the current setting of the range checker, and whether or not it is being set automatically by GDB.

Supported languages

GDB supports C, C++, Fortran, Chill, assembly, and Modula-2. Some GDB features may be used in expressions regardless of the language you use: the GDB `@` and `::` operators, and the `{type}addr' construct (see section Expressions) can be used with the constructs of any supported language.

The following sections detail to what degree each source language is supported by GDB. These sections are not meant to be language tutorials or references, but serve only as a reference guide to what the GDB expression parser accepts, and what input and output formats should look like for different languages. There are many good books written on each of these languages; please look to these for a language reference or tutorial.

C and C++

Since C and C++ are so closely related, many features of GDB apply to both languages. Whenever this is the case, we discuss those languages together.

The C++ debugging facilities are jointly implemented by the C++ compiler and GDB. Therefore, to debug your C++ code effectively, you must compile your C++ programs with a supported C++ compiler, such as GNU `g++`, or the HP ANSI C++ compiler (`aCC`).

For best results when using GNU C++, use the stabs debugging format. You can select that format explicitly with the `g++` command-line options `-gstabs' or `-gstabs+'. See section `Options for Debugging Your Program or GNU CC' in Using GNU CC, for more information.

C and C++ operators

Operators must be defined on values of specific types. For instance, `+` is defined on numbers, but not on structures. Operators are often defined on groups of types.

For the purposes of C and C++, the following definitions hold:

- *Integral types* include `int` with any of its storage-class specifiers; `char`; and `enum`.
- *Floating-point types* include `float` and `double`.
- *Pointer types* include all types defined as `(type *)`.
- *Scalar types* include all of the above.

The following operators are supported. They are listed here in order of increasing precedence:

`,`

The comma or sequencing operator. Expressions in a comma-separated list are evaluated from left to right, with the result of the entire expression being the last expression evaluated.

`=`

Assignment. The value of an assignment expression is the value assigned. Defined on scalar types.

`op=`

Used in an expression of the form `a op= b`, and translated to `a = a op b`. `op=` and `=` have the same precendence. op is any one of the operators `|`, `^`, `&`, `<<`, `>>`, `+`, `-`, `*`, `/`, `%`.

`?:`

The ternary operator. `a ? b : c` can be thought of as: if a then b else c. a should be of an integral type.

`||`

Logical OR. Defined on integral types.

`&&`

Logical AND. Defined on integral types.

`|`

Bitwise OR. Defined on integral types.

`^`

Bitwise exclusive-OR. Defined on integral types.

`&`

Bitwise AND. Defined on integral types.

`==, !=`

Equality and inequality. Defined on scalar types. The value of these expressions is 0 for false and non-zero for true.

`<, >, <=, >=`

Less than, greater than, less than or equal, greater than or equal. Defined on scalar types. The value of these expressions is 0 for false and non-zero for true.

`<<, >>`

left shift, and right shift. Defined on integral types.

`@`

The GDB "artificial array" operator (see section Expressions).

`+, -`

Addition and subtraction. Defined on integral types, floating-point types and pointer types.

`*, /, %`

Multiplication, division, and modulus. Multiplication and division are defined on integral and floating-point types. Modulus is defined on integral types.

`++, --`

Increment and decrement. When appearing before a variable, the operation is performed before the variable is used in an expression; when appearing after it, the variable's value is used before the operation takes place.

`*`

Pointer dereferencing. Defined on pointer types. Same precedence as `++`.

`&`

Address operator. Defined on variables. Same precedence as `++`. For debugging C++, GDB implements a use of `&' beyond what is allowed in the C++ language itself: you can use `&(&ref)' (or, if you prefer, simply `&&ref') to examine the address where a C++ reference variable (declared with `&ref') is stored.

`-`

Negative. Defined on integral and floating-point types. Same precedence as `++`.

`!`

Logical negation. Defined on integral types. Same precedence as `++`.

`~`

Bitwise complement operator. Defined on integral types. Same precedence as `++`.

`., ->`

Structure member, and pointer-to-structure member. For convenience, GDB regards the two as equivalent, choosing whether to dereference a pointer based on the stored type information. Defined on `struct` and `union` data.

`[]`

Array indexing. `a[i]` is defined as `*(a+i)`. Same precedence as `->`.

`()`

Function parameter list. Same precedence as `->`.

`::`

C++ scope resolution operator. Defined on `struct`, `union`, and `class` types.

`::`

Doubled colons also represent the GDB scope operator (see section Expressions). Same precedence as `::`, above.

C and C++ constants

GDB allows you to express the constants of C and C++ in the following ways:

- Integer constants are a sequence of digits. Octal constants are specified by a leading `0' (i.e. zero), and hexadecimal constants by a leading `0x' or `0X'. Constants may also end with a letter `l', specifying that the constant should be treated as a `long` value.
- Floating point constants are a sequence of digits, followed by a decimal point, followed by a sequence of digits, and optionally followed by an exponent. An exponent is of the form: `e[[+]|-]nnn', where nnn is another sequence of digits. The `+' is optional for positive exponents.
- Enumerated constants consist of enumerated identifiers, or their integral equivalents.
- Character constants are a single character surrounded by single quotes (`'`), or a number--the ordinal value of the corresponding character (usually its ASCII value). Within quotes, the single character may be represented by a letter or by **escape sequences**, which are of the form `\nnn', where nnn is the octal representation of the character's ordinal value; or of the form `\x', where `x'

is a predefined special character--for example, `\n' for newline.

- String constants are a sequence of character constants surrounded by double quotes (").
- Pointer constants are an integral value. You can also write pointers to constants using the C operator `&'.
- Array constants are comma-separated lists surrounded by braces `{' and `}'; for example, `{1,2,3}' is a three-element array of integers, `{{1,2}, {3,4}, {5,6}}' is a three-by-two array, and `{&"hi", &"there", &"fred"}' is a three-element array of pointers.

C++ expressions

GDB expression handling can interpret most C++ expressions.

Warning: GDB can only debug C++ code if you use the proper compiler. Typically, C++ debugging depends on the use of additional debugging information in the symbol table, and thus requires special support. In particular, if your compiler generates a.out, MIPS ECOFF, RS/6000 XCOFF, or ELF with stabs extensions to the symbol table, these facilities are all available. (With GNU CC, you can use the `-gstabs' option to request stabs debugging extensions explicitly.) Where the object code format is standard COFF or DWARF in ELF, on the other hand, most of the C++ support in GDB does *not* work.

1. Member function calls are allowed; you can use expressions like

   ```
   count = aml->GetOriginal(x, y)
   ```

2. While a member function is active (in the selected stack frame), your expressions have the same namespace available as the member function; that is, GDB allows implicit references to the class instance pointer `this` following the same rules as C++.
3. You can call overloaded functions; GDB resolves the function call to the right definition, with one restriction--you must use arguments of the type required by the function that you want to call. GDB does not perform conversions requiring constructors or user-defined type operators.
4. GDB understands variables declared as C++ references; you can use them in expressions just as you do in C++ source--they are automatically dereferenced. In the parameter list shown when GDB displays a frame, the values of reference variables are not displayed (unlike other variables); this avoids clutter, since references are often used for large structures. The *address* of a reference variable is always shown, unless you have specified `set print address off'.
5. GDB supports the C++ name resolution operator `::`---your expressions can use it just as expressions in your program do. Since one scope may be defined in another, you can use `::` repeatedly if necessary, for example in an expression like `scope1::scope2::name'. GDB also allows resolving name scope by reference to source files, in both C and C++ debugging (see section Program variables).

C and C++ defaults

If you allow GDB to set type and range checking automatically, they both default to `off` whenever the working language changes to C or C++. This happens regardless of whether you or GDB selects the working language.

If you allow GDB to set the language automatically, it recognizes source files whose names end with `.c', `.C', or `.cc', etc, and when GDB enters code compiled from one of these files, it sets the working language to C or C++. See section Having GDB infer the source language, for further details.

C and C++ type and range checks

By default, when GDB parses C or C++ expressions, type checking is not used. However, if you turn type checking on, GDB considers two variables type equivalent if:

- The two variables are structured and have the same structure, union, or enumerated tag.
- The two variables have the same type name, or types that have been declared equivalent through `typedef`.

Range checking, if turned on, is done on mathematical operations. Array indices are not checked, since they are often used to index a pointer that is not itself an array.

GDB and C

The `set print union` and `show print union` commands apply to the `union` type. When set to `on', any `union` that is inside a `struct` or `class` is also printed. Otherwise, it appears as `{...}'.

The @ operator aids in the debugging of dynamic arrays, formed with pointers and a memory allocation function. See section Expressions.

GDB features for C++

Some GDB commands are particularly useful with C++, and some are designed specifically for use with C++. Here is a summary:

`breakpoint menus`

When you want a breakpoint in a function whose name is overloaded, GDB breakpoint menus help you specify which function definition you want. See section Breakpoint menus.

`rbreak regex`

Setting breakpoints using regular expressions is helpful for setting breakpoints on overloaded functions that are not members of any special classes. See section Setting breakpoints.

`catch throw`

`catch catch`

Debug C++ exception handling using these commands. See section Setting catchpoints.

`ptype typename`

Print inheritance relationships as well as other information for type typename. See section Examining the Symbol Table.

`set print demangle`

`show print demangle`

`set print asm-demangle`

`show print asm-demangle`

Control whether C++ symbols display in their source form, both when displaying code as C++ source and when displaying disassemblies. See section Print settings.

`set print object`

`show print object`

Choose whether to print derived (actual) or declared types of objects. See section Print settings.

`set print vtbl`

`show print vtbl`

Control the format for printing virtual function tables. See section Print settings.

`Overloaded symbol names`

You can specify a particular definition of an overloaded symbol, using the same notation that is used to declare such symbols in C++: type `symbol(types)` rather than just symbol. You can also use the GDB command-line word completion facilities to list the available choices, or to finish the type list for you. See section Command completion, for details on how to do this.

Modula-2

The extensions made to GDB to support Modula-2 only support output from the GNU Modula-2 compiler (which is currently being developed). Other Modula-2 compilers are not currently supported, and attempting to debug executables produced by them is most likely to give an error as GDB reads in the executable's symbol table.

Operators

Operators must be defined on values of specific types. For instance, `+` is defined on numbers, but not on structures. Operators are often defined on groups of types. For the purposes of Modula-2, the following definitions hold:

- *Integral types* consist of `INTEGER`, `CARDINAL`, and their subranges.
- *Character types* consist of `CHAR` and its subranges.
- *Floating-point types* consist of `REAL`.
- *Pointer types* consist of anything declared as `POINTER TO type`.
- *Scalar types* consist of all of the above.
- *Set types* consist of `SET` and `BITSET` types.
- *Boolean types* consist of `BOOLEAN`.

The following operators are supported, and appear in order of increasing precedence:

`,`

Function argument or array index separator.

`:=`

Assignment. The value of var `:=` value is value.

`<, >`

Less than, greater than on integral, floating-point, or enumerated types.

`<=, >=`

Less than, greater than, less than or equal to, greater than or equal to on integral, floating-point and enumerated types, or set inclusion on set types. Same precedence as `<`.

`=, <>, #`

Equality and two ways of expressing inequality, valid on scalar types. Same precedence as `<`. In GDB scripts, only `<>` is available for inequality, since # conflicts with the script comment character.

`IN`

Set membership. Defined on set types and the types of their members. Same precedence as `<`.

`OR`

Boolean disjunction. Defined on boolean types.

`AND, &`

Boolean conjuction. Defined on boolean types.

`@`

The GDB "artificial array" operator (see section Expressions).

`+, -`

Addition and subtraction on integral and floating-point types, or union and difference on set types.

`*`

Multiplication on integral and floating-point types, or set intersection on set types.

`/`

Division on floating-point types, or symmetric set difference on set types. Same precedence as `*`.

`DIV, MOD`

Integer division and remainder. Defined on integral types. Same precedence as `*`.

`-`

Negative. Defined on `INTEGER` and `REAL` data.

`^`

Pointer dereferencing. Defined on pointer types.

`NOT`

Boolean negation. Defined on boolean types. Same precedence as `^`.

`.`

`RECORD` field selector. Defined on `RECORD` data. Same precedence as `^`.

`[]`

Array indexing. Defined on `ARRAY` data. Same precedence as `^`.

`()`

Procedure argument list. Defined on `PROCEDURE` objects. Same precedence as `^`.

`::, .`

GDB and Modula-2 scope operators.

Warning: Sets and their operations are not yet supported, so GDB treats the use of the operator `IN`, or the use of operators `+`, `-`, `*`, `/`, `=`, , `<>`, #, `<=`, and `>=` on sets as an error.

Built-in functions and procedures

Modula-2 also makes available several built-in procedures and functions. In describing these, the following metavariables are used:

a

represents an `ARRAY` variable.

c

represents a `CHAR` constant or variable.

i

represents a variable or constant of integral type.

m

represents an identifier that belongs to a set. Generally used in the same function with the metavariable s. The type of s should be `SET OF mtype` (where mtype is the type of m).

n

represents a variable or constant of integral or floating-point type.

r

represents a variable or constant of floating-point type.

t

represents a type.

v

represents a variable.

x

represents a variable or constant of one of many types. See the explanation of the function for details.

All Modula-2 built-in procedures also return a result, described below.

`ABS(n)`

Returns the absolute value of n.

`CAP(c)`

If c is a lower case letter, it returns its upper case equivalent, otherwise it returns its argument

`CHR(i)`

Returns the character whose ordinal value is i.

`DEC(v)`

Decrements the value in the variable v. Returns the new value.

`DEC(v,i)`
: Decrements the value in the variable v by i. Returns the new value.

`EXCL(m,s)`
: Removes the element m from the set s. Returns the new set.

`FLOAT(i)`
: Returns the floating point equivalent of the integer i.

`HIGH(a)`
: Returns the index of the last member of a.

`INC(v)`
: Increments the value in the variable v. Returns the new value.

`INC(v,i)`
: Increments the value in the variable v by i. Returns the new value.

`INCL(m,s)`
: Adds the element m to the set s if it is not already there. Returns the new set.

`MAX(t)`
: Returns the maximum value of the type t.

`MIN(t)`
: Returns the minimum value of the type t.

`ODD(i)`
: Returns boolean TRUE if i is an odd number.

`ORD(x)`
: Returns the ordinal value of its argument. For example, the ordinal value of a character is its ASCII value (on machines supporting the ASCII character set). x must be of an ordered type, which include integral, character and enumerated types.

`SIZE(x)`
: Returns the size of its argument. x can be a variable or a type.

`TRUNC(r)`
: Returns the integral part of r.

`VAL(t,i)`
: Returns the member of the type t whose ordinal value is i.

Warning: Sets and their operations are not yet supported, so GDB treats the use of procedures `INCL` and `EXCL` as an error.

Constants

GDB allows you to express the constants of Modula-2 in the following ways:

- Integer constants are simply a sequence of digits. When used in an expression, a constant is interpreted to be type-compatible with the rest of the expression. Hexadecimal integers are

specified by a trailing `H', and octal integers by a trailing `B'.

- Floating point constants appear as a sequence of digits, followed by a decimal point and another sequence of digits. An optional exponent can then be specified, in the form `E[+|-]nnn', where `[+|-]nnn' is the desired exponent. All of the digits of the floating point constant must be valid decimal (base 10) digits.
- Character constants consist of a single character enclosed by a pair of like quotes, either single (') or double ("). They may also be expressed by their ordinal value (their ASCII value, usually) followed by a `C'.
- String constants consist of a sequence of characters enclosed by a pair of like quotes, either single (') or double ("). Escape sequences in the style of C are also allowed. See section C and C++ constants, for a brief explanation of escape sequences.
- Enumerated constants consist of an enumerated identifier.
- Boolean constants consist of the identifiers TRUE and FALSE.
- Pointer constants consist of integral values only.
- Set constants are not yet supported.

Modula-2 defaults

If type and range checking are set automatically by GDB, they both default to `on` whenever the working language changes to Modula-2. This happens regardless of whether you, or GDB, selected the working language.

If you allow GDB to set the language automatically, then entering code compiled from a file whose name ends with `` `.mod' `` sets the working language to Modula-2. See section Having GDB infer the source language, for further details.

Deviations from standard Modula-2

A few changes have been made to make Modula-2 programs easier to debug. This is done primarily via loosening its type strictness:

- Unlike in standard Modula-2, pointer constants can be formed by integers. This allows you to modify pointer variables during debugging. (In standard Modula-2, the actual address contained in a pointer variable is hidden from you; it can only be modified through direct assignment to another pointer variable or expression that returned a pointer.)
- C escape sequences can be used in strings and characters to represent non-printable characters. GDB prints out strings with these escape sequences embedded. Single non-printable characters are printed using the `CHR(nnn)' format.
- The assignment operator (`:=`) returns the value of its right-hand argument.
- All built-in procedures both modify *and* return their argument.

Modula-2 type and range checks

Warning: in this release, GDB does not yet perform type or range checking.

GDB considers two Modula-2 variables type equivalent if:

- They are of types that have been declared equivalent via a `TYPE t1 = t2` statement
- They have been declared on the same line. (Note: This is true of the GNU Modula-2 compiler, but it may not be true of other compilers.)

As long as type checking is enabled, any attempt to combine variables whose types are not equivalent is an error.

Range checking is done on all mathematical operations, assignment, array index bounds, and all built-in functions and procedures.

The scope operators `::` and `.`

There are a few subtle differences between the Modula-2 scope operator (`.`) and the GDB scope operator (`::`). The two have similar syntax:

```
module . id
scope :: id
```

where scope is the name of a module or a procedure, module the name of a module, and id is any declared identifier within your program, except another module.

Using the `::` operator makes GDB search the scope specified by scope for the identifier id. If it is not found in the specified scope, then GDB searches all scopes enclosing the one specified by scope.

Using the `.` operator makes GDB search the current scope for the identifier specified by id that was imported from the definition module specified by module. With this operator, it is an error if the identifier id was not imported from definition module module, or if id is not an identifier in module.

GDB and Modula-2

Some GDB commands have little use when debugging Modula-2 programs. Five subcommands of `set print` and `show print` apply specifically to C and C++: `vtbl', `demangle', `asm-demangle', `object', and `union'. The first four apply to C++, and the last to the C `union` type, which has no direct analogue in Modula-2.

The @ operator (see section Expressions), while available while using any language, is not useful with Modula-2. Its intent is to aid the debugging of **dynamic arrays**, which cannot be created in Modula-2 as they can in C or C++. However, because an address can be specified by an integral constant, the construct `{type}adrexp' is still useful. (see section Expressions)

In GDB scripts, the Modula-2 inequality operator # is interpreted as the beginning of a comment. Use <> instead.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Examining the Symbol Table

The commands described in this section allow you to inquire about the symbols (names of variables, functions and types) defined in your program. This information is inherent in the text of your program and does not change as your program executes. GDB finds it in your program's symbol table, in the file indicated when you started GDB (see section Choosing files), or by one of the file-management commands (see section Commands to specify files).

Occasionally, you may need to refer to symbols that contain unusual characters, which GDB ordinarily treats as word delimiters. The most frequent case is in referring to static variables in other source files (see section Program variables). File names are recorded in object files as debugging symbols, but GDB would ordinarily parse a typical file name, like `foo.c', as the three words `foo' `.' `c'. To allow GDB to recognize `foo.c' as a single symbol, enclose it in single quotes; for example,

```
p 'foo.c'::x
```

looks up the value of `x` in the scope of the file `foo.c'.

`info address symbol`

Describe where the data for symbol is stored. For a register variable, this says which register it is kept in. For a non-register local variable, this prints the stack-frame offset at which the variable is always stored. Note the contrast with `print &symbol', which does not work at all for a register variable, and for a stack local variable prints the exact address of the current instantiation of the variable.

`whatis exp`

Print the data type of expression exp. exp is not actually evaluated, and any side-effecting operations (such as assignments or function calls) inside it do not take place. See section Expressions.

`whatis`

Print the data type of `$`, the last value in the value history.

`ptype typename`

Print a description of data type typename. typename may be the name of a type, or for C code it may have the form `class class-name', `struct struct-tag', `union union-tag' or `enum enum-tag'.

`ptype exp`

`ptype`

Print a description of the type of expression exp. `ptype` differs from `whatis` by printing a detailed description, instead of just the name of the type. For example, for this variable declaration:


```
struct complex {double real; double imag;} v;
```

the two commands give this output:

```
(gdb) whatis v
type = struct complex
(gdb) ptype v
type = struct complex {
    double real;
    double imag;
}
```

As with `whatis`, using `ptype` without an argument refers to the type of `$`, the last value in the value history.

`info types regexp`

`info types`

Print a brief description of all types whose name matches regexp (or all types in your program, if you supply no argument). Each complete typename is matched as though it were a complete line; thus, `i type value' gives information on all types in your program whose name includes the string `value`, but `i type ^value$' gives information only on types whose complete name is `value`. This command differs from `ptype` in two ways: first, like `whatis`, it does not print a detailed description; second, it lists all source files where a type is defined.

`info source`

Show the name of the current source file--that is, the source file for the function containing the current point of execution--and the language it was written in.

`info sources`

Print the names of all source files in your program for which there is debugging information, organized into two lists: files whose symbols have already been read, and files whose symbols will be read when needed.

`info functions`

Print the names and data types of all defined functions.

`info functions regexp`

Print the names and data types of all defined functions whose names contain a match for regular expression regexp. Thus, `info fun step' finds all functions whose names include `step`; `info fun ^step' finds those whose names start with `step`.

`info variables`

Print the names and data types of all variables that are declared outside of functions (i.e., excluding local variables).

`info variables regexp`

Print the names and data types of all variables (except for local variables) whose names contain a match for regular expression regexp. Some systems allow individual object files that make up your program to be replaced without stopping and restarting your program. For example, in VxWorks you can simply recompile a defective object file and keep on running. If you are running on one of

these systems, you can allow GDB to reload the symbols for automatically relinked modules:

`set symbol-reloading on`

Replace symbol definitions for the corresponding source file when an object file with a particular name is seen again.

`set symbol-reloading off`

Do not replace symbol definitions when re-encountering object files of the same name. This is the default state; if you are not running on a system that permits automatically relinking modules, you should leave `symbol-reloading` off, since otherwise GDB may discard symbols when linking large programs, that may contain several modules (from different directories or libraries) with the same name.

`show symbol-reloading`

Show the current `on` or `off` setting.

`maint print symbols filename`

`maint print psymbols filename`

`maint print msymbols filename`

Write a dump of debugging symbol data into the file filename. These commands are used to debug the GDB symbol-reading code. Only symbols with debugging data are included. If you use `maint print symbols', GDB includes all the symbols for which it has already collected full details: that is, filename reflects symbols for only those files whose symbols GDB has read. You can use the command `info sources` to find out which files these are. If you use `maint print psymbols' instead, the dump shows information about symbols that GDB only knows partially--that is, symbols defined in files that GDB has skimmed, but not yet read completely. Finally, `maint print msymbols' dumps just the minimal symbol information required for each object file from which GDB has read some symbols. See section Commands to specify files, for a discussion of how GDB reads symbols (in the description of `symbol-file`).

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Altering Execution

Once you think you have found an error in your program, you might want to find out for certain whether correcting the apparent error would lead to correct results in the rest of the run. You can find the answer by experiment, using the GDB features for altering execution of the program.

For example, you can store new values into variables or memory locations, give your program a signal, restart it at a different address, or even return prematurely from a function.

Assignment to variables

To alter the value of a variable, evaluate an assignment expression. See section Expressions. For example,

```
print x=4
```

stores the value 4 into the variable `x`, and then prints the value of the assignment expression (which is 4). See section Using GDB with Different Languages, for more information on operators in supported languages.

If you are not interested in seeing the value of the assignment, use the `set` command instead of the `print` command. `set` is really the same as `print` except that the expression's value is not printed and is not put in the value history (see section Value history). The expression is evaluated only for its effects.

If the beginning of the argument string of the `set` command appears identical to a `set` subcommand, use the `set variable` command instead of just `set`. This command is identical to `set` except for its lack of subcommands. For example, if your program has a variable `width`, you get an error if you try to set a new value with just `set width=13', because GDB has the command `set width`:

```
(gdb) whatis width
type = double
(gdb) p width
$4 = 13
(gdb) set width=47
Invalid syntax in expression.
```

The invalid expression, of course, is `=47'. In order to actually set the program's variable `width`, use

```
(gdb) set var width=47
```

GDB allows more implicit conversions in assignments than C; you can freely store an integer value into

a pointer variable or vice versa, and you can convert any structure to any other structure that is the sa length or shorter.

To store values into arbitrary places in memory, use the `{...}' construct to generate a value of specified type at a specified address (see section Expressions). For example, `{int}0x83040` refers to memory location `0x83040` as an integer (which implies a certain size and representation in memory), and

```
set {int}0x83040 = 4
```

stores the value 4 into that memory location.

Continuing at a different address

Ordinarily, when you continue your program, you do so at the place where it stopped, with the `continue` command. You can instead continue at an address of your own choosing, with the following commands:

`jump linespec`

Resume execution at line linespec. Execution stops again immediately if there is a breakpoint there. See section Printing source lines, for a description of the different forms of linespec. It is common practice to use the `tbreak` command in conjunction with `jump`. See section Setting breakpoints. The `jump` command does not change the current stack frame, or the stack pointer, or the contents of any memory location or any register other than the program counter. If line linespec is in a different function from the one currently executing, the results may be bizarre if the two functions expect different patterns of arguments or of local variables. For this reason, the `jump` command requests confirmation if the specified line is not in the function currently executing. However, even bizarre results are predictable if you are well acquainted with the machine-language code of your program.

`jump *address`

Resume execution at the instruction at address address.

You can get much the same effect as the `jump` command by storing a new value into the register `$pc`. The difference is that this does not start your program running; it only changes the address of where it *will* run when you continue. For example,

```
set $pc = 0x485
```

makes the next `continue` command or stepping command execute at address `0x485`, rather than at the address where your program stopped. See section Continuing and stepping.

The most common occasion to use the `jump` command is to back up--perhaps with more breakpoints set--over a portion of a program that has already executed, in order to examine its execution in more detail.

Giving your program a signal

`signal signal`

> Resume execution where your program stopped, but immediately give it the signal signal. signal can be the name or the number of a signal. For example, on many systems `signal 2` and `signal SIGINT` are both ways of sending an interrupt signal. Alternatively, if signal is zero, continue execution without giving a signal. This is useful when your program stopped on account of a signal and would ordinary see the signal when resumed with the `continue` command; `signal 0' causes it to resume without a signal. `signal` does not repeat when you press RET a second time after executing the command.

Invoking the `signal` command is not the same as invoking the `kill` utility from the shell. Sending a signal with `kill` causes GDB to decide what to do with the signal depending on the signal handling tables (see section Signals). The `signal` command passes the signal directly to your program.

Returning from a function

`return`

`return expression`

> You can cancel execution of a function call with the `return` command. If you give an expression argument, its value is used as the function's return value.

When you use `return`, GDB discards the selected stack frame (and all frames within it). You can think of this as making the discarded frame return prematurely. If you wish to specify a value to be returned, give that value as the argument to `return`.

This pops the selected stack frame (see section Selecting a frame), and any other frames inside of it, leaving its caller as the innermost remaining frame. That frame becomes selected. The specified value is stored in the registers used for returning values of functions.

The `return` command does not resume execution; it leaves the program stopped in the state that would exist if the function had just returned. In contrast, the `finish` command (see section Continuing and stepping) resumes execution until the selected stack frame returns naturally.

Calling program functions

`call expr`

> Evaluate the expression expr without displaying `void` returned values.

You can use this variant of the `print` command if you want to execute a function from your program, but without cluttering the output with `void` returned values. If the result is not void, it is printed and saved in the value history.

For the A29K, a user-controlled variable `call_scratch_address`, specifies the location of a

scratch area to be used when GDB calls a function in the target. This is necessary because the usual method of putting the scratch area on the stack does not work in systems that have separate instructio and data spaces.

Patching programs

By default, GDB opens the file containing your program's executable code (or the corefile) read-only. This prevents accidental alterations to machine code; but it also prevents you from intentionally patching your program's binary.

If you'd like to be able to patch the binary, you can specify that explicitly with the `set write` command. For example, you might want to turn on internal debugging flags, or even to make emergency repairs.

`set write on`

`set write off`

If you specify `set write on', GDB opens executable and core files for both reading and writing; if you specify `set write off' (the default), GDB opens them read-only. If you have already loaded a file, you must load it again (using the `exec-file` or `core-file` command) after changing `set write`, for your new setting to take effect.

`show write`

Display whether executable files and core files are opened for writing as well as reading.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

GDB Files

GDB needs to know the file name of the program to be debugged, both in order to read its symbol table and in order to start your program. To debug a core dump of a previous run, you must also tell GDB the name of the core dump file.

Commands to specify files

You may want to specify executable and core dump file names. The usual way to do this is at start-up time, using the arguments to GDB's start-up commands (see section Getting In and Out of GDB).

Occasionally it is necessary to change to a different file during a GDB session. Or you may run GDB and forget to specify a file you want to use. In these situations the GDB commands to specify new files are useful.

`file filename`

Use filename as the program to be debugged. It is read for its symbols and for the contents of pure memory. It is also the program executed when you use the `run` command. If you do not specify a directory and the file is not found in the GDB working directory, GDB uses the environment variable `PATH` as a list of directories to search, just as the shell does when looking for a program to run. You can change the value of this variable, for both GDB and your program, using the `path` command. On systems with memory-mapped files, an auxiliary file `` `filename.syms' `` may hold symbol table information for filename. If so, GDB maps in the symbol table from `` `filename.syms' ``, starting up more quickly. See the descriptions of the file options `-mapped' and `-readnow' (available on the command line, and with the commands `file`, `symbol-file`, or `add-symbol-file`, described below), for more information.

`file`

`file` with no argument makes GDB discard any information it has on both executable file and the symbol table.

`exec-file [ filename ]`

Specify that the program to be run (but not the symbol table) is found in filename. GDB searches the environment variable `PATH` if necessary to locate your program. Omitting filename means to discard information on the executable file.

`symbol-file [ filename ]`

Read symbol table information from file filename. `PATH` is searched when necessary. Use the `file` command to get both symbol table and program to run from the same file. `symbol-file` with no argument clears out GDB information on your program's symbol table. The `symbol-file` command causes GDB to forget the contents of its convenience variables, the value history, and all breakpoints and auto-display expressions. This is because they may contain

pointers to the internal data recording symbols and data types, which are part of the old symbol table data being discarded inside GDB. `symbol-file` does not repeat if you press RET again after executing it once. When GDB is configured for a particular environment, it understands debugging information in whatever format is the standard generated for that environment; you may use either a GNU compiler, or other compilers that adhere to the local conventions. Best results are usually obtained from GNU compilers; for example, using `gcc` you can generate debugging information for optimized code. For most kinds of object files, with the exception of old SVR3 systems using COFF, the `symbol-file` command does not normally read the symbol table in full right away. Instead, it scans the symbol table quickly to find which source files and which symbols are present. The details are read later, one source file at a time, as they are needed. The purpose of this two-stage reading strategy is to make GDB start up faster. For the most part, it is invisible except for occasional pauses while the symbol table details for a particular source file are being read. (The `set verbose` command can turn these pauses into messages if desired. See section Optional warnings and messages.) We have not implemented the two-stage strategy for COFF yet. When the symbol table is stored in COFF format, `symbol-file` reads the symbol table data in full right away. Note that "stabs-in-COFF" still does the two-stage strategy, since the debug info is actually in stabs format.

`symbol-file filename [ -readnow ] [ -mapped ]`

`file filename [ -readnow ] [ -mapped ]`

You can override the GDB two-stage strategy for reading symbol tables by using the `-readnow' option with any of the commands that load symbol table information, if you want to be sure GDB has the entire symbol table available. If memory-mapped files are available on your system through the `mmap` system call, you can use another option, `-mapped', to cause GDB to write the symbols for your program into a reusable file. Future GDB debugging sessions map in symbol information from this auxiliary symbol file (if the program has not changed), rather than spending time reading the symbol table from the executable program. Using the `-mapped' option has the same effect as starting GDB with the `-mapped' command-line option. You can use both options together, to make sure the auxiliary symbol file has all the symbol information for your program. The auxiliary symbol file for a program called myprog is called `myprog.syms'. Once this file exists (so long as it is newer than the corresponding executable), GDB always attempts to use it when you debug myprog; no special options or commands are needed. The `.syms' file is specific to the host machine where you run GDB. It holds an exact image of the internal GDB symbol table. It cannot be shared across multiple host platforms.

`core-file [ filename ]`

Specify the whereabouts of a core dump file to be used as the "contents of memory". Traditionally, core files contain only some parts of the address space of the process that generated them; GDB can access the executable file itself for other parts. `core-file` with no argument specifies that no core file is to be used. Note that the core file is ignored when your program is actually running under GDB. So, if you have been running your program and you wish to debug a core file instead, you must kill the subprocess in which the program is running. To do this, use the `kill` command (see section Killing the child process).

`add-symbol-file filename address`

`add-symbol-file filename address [ -readnow ] [ -mapped ]`

The `add-symbol-file` command reads additional symbol table information from the file filename. You would use this command when filename has been dynamically loaded (by some other means) into the program that is running. address should be the memory address at which the file has been loaded; GDB cannot figure this out for itself. You can specify address as an expression. The symbol table of the file filename is added to the symbol table originally read with the `symbol-file` command. You can use the `add-symbol-file` command any number of times; the new symbol data thus read keeps adding to the old. To discard all old symbol data instead, use the `symbol-file` command. `add-symbol-file` does not repeat if you press RET after using it. You can use the `-mapped' and `-readnow' options just as with the `symbol-file` command, to change how GDB manages the symbol table information for filename.

`add-shared-symbol-file`

The `add-shared-symbol-file` command can be used only under Harris' CXUX operating system for the Motorola 88k. GDB automatically looks for shared libraries, however if GDB does not find yours, you can run `add-shared-symbol-file`. It takes no arguments.

`section`

The `section` command changes the base address of section SECTION of the exec file to ADDR. This can be used if the exec file does not contain section addresses, (such as in the a.out format), or when the addresses specified in the file itself are wrong. Each section must be changed separately. The "info files" command lists all the sections and their addresses.

`info files`

`info target`

`info files` and `info target` are synonymous; both print the current target (see section Specifying a Debugging Target), including the names of the executable and core dump files currently in use by GDB, and the files from which symbols were loaded. The command `help target` lists all possible targets rather than current ones.

All file-specifying commands allow both absolute and relative file names as arguments. GDB always converts the file name to an absolute file name and remembers it that way.

GDB supports HP-UX, SunOS, SVr4, Irix 5, and IBM RS/6000 shared libraries. GDB automatically loads symbol definitions from shared libraries when you use the `run` command, or when you examine a core file. (Before you issue the `run` command, GDB does not understand references to a function in a shared library, however--unless you are debugging a core file).

`info share`

`info sharedlibrary`

Print the names of the shared libraries which are currently loaded.

`sharedlibrary regex`

`share regex`

Load shared object library symbols for files matching a Unix regular expression. As with files loaded automatically, it only loads shared libraries required by your program for a core file or after typing `run`. If regex is omitted all shared libraries required by your program are loaded.

Errors reading symbol files

While reading a symbol file, GDB occasionally encounters problems, such as symbol types it does not recognize, or known bugs in compiler output. By default, GDB does not notify you of such problems, since they are relatively common and primarily of interest to people debugging compilers. If you are interested in seeing information about ill-constructed symbol tables, you can either ask GDB to print only one message about each such type of problem, no matter how many times the problem occurs; or you can ask GDB to print more messages, to see how many times the problems occur, with the `set complaints` command (see section Optional warnings and messages).

The messages currently printed, and their meanings, include:

`inner block not inside outer block in symbol`

> The symbol information shows where symbol scopes begin and end (such as at the start of a function or a block of statements). This error indicates that an inner scope block is not fully contained in its outer scope blocks. GDB circumvents the problem by treating the inner block as if it had the same scope as the outer block. In the error message, symbol may be shown as "`(don't know)`" if the outer block is not a function.

`block at address out of order`

> The symbol information for symbol scope blocks should occur in order of increasing addresses. This error indicates that it does not do so. GDB does not circumvent this problem, and has trouble locating symbols in the source file whose symbols it is reading. (You can often determine what source file is affected by specifying `set verbose on`. See section Optional warnings and messages.)

`bad block start address patched`

> The symbol information for a symbol scope block has a start address smaller than the address of the preceding source line. This is known to occur in the SunOS 4.1.1 (and earlier) C compiler. GDB circumvents the problem by treating the symbol scope block as starting on the previous source line.

`bad string table offset in symbol n`

> Symbol number n contains a pointer into the string table which is larger than the size of the string table. GDB circumvents the problem by considering the symbol to have the name `foo`, which may cause other problems if many symbols end up with this name.

`unknown symbol type 0xnn`

> The symbol information contains new data types that GDB does not yet know how to read. `0xnn` is the symbol type of the misunderstood information, in hexadecimal. GDB circumvents the error by ignoring this symbol information. This usually allows you to debug your program, though certain symbols are not accessible. If you encounter such a problem and feel like debugging it, you can debug `gdb` with itself, breakpoint on `complain`, then go up to the function `read_dbx_symtab` and examine `*bufp` to see the symbol.

`stub type has NULL name`

> GDB could not find the full definition for a struct or class.

`const/volatile indicator missing (ok if using g++ v1.x), got...`

The symbol information for a C++ member function is missing some information that recent versions of the compiler should have output for it.

`info mismatch between compiler and debugger`

GDB could not parse a type specification output by the compiler.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Specifying a Debugging Target

A **target** is the execution environment occupied by your program. Often, GDB runs in the same host environment as your program; in that case, the debugging target is specified as a side effect when you use the `file` or `core` commands. When you need more flexibility--for example, running GDB on a physically separate host, or controlling a standalone system over a serial port or a realtime system over a TCP/IP connection--you can use the `target` command to specify one of the target types configured for GDB (see section Commands for managing targets).

Active targets

There are three classes of targets: processes, core files, and executable files. GDB can work concurrently on up to three active targets, one in each class. This allows you to (for example) start a process and inspect its activity without abandoning your work on a core file.

For example, if you execute `gdb a.out', then the executable file `a.out` is the only active target. If you designate a core file as well--presumably from a prior run that crashed and coredumped--then GDB has two active targets and uses them in tandem, looking first in the corefile target, then in the executable file, to satisfy requests for memory addresses. (Typically, these two classes of target are complementary, since core files contain only a program's read-write memory--variables and so on--plus machine status, while executable files contain only the program text and initialized data.)

When you type `run`, your executable file becomes an active process target as well. When a process target is active, all GDB commands requesting memory addresses refer to that target; addresses in an active core file or executable file target are obscured while the process target is active.

Use the `core-file` and `exec-file` commands to select a new core file or executable target (see section Commands to specify files). To specify as a target a process that is already running, use the `attach` command (see section Debugging an already-running process).

Commands for managing targets

`target type parameters`

Connects the GDB host environment to a target machine or process. A target is typically a protocol for talking to debugging facilities. You use the argument type to specify the type or protocol of the target machine. Further parameters are interpreted by the target protocol, but typically include things like device names or host names to connect with, process numbers, and baud rates. The `target` command does not repeat if you press RET again after executing the command.

`help target`

Displays the names of all targets available. To display targets currently selected, use either `info target` or `info files` (see section Commands to specify files).

`help target name`

Describe a particular target, including any parameters necessary to select it.

`set gnutarget args`

GDB uses its own library BFD to read your files. GDB knows whether it is reading an **executable**, a **core**, or a **.o** file; however, you can specify the file format with the `set gnutarget` command. Unlike most `target` commands, with `gnutarget` the `target` refers to a program, not a machine. *Warning:* To specify a file format with `set gnutarget`, you must know the actual BFD name. See section Commands to specify files.

`show gnutarget`

Use the `show gnutarget` command to display what file format `gnutarget` is set to read. If you have not set `gnutarget`, GDB will determine the file format for each file automatically, and `show gnutarget` displays `The current BDF target is "auto"'.

Here are some common targets (available, or not, depending on the GDB configuration):

`target exec program`

An executable file. `target exec program' is the same as `exec-file program'.

`target core filename`

A core dump file. `target core filename' is the same as `core-file filename'.

`target remote dev`

Remote serial target in GDB-specific protocol. The argument dev specifies what serial device to use for the connection (e.g. `` `/dev/ttya' ``). See section Remote debugging. `target remote` now supports the `load` command. This is only useful if you have some other way of getting the stub to the target system, and you can put it somewhere in memory where it won't get clobbered by the download.

`target sim`

CPU simulator. See section Simulated CPU target.

The following targets are all CPU-specific, and only available for specific configurations.

`target abug dev`

ABug ROM monitor for M68K.

`target adapt dev`

Adapt monitor for A29K.

`target amd-eb dev speed PROG`

Remote PC-resident AMD EB29K board, attached over serial lines. dev is the serial device, as for `target remote`; speed allows you to specify the linespeed; and PROG is the name of the program to be debugged, as it appears to DOS on the PC. See section The EBMON protocol for AMD29K.

`target array dev`

Array Tech LSI33K RAID controller board.

`target bug dev`

BUG monitor, running on a MVME187 (m88k) board.

`target cpu32bug dev`

CPU32BUG monitor, running on a CPU32 (M68K) board.

`target dbug dev`

dBUG ROM monitor for Motorola ColdFire.

`target ddb dev`

NEC's DDB monitor for Mips Vr4300.

`target dink32 dev`

DINK32 ROM monitor for PowerPC.

`target e7000 dev`

E7000 emulator for Hitachi H8 and SH.

`target es1800 dev`

ES-1800 emulator for M68K.

`target est dev`

EST-300 ICE monitor, running on a CPU32 (M68K) board.

`target hms dev`

A Hitachi SH, H8/300, or H8/500 board, attached via serial line to your host. Use special commands `device` and `speed` to control the serial line and the communications speed used. See section GDB and Hitachi microprocessors.

`target lsi dev`

LSI ROM monitor for Mips.

`target m32r dev`

Mitsubishi M32R/D ROM monitor.

`target mips dev`

IDT/SIM ROM monitor for Mips.

`target mon960 dev`

MON960 monitor for Intel i960.

`target nindy devicename`

An Intel 960 board controlled by a Nindy Monitor. devicename is the name of the serial device to use for the connection, e.g. `` `/dev/ttya' ``. See section GDB with a remote i960 (Nindy).

`target nrom dev`

NetROM ROM emulator. This target only supports downloading.

`target op50n dev`

OP50N monitor, running on an OKI HPPA board.

`target pmon dev`

PMON ROM monitor for Mips.

`target ppcbug dev`

`target ppcbug1 dev`

PPCBUG ROM monitor for PowerPC.

`target r3900 dev`

Densan DVE-R3900 ROM monitor for Toshiba R3900 Mips.

`target rdi dev`

ARM Angel monitor, via RDI library interface.

`target rdp dev`

ARM Demon monitor.

`target rom68k dev`

ROM 68K monitor, running on an M68K IDP board.

`target rombug dev`

ROMBUG ROM monitor for OS/9000.

`target sds dev`

SDS monitor, running on a PowerPC board (such as Motorola's ADS).

`target sparclite dev`

Fujitsu sparclite boards, used only for the purpose of loading. You must use an additional command to debug the program. For example: target remote dev using GDB standard remote protocol.

`target sh3 dev`

`target sh3e dev`

Hitachi SH-3 and SH-3E target systems.

`target st2000 dev speed`

A Tandem ST2000 phone switch, running Tandem's STDBUG protocol. dev is the name of the device attached to the ST2000 serial line; speed is the communication line speed. The arguments are not used if GDB is configured to connect to the ST2000 using TCP or Telnet. See section GDB with a Tandem ST2000.

`target udi keyword`

Remote AMD29K target, using the AMD UDI protocol. The keyword argument specifies which 29K board or simulator to use. See section The UDI protocol for AMD29K.

`target vxworks machinename`

A VxWorks system, attached via TCP/IP. The argument machinename is the target system's machine name or IP address. See section GDB and VxWorks.

`target w89k dev`

W89K monitor, running on a Winbond HPPA board.

Different targets are available on different configurations of GDB; your configuration may have more or

fewer targets.

Many remote targets require you to download the executable's code once you've successfully established a connection.

`load filename`

Depending on what remote debugging facilities are configured into GDB, the `load` command may be available. Where it exists, it is meant to make filename (an executable) available for debugging on the remote system--by downloading, or dynamic linking, for example. `load` also records the filename symbol table in GDB, like the `add-symbol-file` command. If your GDB does not have a `load` command, attempting to execute it gets the error message "`You can't do that when your target is ...`" The file is loaded at whatever address is specified in the executable. For some object file formats, you can specify the load address when you link the program; for other formats, like a.out, the object file format specifies a fixed address. On VxWorks, `load` links filename dynamically on the current target system as well as adding its symbols in GDB. With the Nindy interface to an Intel 960 board, `load` downloads filename to the 960 as well as adding its symbols in GDB. When you select remote debugging to a Hitachi SH, H8/300, or H8/500 board (see section GDB and Hitachi microprocessors), the `load` command downloads your program to the Hitachi board and also opens it as the current executable target for GDB on your host (like the `file` command). `load` does not repeat if you press RET again after using it.

Choosing target byte order

Some types of processors, such as the MIPS, PowerPC, and Hitachi SH, offer the ability to run either big-endian or little-endian byte orders. Usually the executable or symbol will include a bit to designate the endian-ness, and you will not need to worry about which to use. However, you may still find it useful to adjust GDB's idea of processor endian-ness manually.

`set endian big`

Instruct GDB to assume the target is big-endian.

`set endian little`

Instruct GDB to assume the target is little-endian.

`set endian auto`

Instruct GDB to use the byte order associated with the executable.

`show endian`

Display GDB's current idea of the target byte order.

Note that these commands merely adjust interpretation of symbolic data on the host, and that they have absolutely no effect on the target system.

Remote debugging

If you are trying to debug a program running on a machine that cannot run GDB in the usual way, it is often useful to use remote debugging. For example, you might use remote debugging on an operating system kernel, or on a small system which does not have a general purpose operating system powerful enough to run a full-featured debugger.

Some configurations of GDB have special serial or TCP/IP interfaces to make this work with particular debugging targets. In addition, GDB comes with a generic serial protocol (specific to GDB, but not specific to any particular target system) which you can use if you write the remote stubs--the code that runs on the remote system to communicate with GDB.

Other remote targets may be available in your configuration of GDB; use `help target` to list them.

The GDB remote serial protocol

To debug a program running on another machine (the debugging **target** machine), you must first arrange for all the usual prerequisites for the program to run by itself. For example, for a C program, you need:

1. A startup routine to set up the C runtime environment; these usually have a name like `` `crt0' ``. The startup routine may be supplied by your hardware supplier, or you may have to write your own.
2. You probably need a C subroutine library to support your program's subroutine calls, notably managing input and output.
3. A way of getting your program to the other machine--for example, a download program. These are often supplied by the hardware manufacturer, but you may have to write your own from hardware documentation.

The next step is to arrange for your program to use a serial port to communicate with the machine where GDB is running (the **host** machine). In general terms, the scheme looks like this:

On the host,

GDB already understands how to use this protocol; when everything else is set up, you can simply use the `target remote' command (see section Specifying a Debugging Target).

On the target,

you must link with your program a few special-purpose subroutines that implement the GDB remote serial protocol. The file containing these subroutines is called a **debugging stub**. On certain remote targets, you can use an auxiliary program `gdbserver` instead of linking a stub into your program. See section Using the `gdbserver` program, for details.

The debugging stub is specific to the architecture of the remote machine; for example, use `` `sparc-stub.c' `` to debug programs on SPARC boards.

These working remote stubs are distributed with GDB:

`i386-stub.c`

For Intel 386 and compatible architectures.

`m68k-stub.c`
: For Motorola 680x0 architectures.

`sh-stub.c`
: For Hitachi SH architectures.

`sparc-stub.c`
: For SPARC architectures.

`sparcl-stub.c`
: For Fujitsu SPARCLITE architectures.

The `README' file in the GDB distribution may list other recently added stubs.

What the stub can do for you

The debugging stub for your architecture supplies these three subroutines:

`set_debug_traps`
: This routine arranges for `handle_exception` to run when your program stops. You must call this subroutine explicitly near the beginning of your program.

`handle_exception`
: This is the central workhorse, but your program never calls it explicitly--the setup code arranges for `handle_exception` to run when a trap is triggered. `handle_exception` takes control when your program stops during execution (for example, on a breakpoint), and mediates communications with GDB on the host machine. This is where the communications protocol is implemented; `handle_exception` acts as the GDB representative on the target machine; it begins by sending summary information on the state of your program, then continues to execute, retrieving and transmitting any information GDB needs, until you execute a GDB command that makes your program resume; at that point, `handle_exception` returns control to your own code on the target machine.

`breakpoint`
: Use this auxiliary subroutine to make your program contain a breakpoint. Depending on the particular situation, this may be the only way for GDB to get control. For instance, if your target machine has some sort of interrupt button, you won't need to call this; pressing the interrupt button transfers control to `handle_exception`---in effect, to GDB. On some machines, simply receiving characters on the serial port may also trigger a trap; again, in that situation, you don't need to call `breakpoint` from your own program--simply running `target remote' from the host GDB session gets control. Call `breakpoint` if none of these is true, or if you simply want to make certain your program stops at a predetermined point for the start of your debugging session.

What you must do for the stub

The debugging stubs that come with GDB are set up for a particular chip architecture, but they have no information about the rest of your debugging target machine.

First of all you need to tell the stub how to communicate with the serial port.

`int getDebugChar()`

Write this subroutine to read a single character from the serial port. It may be identical to `getchar` for your target system; a different name is used to allow you to distinguish the two if you wish.

`void putDebugChar(int)`

Write this subroutine to write a single character to the serial port. It may be identical to `putchar` for your target system; a different name is used to allow you to distinguish the two if you wish.

If you want GDB to be able to stop your program while it is running, you need to use an interrupt-driven serial driver, and arrange for it to stop when it receives a `^C` (`\003', the control-C character). That is the character which GDB uses to tell the remote system to stop.

Getting the debugging target to return the proper status to GDB probably requires changes to the standard stub; one quick and dirty way is to just execute a breakpoint instruction (the "dirty" part is that GDB reports a `SIGTRAP` instead of a `SIGINT`).

Other routines you need to supply are:

`void exceptionHandler (int exception_number, void *exception_address)`

Write this function to install exception_address in the exception handling tables. You need to do this because the stub does not have any way of knowing what the exception handling tables on your target system are like (for example, the processor's table might be in ROM, containing entries which point to a table in RAM). exception_number is the exception number which should be changed; its meaning is architecture-dependent (for example, different numbers might represent divide by zero, misaligned access, etc). When this exception occurs, control should be transferred directly to exception_address, and the processor state (stack, registers, and so on) should be just as it is when a processor exception occurs. So if you want to use a jump instruction to reach exception_address, it should be a simple jump, not a jump to subroutine. For the 386, exception_address should be installed as an interrupt gate so that interrupts are masked while the handler runs. The gate should be at privilege level 0 (the most privileged level). The SPARC and 68k stubs are able to mask interrup themselves without help from `exceptionHandler`.

`void flush_i_cache()`

(sparc and sparclite only) Write this subroutine to flush the instruction cache, if any, on your target machine. If there is no instruction cache, this subroutine may be a no-op. On target machines that have instruction caches, GDB requires this function to make certain that the state of your program is stable.

You must also make sure this library routine is available:

`void *memset(void *, int, int)`

This is the standard library function `memset` that sets an area of memory to a known value. If you have one of the free versions of `libc.a`, `memset` can be found there; otherwise, you must either obtain it from your hardware manufacturer, or write your own.

If you do not use the GNU C compiler, you may need other standard library subroutines as well; this varies from one stub to another, but in general the stubs are likely to use any of the common library subroutines which `gcc` generates as inline code.

Putting it all together

In summary, when your program is ready to debug, you must follow these steps.

1. Make sure you have the supporting low-level routines (see section What you must do for the stub):

   ```
   getDebugChar, putDebugChar,
   flush_i_cache, memset, exceptionHandler.
   ```

2. Insert these lines near the top of your program:

   ```
   set_debug_traps();
   breakpoint();
   ```

3. For the 680x0 stub only, you need to provide a variable called `exceptionHook`. Normally you just use:

   ```
   void (*exceptionHook)() = 0;
   ```

   but if before calling `set_debug_traps`, you set it to point to a function in your program, that function is called when `GDB` continues after stopping on a trap (for example, bus error). The function indicated by `exceptionHook` is called with one parameter: an `int` which is the exception number.

4. Compile and link together: your program, the GDB debugging stub for your target architecture, and the supporting subroutines.
5. Make sure you have a serial connection between your target machine and the GDB host, and identify the serial port on the host.
6. Download your program to your target machine (or get it there by whatever means the manufacturer provides), and start it.
7. To start remote debugging, run GDB on the host machine, and specify as an executable file the program that is running in the remote machine. This tells GDB how to find your program's symbols and the contents of its pure text. Then establish communication using the `target remote` command. Its argument specifies how to communicate with the target machine--either via a devicename attached to a direct serial line, or a TCP port (usually to a terminal server which in turn has a serial line to the target). For example, to use a serial line connected to the device named `` `/dev/ttyb' ``:

   ```
   target remote /dev/ttyb
   ```

   To use a TCP connection, use an argument of the form `host:port`. For example, to connect to port 2828 on a terminal server named `manyfarms`:

   ```
   target remote manyfarms:2828
   ```

Now you can use all the usual commands to examine and change data and to step and continue the remote program.

To resume the remote program and stop debugging it, use the `detach` command.

Whenever GDB is waiting for the remote program, if you type the interrupt character (often C-C), GDB attempts to stop the program. This may or may not succeed, depending in part on the hardware and the serial drivers the remote system uses. If you type the interrupt character once again, GDB displays this prompt:

```
Interrupted while waiting for the program.
Give up (and stop debugging it)?  (y or n)
```

If you type y, GDB abandons the remote debugging session. (If you decide you want to try again later, you can use `target remote' again to connect once more.) If you type n, GDB goes back to waiting.

Communication protocol

The stub files provided with GDB implement the target side of the communication protocol, and the GDB side is implemented in the GDB source file `remote.c'. Normally, you can simply allow these subroutines to communicate, and ignore the details. (If you're implementing your own stub file, you can still ignore the details: start with one of the existing stub files. `sparc-stub.c' is the best organized, and therefore the easiest to read.)

However, there may be occasions when you need to know something about the protocol--for example, if there is only one serial port to your target machine, you might want your program to do something special if it recognizes a packet meant for GDB.

All GDB commands and responses (other than acknowledgements, which are single characters) are sent as a packet which includes a checksum. A packet is introduced with the character `$', and ends with the character `#' followed by a two-digit checksum:

```
$packet info#checksum
```

checksum is computed as the modulo 256 sum of the packet info characters.

When either the host or the target machine receives a packet, the first response expected is an acknowledgement: a single character, either `+' (to indicate the package was received correctly) or `-' (to request retransmission).

The host (GDB) sends commands, and the target (the debugging stub incorporated in your program) sends data in response. The target also sends data when your program stops.

Command packets are distinguished by their first character, which identifies the kind of command.

These are some of the commands currently supported (for a complete list of commands, look in `gdb/remote.c'):

`g`

Requests the values of CPU registers.

`G`

Sets the values of CPU registers.

`maddr,count`

Read count bytes at location addr.

`Maddr,count:...`

Write count bytes at location addr.

`c`

`caddr`

Resume execution at the current address (or at addr if supplied).

`s`

`saddr`

Step the target program for one instruction, from either the current program counter or from addr if supplied.

`k`

Kill the target program.

`?`

Report the most recent signal. To allow you to take advantage of the GDB signal handling commands, one of the functions of the debugging stub is to report CPU traps as the corresponding POSIX signal values.

`T`

Allows the remote stub to send only the registers that GDB needs to make a quick decision about single-stepping or conditional breakpoints. This eliminates the need to fetch the entire register set for each instruction being stepped through. GDB now implements a write-through cache for registers and only re-reads the registers if the target has run.

If you have trouble with the serial connection, you can use the command `set remotedebug`. This makes GDB report on all packets sent back and forth across the serial line to the remote machine. The packet-debugging information is printed on the GDB standard output stream. `set remotedebug off` turns it off, and `show remotedebug` shows you its current state.

Using the `gdbserver` program

`gdbserver` is a control program for Unix-like systems, which allows you to connect your program with a remote GDB via `target remote`---but without linking in the usual debugging stub.

`gdbserver` is not a complete replacement for the debugging stubs, because it requires essentially the same operating-system facilities that GDB itself does. In fact, a system that can run `gdbserver` to connect to a remote GDB could also run GDB locally! `gdbserver` is sometimes useful nevertheless, because it is a much smaller program than GDB itself. It is also easier to port than all of GDB, so you may be able to get started more quickly on a new system by using `gdbserver`. Finally, if you develop code for real-time systems, you may find that the tradeoffs involved in real-time operation make it more convenient to do as much development work as possible on another system, for example by cross-compiling. You can use `gdbserver` to make a similar choice for debugging.

GDB and `gdbserver` communicate via either a serial line or a TCP connection, using the standard GDB remote serial protocol.

On the target machine,

you need to have a copy of the program you want to debug. `gdbserver` does not need your program's symbol table, so you can strip the program if necessary to save space. GDB on the host system does all the symbol handling. To use the server, you must tell it how to communicate with GDB; the name of your program; and the arguments for your program. The syntax is:

```
target> gdbserver comm program [ args ... ]
```

comm is either a device name (to use a serial line) or a TCP hostname and portnumber. For example, to debug Emacs with the argument `foo.txt' and communicate with GDB over the serial port `/dev/com1':

```
target> gdbserver /dev/com1 emacs foo.txt
```

`gdbserver` waits passively for the host GDB to communicate with it. To use a TCP connection instead of a serial line:

```
target> gdbserver host:2345 emacs foo.txt
```

The only difference from the previous example is the first argument, specifying that you are communicating with the host GDB via TCP. The `host:2345' argument means that `gdbserver` is to expect a TCP connection from machine `host' to local TCP port 2345. (Currently, the `host' part is ignored.) You can choose any number you want for the port number as long as it does not conflict with any TCP ports already in use on the target system (for example, `23` is reserved for `telnet`).(3) You must use the same port number with the host GDB `target remote` command.

On the GDB host machine,

you need an unstripped copy of your program, since GDB needs symbols and debugging information. Start up GDB as usual, using the name of the local copy of your program as the first argument. (You may also need the `--baud' option if the serial line is running at anything other than 9600 bps.) After that, use `target remote` to establish communications with `gdbserver`. Its argument is either a device name (usually a serial device, like `/dev/ttyb'), or a TCP port descriptor in the form `host:PORT`. For example:

```
(gdb) target remote /dev/ttyb
```

communicates with the server via serial line `/dev/ttyb', and

```
(gdb) target remote the-target:2345
```

communicates via a TCP connection to port 2345 on host `the-target'. For TCP connections, you must start up `gdbserver` prior to using the `target remote` command. Otherwise you may get an error whose text depends on the host system, but which usually looks something like `Connection refused'.

Using the `gdbserve.nlm` program

`gdbserve.nlm` is a control program for NetWare systems, which allows you to connect your program with a remote GDB via `target remote`.

GDB and `gdbserve.nlm` communicate via a serial line, using the standard GDB remote serial protocol.

On the target machine,

you need to have a copy of the program you want to debug. `gdbserve.nlm` does not need your program's symbol table, so you can strip the program if necessary to save space. GDB on the host system does all the symbol handling. To use the server, you must tell it how to communicate with GDB; the name of your program; and the arguments for your program. The syntax is:

```
load gdbserve [ BOARD=board ] [ PORT=port ]
              [ BAUD=baud ] program [ args ... ]
```

board and port specify the serial line; baud specifies the baud rate used by the connection. port and node default to 0, baud defaults to 9600 bps. For example, to debug Emacs with the argument `foo.txt'and communicate with GDB over serial port number 2 or board 1 using a 19200 bps connection:

```
load gdbserve BOARD=1 PORT=2 BAUD=19200 emacs foo.txt
```

On the GDB host machine,

you need an unstripped copy of your program, since GDB needs symbols and debugging information. Start up GDB as usual, using the name of the local copy of your program as the first argument. (You may also need the `--baud' option if the serial line is running at anything other than 9600 bps. After that, use `target remote` to establish communications with `gdbserve.nlm`. Its argument is a device name (usually a serial device, like `` `/dev/ttyb' ``). For example:

```
(gdb) target remote /dev/ttyb
```

communications with the server via serial line `` `/dev/ttyb' ``.

GDB with a remote i960 (Nindy)

Nindy is a ROM Monitor program for Intel 960 target systems. When GDB is configured to control a remote Intel 960 using Nindy, you can tell GDB how to connect to the 960 in several ways:

- Through command line options specifying serial port, version of the Nindy protocol, and communications speed;
- By responding to a prompt on startup;
- By using the `target` command at any point during your GDB session. See section Commands for managing targets.

Startup with Nindy

If you simply start `gdb` without using any command-line options, you are prompted for what serial port to use, *before* you reach the ordinary GDB prompt:

```
Attach /dev/ttyNN -- specify NN, or "quit" to quit:
```

Respond to the prompt with whatever suffix (after `/dev/tty') identifies the serial port you want to use. You can, if you choose, simply start up with no Nindy connection by responding to the prompt with an empty line. If you do this and later wish to attach to Nindy, use `target` (see section Commands for managing targets).

Options for Nindy

These are the startup options for beginning your GDB session with a Nindy-960 board attached:

`-r port`

Specify the serial port name of a serial interface to be used to connect to the target system. This option is only available when GDB is configured for the Intel 960 target architecture. You may specify port as any of: a full pathname (e.g. `-r /dev/ttya'), a device name in `/dev' (e.g. `-r ttya'), or simply the unique suffix for a specific `tty` (e.g. `-r a').

`-O`

(An uppercase letter "O", not a zero.) Specify that GDB should use the "old" Nindy monitor protocol to connect to the target system. This option is only available when GDB is configured for the Intel 960 target architecture.

Warning: if you specify `-O', but are actually trying to connect to a target system that expects the newer protocol, the connection fails, appearing to be a speed mismatch. GDB repeatedly attempts to reconnect at several different line speeds. You can abort this process with an interrupt.

`-brk`

Specify that GDB should first send a BREAK signal to the target system, in an attempt to reset it, before connecting to a Nindy target.

Warning: Many target systems do not have the hardware that this requires; it only works with a few boards.

The standard `-b' option controls the line speed used on the serial port.

Nindy reset command

`reset`

For a Nindy target, this command sends a "break" to the remote target system; this is only useful if the target has been equipped with a circuit to perform a hard reset (or some other interesting action) when a break is detected.

The UDI protocol for AMD29K

GDB supports AMD's UDI ("Universal Debugger Interface") protocol for debugging the a29k processor family. To use this configuration with AMD targets running the MiniMON monitor, you need the program `MONTIP`, available from AMD at no charge. You can also use GDB with the UDI-conformant a29k simulator program `ISSTIP`, also available from AMD.

`target udi keyword`

> Select the UDI interface to a remote a29k board or simulator, where keyword is an entry in the AMD configuration file `` `udi_soc' ``. This file contains keyword entries which specify parameters used to connect to a29k targets. If the `` `udi_soc' `` file is not in your working directory, you must set the environment variable `UDICONF' to its pathname.

The EBMON protocol for AMD29K

AMD distributes a 29K development board meant to fit in a PC, together with a DOS-hosted monitor program called `EBMON`. As a shorthand term, this development system is called the "EB29K". To use GDB from a Unix system to run programs on the EB29K board, you must first connect a serial cable between the PC (which hosts the EB29K board) and a serial port on the Unix system. In the following, we assume you've hooked the cable between the PC's `` `COM1' `` port and `` `/dev/ttya' `` on the Unix system.

Communications setup

The next step is to set up the PC's port, by doing something like this in DOS on the PC:

```
C:\> MODE com1:9600,n,8,1,none
```

This example--run on an MS DOS 4.0 system--sets the PC port to 9600 bps, no parity, eight data bits, one stop bit, and no "retry" action; you must match the communications parameters when establishing the Unix end of the connection as well.

To give control of the PC to the Unix side of the serial line, type the following at the DOS console:

```
C:\> CTTY com1
```

(Later, if you wish to return control to the DOS console, you can use the command `CTTY con`---but you must send it over the device that had control, in our example over the `` `COM1' `` serial line).

From the Unix host, use a communications program such as `tip` or `cu` to communicate with the PC; for example,

```
cu -s 9600 -l /dev/ttya
```

The `cu` options shown specify, respectively, the linespeed and the serial port to use. If you use `tip` instead, your command line may look something like the following:


```
tip -9600 /dev/ttya
```

Your system may require a different name where we show `/dev/ttya' as the argument to `tip'. The communications parameters, including which port to use, are associated with the `tip' argument in the "remote" descriptions file--normally the system table `/etc/remote'.

Using the `tip' or `cu' connection, change the DOS working directory to the directory containing a copy of your 29K program, then start the PC program `EBMON' (an EB29K control program supplied with your board by AMD). You should see an initial display from `EBMON' similar to the one that follows, ending with the `EBMON' prompt `#'---

```
C:\> G:

G:\> CD \usr\joe\work29k

G:\USR\JOE\WORK29K> EBMON
Am29000 PC Coprocessor Board Monitor, version 3.0-18
Copyright 1990 Advanced Micro Devices, Inc.
Written by Gibbons and Associates, Inc.

Enter '?' or 'H' for help

PC Coprocessor Type   = EB29K
I/O Base              = 0x208
Memory Base           = 0xd0000

Data Memory Size      = 2048KB
Available I-RAM Range = 0x8000 to 0x1fffff
Available D-RAM Range = 0x80002000 to 0x801fffff

PageSize              = 0x400
Register Stack Size   = 0x800
Memory Stack Size     = 0x1800

CPU PRL               = 0x3
Am29027 Available     = No
Byte Write Available  = Yes

# ~.
```

Then exit the `cu' or `tip' program (done in the example by typing `~.' at the `EBMON' prompt). `EBMON' keeps running, ready for GDB to take over.

For this example, we've assumed what is probably the most convenient way to make sure the same 29K program is on both the PC and the Unix system: a PC/NFS connection that establishes "drive `G:'" on the PC as a file system on the Unix host. If you do not have PC/NFS or something similar connecting the two systems, you must arrange some other way--perhaps floppy-disk transfer--of getting the 29K

program from the Unix system to the PC; GDB does *not* download it over the serial line.

EB29K cross-debugging

Finally, `cd` to the directory containing an image of your 29K program on the Unix system, and start GDB---specifying as argument the name of your 29K program:

```
cd /usr/joe/work29k
gdb myfoo
```

Now you can use the `target` command:

```
target amd-eb /dev/ttya 9600 MYFOO
```

In this example, we've assumed your program is in a file called `` `myfoo' ``. Note that the filename given as the last argument to `target amd-eb` should be the name of the program as it appears to DOS. In our example this is simply `MYFOO`, but in general it can include a DOS path, and depending on your transfer mechanism may not resemble the name on the Unix side.

At this point, you can set any breakpoints you wish; when you are ready to see your program run on the 29K board, use the GDB command `run`.

To stop debugging the remote program, use the GDB `detach` command.

To return control of the PC to its console, use `tip` or `cu` once again, after your GDB session has concluded, to attach to `EBMON`. You can then type the command `q` to shut down `EBMON`, returning control to the DOS command-line interpreter. Type `CTTY con` to return command input to the main DOS console, and type ~. to leave `tip` or `cu`.

Remote log

The `target amd-eb` command creates a file `` `eb.log' `` in the current working directory, to help debug problems with the connection. `` `eb.log' `` records all the output from `EBMON`, including echoes of the commands sent to it. Running `tail -f' on this file in another window often helps to understand trouble with `EBMON`, or unexpected events on the PC side of the connection.

GDB with a Tandem ST2000

To connect your ST2000 to the host system, see the manufacturer's manual. Once the ST2000 is physically attached, you can run:

```
target st2000 dev speed
```

to establish it as your debugging environment. dev is normally the name of a serial device, such as `` `/dev/ttya' ``, connected to the ST2000 via a serial line. You can instead specify dev as a TCP connection (for example, to a serial line attached via a terminal concentrator) using the syntax `hostname:portnumber`.

The `load` and `attach` commands are *not* defined for this target; you must load your program into the ST2000 as you normally would for standalone operation. GDB reads debugging information (such as symbols) from a separate, debugging version of the program available on your host computer.

These auxiliary GDB commands are available to help you with the ST2000 environment:

`st2000 command`

Send a command to the STDBUG monitor. See the manufacturer's manual for available commands.

`connect`

Connect the controlling terminal to the STDBUG command monitor. When you are done interacting with STDBUG, typing either of two character sequences gets you back to the GDB command prompt: RET~. (Return, followed by tilde and period) or RET~C-d (Return, followed by tilde and control-D).

GDB and VxWorks

GDB enables developers to spawn and debug tasks running on networked VxWorks targets from a Unix host. Already-running tasks spawned from the VxWorks shell can also be debugged. GDB uses code that runs on both the Unix host and on the VxWorks target. The program `gdb` is installed and executed on the Unix host. (It may be installed with the name `vxgdb`, to distinguish it from a GDB for debugging programs on the host itself.)

`VxWorks-timeout args`

All VxWorks-based targets now support the option `vxworks-timeout`. This option is set by the user, and args represents the number of seconds GDB waits for responses to rpc's. You might use this if your VxWorks target is a slow software simulator or is on the far side of a thin network line.

The following information on connecting to VxWorks was current when this manual was produced; newer releases of VxWorks may use revised procedures.

To use GDB with VxWorks, you must rebuild your VxWorks kernel to include the remote debugging interface routines in the VxWorks library `` `rdb.a' ``. To do this, define `INCLUDE_RDB` in the VxWorks configuration file `` `configAll.h' `` and rebuild your VxWorks kernel. The resulting kernel contains `` `rdb.a' ``, and spawns the source debugging task `tRdbTask` when VxWorks is booted. For more information on configuring and remaking VxWorks, see the manufacturer's manual.

Once you have included `` `rdb.a' `` in your VxWorks system image and set your Unix execution search path to find GDB, you are ready to run GDB. From your Unix host, run `gdb` (or `vxgdb`, depending on your installation).

GDB comes up showing the prompt:

```
(vxgdb)
```

Connecting to VxWorks

The GDB command `target` lets you connect to a VxWorks target on the network. To connect to a target whose host name is "`tt`", type:

```
(vxgdb) target vxworks tt
```

GDB displays messages like these:

```
Attaching remote machine across net...
Connected to tt.
```

GDB then attempts to read the symbol tables of any object modules loaded into the VxWorks target since it was last booted. GDB locates these files by searching the directories listed in the command search path (see section Your program's environment); if it fails to find an object file, it displays a message such as:

```
prog.o: No such file or directory.
```

When this happens, add the appropriate directory to the search path with the GDB command `path`, and execute the `target` command again.

VxWorks download

If you have connected to the VxWorks target and you want to debug an object that has not yet been loaded, you can use the GDB `load` command to download a file from Unix to VxWorks incrementally. The object file given as an argument to the `load` command is actually opened twice: first by the VxWorks target in order to download the code, then by GDB in order to read the symbol table. This can lead to problems if the current working directories on the two systems differ. If both systems have NFS mounted the same filesystems, you can avoid these problems by using absolute paths. Otherwise, it is simplest to set the working directory on both systems to the directory in which the object file resides, and then to reference the file by its name, without any path. For instance, a program `` `prog.o' `` may reside in `` `vxpath/vw/demo/rdb' `` in VxWorks and in `` `hostpath/vw/demo/rdb' `` on the host. To load this program, type this on VxWorks:

```
-> cd "vxpath/vw/demo/rdb"
```

v Then, in GDB, type:

```
(vxgdb) cd hostpath/vw/demo/rdb
(vxgdb) load prog.o
```

GDB displays a response similar to this:

```
Reading symbol data from wherever/vw/demo/rdb/prog.o... done.
```

You can also use the `load` command to reload an object module after editing and recompiling the corresponding source file. Note that this makes GDB delete all currently-defined breakpoints, auto-displays, and convenience variables, and to clear the value history. (This is necessary in order to

preserve the integrity of debugger data structures that reference the target system's symbol table.)

Running tasks

You can also attach to an existing task using the `attach` command as follows:

```
(vxgdb) attach task
```

where task is the VxWorks hexadecimal task ID. The task can be running or suspended when you attach to it. Running tasks are suspended at the time of attachment.

GDB and Sparclet

GDB enables developers to debug tasks running on Sparclet targets from a Unix host. GDB uses code that runs on both the Unix host and on the Sparclet target. The program `gdb` is installed and executed on the Unix host.

`timeout args`

> GDB now supports the option `remotetimeout`. This option is set by the user, and args represents the number of seconds GDB waits for responses.

When compiling for debugging, include the options "-g" to get debug information and "-Ttext" to relocate the program to where you wish to load it on the target. You may also want to add the options "-n" or "-N" in order to reduce the size of the sections.

```
sparclet-aout-gcc prog.c -Ttext 0x12010000 -g -o prog -N
```

You can use objdump to verify that the addresses are what you intended.

```
sparclet-aout-objdump --headers --syms prog
```

Once you have set your Unix execution search path to find GDB, you are ready to run GDB. From your Unix host, run `gdb` (or `sparclet-aout-gdb`, depending on your installation).

GDB comes up showing the prompt:

```
(gdbslet)
```

Setting file to debug

The GDB command `file` lets you choose with program to debug.

```
(gdbslet) file prog
```

GDB then attempts to read the symbol table of `` `prog' ``. GDB locates the file by searching the directories listed in the command search path. If the file was compiled with debug information (option "-g"), source files will be searched as well. GDB locates the source files by searching the directories

listed in the directory search path (see section Your program's environment). If it fails to find a file, i displays a message such as:

```
prog: No such file or directory.
```

When this happens, add the appropriate directories to the search paths with the GDB commands `path` and `dir`, and execute the `target` command again.

Connecting to Sparclet

The GDB command `target` lets you connect to a Sparclet target. To connect to a target on serial port "`ttya`", type:

```
(gdbslet) target sparclet /dev/ttya
Remote target sparclet connected to /dev/ttya
main () at ../prog.c:3
```

GDB displays messages like these:

```
Connected to ttya.
```

Sparclet download

Once connected to the Sparclet target, you can use the GDB `load` command to download the file from the host to the target. The file name and load offset should be given as arguments to the `load` command. Since the file format is aout, the program must be loaded to the starting address. You can use objdump to find out what this value is. The load offset is an offset which is added to the VMA (virtual memory address) of each of the file's sections. For instance, if the program `` `prog' `` was linked to text address 0x1201000, with data at 0x12010160 and bss at 0x12010170, in GDB, type:

```
(gdbslet) load prog 0x12010000
Loading section .text, size 0xdb0 vma 0x12010000
```

If the code is loaded at a different address then what the program was linked to, you may need to use the `section` and `add-symbol-file` commands to tell GDB where to map the symbol table.

Running and debugging

You can now begin debugging the task using GDB's execution control commands, `b`, `step`, `run`, etc. See the GDB manual for the list of commands.

```
(gdbslet) b main
Breakpoint 1 at 0x12010000: file prog.c, line 3.
(gdbslet) run
Starting program: prog
Breakpoint 1, main (argc=1, argv=0xeffff21c) at prog.c:3
```


```
3         char *symarg = 0;
(gdbslet) step
4         char *execarg = "hello!";
(gdbslet)
```

GDB and Hitachi microprocessors

GDB needs to know these things to talk to your Hitachi SH, H8/300, or H8/500:

1. that you want to use `target hms', the remote debugging interface for Hitachi microprocessors, or `target e7000', the in-circuit emulator for the Hitachi SH and the Hitachi 300H. (`target hms' is the default when GDB is configured specifically for the Hitachi SH, H8/300, or H8/500.)
2. what serial device connects your host to your Hitachi board (the first serial device available on your host is the default).
3. what speed to use over the serial device.

Connecting to Hitachi boards

Use the special `gdb` command `device port' if you need to explicitly set the serial device. The default port is the first available port on your host. This is only necessary on Unix hosts, where it is typically something like `` `/dev/ttya' ``.

`gdb` has another special command to set the communications speed: `speed bps'. This command also is only used from Unix hosts; on DOS hosts, set the line speed as usual from outside GDB with the DOS mode command (for instance, `mode com2:9600,n,8,1,p' for a 9600 bps connection).

The `device' and `speed' commands are available only when you use a Unix host to debug your Hitachi microprocessor programs. If you use a DOS host, GDB depends on an auxiliary terminate-and-stay-resident program called `asynctsr` to communicate with the development board through a PC serial port. You must also use the DOS `mode` command to set up the serial port on the DOS side.

Using the E7000 in-circuit emulator

You can use the E7000 in-circuit emulator to develop code for either the Hitachi SH or the H8/300H. Use one of these forms of the `target e7000' command to connect GDB to your E7000:

`target e7000 port speed`

Use this form if your E7000 is connected to a serial port. The port argument identifies what serial port to use (for example, `com2'). The third argument is the line speed in bits per second (for example, `9600').

`target e7000 hostname`

If your E7000 is installed as a host on a TCP/IP network, you can just specify its hostname; GDB uses `telnet` to connect.

Special GDB commands for Hitachi micros

Some GDB commands are available only on the H8/300 or the H8/500 configurations:

`set machine h8300`

`set machine h8300h`

Condition GDB for one of the two variants of the H8/300 architecture with `set machine'. You can use `show machine' to check which variant is currently in effect.

`set memory mod`

`show memory`

Specify which H8/500 memory model (mod) you are using with `set memory'; check which memory model is in effect with `show memory'. The accepted values for mod are `small`, `big`, `medium`, and `compact`.

GDB and remote MIPS boards

GDB can use the MIPS remote debugging protocol to talk to a MIPS board attached to a serial line. This is available when you configure GDB with `--target=mips-idt-ecoff'.

Use these GDB commands to specify the connection to your target board:

`target mips port`

To run a program on the board, start up `gdb` with the name of your program as the argument. To connect to the board, use the command `target mips port', where port is the name of the serial port connected to the board. If the program has not already been downloaded to the board, you may use the `load` command to download it. You can then use all the usual GDB commands. For example, this sequence connects to the target board through a serial port, and loads and runs a program called prog through the debugger:

```
host$ gdb prog
GDB is free software and ...
(gdb) target mips /dev/ttyb
(gdb) load prog
(gdb) run
```

`target mips hostname:portnumber`

On some GDB host configurations, you can specify a TCP connection (for instance, to a serial line managed by a terminal concentrator) instead of a serial port, using the syntax `hostname:portnumber'.

`target pmon port`

`target ddb port`

`target lsi port`

GDB also supports these special commands for MIPS targets:

`set processor args`

`show processor`

Use the `set processor` command to set the type of MIPS processor when you want to access

processor-type-specific registers. For example, `set processor r3041` tells GDB to use t CPO registers appropriate for the 3041 chip. Use the `show processor` command to see wh MIPS processor GDB is using. Use the `info reg` command to see what registers GDB is using.

`set mipsfpu double`

`set mipsfpu single`

`set mipsfpu none`

`show mipsfpu`

If your target board does not support the MIPS floating point coprocessor, you should use the command `set mipsfpu none' (if you need this, you may wish to put the command in your file). This tells GDB how to find the return value of functions which return floating point values. It also allows GDB to avoid saving the floating point registers when calling functions on the board. If you are using a floating point coprocessor with only single precision floating point support, as on the R4650 processor, use the command `set mipsfpu single'. The default double precision floating point coprocessor may be selected using `set mipsfpu double'. In previous versions the only choices were double precision or no floating point, so `set mipsfpu on' will select double precision and `set mipsfpu off' will select no floating point. As usual, you can inquire about the `mipsfpu` variable with `show mipsfpu'.

`set remotedebug n`

`show remotedebug`

You can see some debugging information about communications with the board by setting the `remotedebug` variable. If you set it to `1` using `set remotedebug 1', every packet is displayed. If you set it to `2`, every character is displayed. You can check the current value at any time with the command `show remotedebug'.

`set timeout seconds`

`set retransmit-timeout seconds`

`show timeout`

`show retransmit-timeout`

You can control the timeout used while waiting for a packet, in the MIPS remote protocol, with the `set timeout seconds` command. The default is 5 seconds. Similarly, you can control the timeout used while waiting for an acknowledgement of a packet with the `set retransmit-timeout seconds` command. The default is 3 seconds. You can inspect both values with `show timeout` and `show retransmit-timeout`. (These commands are *only* available when GDB is configured for `--target=mips-idt-ecoff'.) The timeout set by `set timeout` does not apply when GDB is waiting for your program to stop. In that case, GDB waits forever because it has no way of knowing how long the program is going to run before stopping.

Simulated CPU target

For some configurations, GDB includes a CPU simulator that you can use instead of a hardware CPU to debug your programs. Currently, simulators are available for ARM, D10V, D30V, FR30, H8/300, H8/500, i960, M32R, MIPS, MN10200, MN10300, PowerPC, SH, Sparc, V850, W65, and Z8000.

For the Z8000 family, `target sim' simulates either the Z8002 (the unsegmented variant of the Z8000 architecture) or the Z8001 (the segmented variant). The simulator recognizes which architecture is appropriate by inspecting the object code.

`target sim args`

Debug programs on a simulated CPU. If the simulator supports setup options, specify them via args.

After specifying this target, you can debug programs for the simulated CPU in the same style as programs for your host computer; use the `file` command to load a new program image, the `run` command to run your program, and so on.

As well as making available all the usual machine registers (see `info reg`), the Z8000 simulator provides three additional items of information as specially named registers:

`cycles`

Counts clock-ticks in the simulator.

`insts`

Counts instructions run in the simulator.

`time`

Execution time in 60ths of a second.

You can refer to these values in GDB expressions with the usual conventions; for example, `b fputc if $cycles>5000' sets a conditional breakpoint that suspends only after at least 5000 simulated clock ticks.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Controlling GDB

You can alter the way GDB interacts with you by using the `set` command. For commands controlling how GDB displays data, see section Print settings; other settings are described here.

Prompt

GDB indicates its readiness to read a command by printing a string called the **prompt**. This string is normally `(gdb)'. You can change the prompt string with the `set prompt` command. For instance, when debugging GDB with GDB, it is useful to change the prompt in one of the GDB sessions so that you can always tell which one you are talking to.

Note: `set prompt` no longer adds a space for you after the prompt you set. This allows you to set a prompt which ends in a space or a prompt that does not.

`set prompt newprompt`

Directs GDB to use newprompt as its prompt string henceforth.

`show prompt`

Prints a line of the form: `Gdb's prompt is: your-prompt'

Command editing

GDB reads its input commands via the **readline** interface. This GNU library provides consistent behavior for programs which provide a command line interface to the user. Advantages are GNU Emacs-style or **vi**-style inline editing of commands, `csh`-like history substitution, and a storage and recall of command history across debugging sessions.

You may control the behavior of command line editing in GDB with the command `set`.

`set editing`

`set editing on`

Enable command line editing (enabled by default).

`set editing off`

Disable command line editing.

`show editing`

Show whether command line editing is enabled.

TOC

Command history

GDB can keep track of the commands you type during your debugging sessions, so that you can be certain of precisely what happened. Use these commands to manage the GDB command history facility.

`set history filename fname`

Set the name of the GDB command history file to fname. This is the file where GDB reads an initial command history list, and where it writes the command history from this session when it exits. You can access this list through history expansion or through the history command editing characters listed below. This file defaults to the value of the environment variable `GDBHISTFILE`, or to `` `./.gdb_history' `` if this variable is not set.

`set history save`

`set history save on`

Record command history in a file, whose name may be specified with the `set history filename` command. By default, this option is disabled.

`set history save off`

Stop recording command history in a file.

`set history size size`

Set the number of commands which GDB keeps in its history list. This defaults to the value of the environment variable `HISTSIZE`, or to 256 if this variable is not set.

History expansion assigns special meaning to the character !.

Since ! is also the logical not operator in C, history expansion is off by default. If you decide to enable history expansion with the `set history expansion on` command, you may sometimes need to follow ! (when it is used as logical not, in an expression) with a space or a tab to prevent it from being expanded. The readline history facilities do not attempt substitution on the strings != and !(, even when history expansion is enabled.

The commands to control history expansion are:

`set history expansion on`

`set history expansion`

Enable history expansion. History expansion is off by default.

`set history expansion off`

Disable history expansion. The readline code comes with more complete documentation of editing and history expansion features. Users unfamiliar with GNU Emacs or `vi` may wish to read it.

`show history`

`show history filename`

`show history save`

`show history size`

`show history expansion`

These commands display the state of the GDB history parameters. `show history` by itself

displays all four states.

`show commands`

Display the last ten commands in the command history.

`show commands n`

Print ten commands centered on command number n.

`show commands +`

Print ten commands just after the commands last printed.

Screen size

Certain commands to GDB may produce large amounts of information output to the screen. To help you read all of it, GDB pauses and asks you for input at the end of each page of output. Type RET when you want to continue the output, or q to discard the remaining output. Also, the screen width setting determines when to wrap lines of output. Depending on what is being printed, GDB tries to break the line at a readable place, rather than simply letting it overflow onto the following line.

Normally GDB knows the size of the screen from the termcap data base together with the value of the `TERM` environment variable and the `stty rows` and `stty cols` settings. If this is not correct, you can override it with the `set height` and `set width` commands:

`set height lpp`

`show height`

`set width cpl`

`show width`

These `set` commands specify a screen height of lpp lines and a screen width of cpl characters. The associated `show` commands display the current settings. If you specify a height of zero lines, GDB does not pause during output no matter how long the output is. This is useful if output is to a file or to an editor buffer. Likewise, you can specify `set width 0' to prevent GDB from wrapping its output.

Numbers

You can always enter numbers in octal, decimal, or hexadecimal in GDB by the usual conventions: octal numbers begin with `0', decimal numbers end with `.', and hexadecimal numbers begin with `0x'. Numbers that begin with none of these are, by default, entered in base 10; likewise, the default display for numbers--when no particular format is specified--is base 10. You can change the default base for both input and output with the `set radix` command.

`set input-radix base`

Set the default base for numeric input. Supported choices for base are decimal 8, 10, or 16. base must itself be specified either unambiguously or using the current default radix; for example, any of


```
set radix 012
set radix 10.
set radix 0xa
```

sets the base to decimal. On the other hand, `set radix 10' leaves the radix unchanged no matter what it was.

`set output-radix base`

Set the default base for numeric display. Supported choices for base are decimal 8, 10, or 16. base must itself be specified either unambiguously or using the current default radix.

`show input-radix`

Display the current default base for numeric input.

`show output-radix`

Display the current default base for numeric display.

Optional warnings and messages

By default, GDB is silent about its inner workings. If you are running on a slow machine, you may want to use the `set verbose` command. This makes GDB tell you when it does a lengthy internal operation, so you will not think it has crashed.

Currently, the messages controlled by `set verbose` are those which announce that the symbol table for a source file is being read; see `symbol-file` in section Commands to specify files.

`set verbose on`

Enables GDB output of certain informational messages.

`set verbose off`

Disables GDB output of certain informational messages.

`show verbose`

Displays whether `set verbose` is on or off.

By default, if GDB encounters bugs in the symbol table of an object file, it is silent; but if you are debugging a compiler, you may find this information useful (see section Errors reading symbol files).

`set complaints limit`

Permits GDB to output limit complaints about each type of unusual symbols before becoming silent about the problem. Set limit to zero to suppress all complaints; set it to a large number to prevent complaints from being suppressed.

`show complaints`

Displays how many symbol complaints GDB is permitted to produce.

By default, GDB is cautious, and asks what sometimes seems to be a lot of stupid questions to confirm certain commands. For example, if you try to run a program which is already running:

```
(gdb) run
```


```
The program being debugged has been started already.
Start it from the beginning? (y or n)
```

If you are willing to unflinchingly face the consequences of your own commands, you can disable this "feature":

`set confirm off`

Disables confirmation requests.

`set confirm on`

Enables confirmation requests (the default).

`show confirm`

Displays state of confirmation requests.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Canned Sequences of Commands

Aside from breakpoint commands (see section Breakpoint command lists), GDB provides two ways to store sequences of commands for execution as a unit: user-defined commands and command files.

User-defined commands

A **user-defined command** is a sequence of GDB commands to which you assign a new name as a command. This is done with the `define` command. User commands may accept up to 10 arguments separated by whitespace. Arguments are accessed within the user command via $arg0...$arg9. A trivial example:

```
define adder
  print $arg0 + $arg1 + $arg2
```

To execute the command use:

```
adder 1 2 3
```

This defines the command `adder`, which prints the sum of its three arguments. Note the arguments are text substitutions, so they may reference variables, use complex expressions, or even perform inferior functions calls.

`define commandname`

Define a command named commandname. If there is already a command by that name, you are asked to confirm that you want to redefine it. The definition of the command is made up of other GDB command lines, which are given following the `define` command. The end of these commands is marked by a line containing `end`.

`if`

Takes a single argument, which is an expression to evaluate. It is followed by a series of commands that are executed only if the expression is true (nonzero). There can then optionally be a line `else`, followed by a series of commands that are only executed if the expression was false. The end of the list is marked by a line containing `end`.

`while`

The syntax is similar to `if`: the command takes a single argument, which is an expression to evaluate, and must be followed by the commands to execute, one per line, terminated by an `end`. The commands are executed repeatedly as long as the expression evaluates to true.

`document commandname`

Document the user-defined command commandname, so that it can be accessed by `help`. The

command commandname must already be defined. This command reads lines of documentatic just as `define` reads the lines of the command definition, ending with `end`. After the `docur` command is finished, `help` on command commandname displays the documentation you have written. You may use the `document` command again to change the documentation of a command. Redefining the command with `define` does not change the documentation.

`help user-defined`

List all user-defined commands, with the first line of the documentation (if any) for each.

`show user`

`show user commandname`

Display the GDB commands used to define commandname (but not its documentation). If no commandname is given, display the definitions for all user-defined commands.

When user-defined commands are executed, the commands of the definition are not printed. An error in any command stops execution of the user-defined command.

If used interactively, commands that would ask for confirmation proceed without asking when used inside a user-defined command. Many GDB commands that normally print messages to say what they are doing omit the messages when used in a user-defined command.

User-defined command hooks

You may define *hooks*, which are a special kind of user-defined command. Whenever you run the command `foo', if the user-defined command `hook-foo' exists, it is executed (with no arguments) before that command.

In addition, a pseudo-command, `stop' exists. Defining (`hook-stop') makes the associated commands execute every time execution stops in your program: before breakpoint commands are run, displays are printed, or the stack frame is printed.

For example, to ignore `SIGALRM` signals while single-stepping, but treat them normally during normal execution, you could define:

```
define hook-stop
handle SIGALRM nopass
end

define hook-run
handle SIGALRM pass
end

define hook-continue
handle SIGLARM pass
end
```

You can define a hook for any single-word command in GDB, but not for command aliases; you should

define a hook for the basic command name, e.g. `backtrace` rather than `bt`. If an error occurs dur the execution of your hook, execution of GDB commands stops and GDB issues a prompt (before th command that you actually typed had a chance to run).

If you try to define a hook which does not match any known command, you get a warning from the `define` command.

Command files

A command file for GDB is a file of lines that are GDB commands. Comments (lines starting with #) may also be included. An empty line in a command file does nothing; it does not mean to repeat the last command, as it would from the terminal.

When you start GDB, it automatically executes commands from its **init files**. These are files named `` `.gdbinit' `` on Unix, or `` `gdb.ini' `` on DOS/Windows. GDB reads the init file (if any) in your home directory, then processes command line options and operands, and then reads the init file (if any) in the current working directory. This is so the init file in your home directory can set options (such as `set complaints`) which affect the processing of the command line options and operands. The init files are not executed if you use the `-nx' option; see section Choosing modes.

On some configurations of GDB, the init file is known by a different name (these are typically environments where a specialized form of GDB may need to coexist with other forms, hence a different name for the specialized version's init file). These are the environments with special init file names:

- VxWorks (Wind River Systems real-time OS): `.vxgdbinit'
- OS68K (Enea Data Systems real-time OS): `.os68gdbinit'
- ES-1800 (Ericsson Telecom AB M68000 emulator): `.esgdbinit'

You can also request the execution of a command file with the `source` command:

`source filename`

Execute the command file filename.

The lines in a command file are executed sequentially. They are not printed as they are executed. An error in any command terminates execution of the command file.

Commands that would ask for confirmation if used interactively proceed without asking when used in a command file. Many GDB commands that normally print messages to say what they are doing omit the messages when called from command files.

Commands for controlled output

During the execution of a command file or a user-defined command, normal GDB output is suppressed; the only output that appears is what is explicitly printed by the commands in the definition. This section describes three commands useful for generating exactly the output you want.

`echo text`

Print text. Nonprinting characters can be included in text using C escape sequences, such as `\n' to print a newline. **No newline is printed unless you specify one.** In addition to the standard C escape sequences, a backslash followed by a space stands for a space. This is useful for displaying a string with spaces at the beginning or the end, since leading and trailing spaces are otherwise trimmed from all arguments. To print ` and foo = ', use the command `echo \ and foo = \ '. A backslash at the end of text can be used, as in C, to continue the command onto subsequent lines. For example,

```
echo This is some text\n\
which is continued\n\
onto several lines.\n
```

produces the same output as

```
echo This is some text\n
echo which is continued\n
echo onto several lines.\n
```

`output expression`

Print the value of expression and nothing but that value: no newlines, no `$nn = '. The value is not entered in the value history either. See section Expressions, for more information on expressions.

`output/fmt expression`

Print the value of expression in format fmt. You can use the same formats as for `print`. See section Output formats, for more information.

`printf string, expressions...`

Print the values of the expressions under the control of string. The expressions are separated by commas and may be either numbers or pointers. Their values are printed as specified by string, exactly as if your program were to execute the C subroutine

```
printf (string, expressions...);
```

For example, you can print two values in hex like this:

```
printf "foo, bar-foo = 0x%x, 0x%x\n", foo, bar-foo
```

The only backslash-escape sequences that you can use in the format string are the simple ones that consist of backslash followed by a letter.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Using GDB under GNU Emacs

A special interface allows you to use GNU Emacs to view (and edit) the source files for the program you are debugging with GDB.

To use this interface, use the command M-x gdb in Emacs. Give the executable file you want to debug as an argument. This command starts GDB as a subprocess of Emacs, with input and output through a newly created Emacs buffer.

Using GDB under Emacs is just like using GDB normally except for two things:

- All "terminal" input and output goes through the Emacs buffer.

This applies both to GDB commands and their output, and to the input and output done by the program you are debugging.

This is useful because it means that you can copy the text of previous commands and input them again; you can even use parts of the output in this way.

All the facilities of Emacs' Shell mode are available for interacting with your program. In particular, you can send signals the usual way--for example, C-c C-c for an interrupt, C-c C-z for a stop.

- GDB displays source code through Emacs.

Each time GDB displays a stack frame, Emacs automatically finds the source file for that frame and puts an arrow (`=>') at the left margin of the current line. Emacs uses a separate buffer for source display, and splits the screen to show both your GDB session and the source.

Explicit GDB `list` or search commands still produce output as usual, but you probably have no reason to use them from Emacs.

> *Warning:* If the directory where your program resides is not your current directory, it can be easy to confuse Emacs about the location of the source files, in which case the auxiliary display buffer does not appear to show your source. GDB can find programs by searching your environment's `PATH` variable, so the GDB input and output session proceeds normally; but Emacs does not get enough information back from GDB to locate the source files in this situation. To avoid this problem, either start GDB mode from the directory where your program resides, or specify an absolute file name when prompted for the M-x gdb argument.
>
> A similar confusion can result if you use the GDB `file` command to switch to debugging a program in some other location, from an existing GDB buffer in Emacs.

By default, M-x gdb calls the program called `gdb'. If you need to call GDB by a different name (for example, if you keep several configurations around, with different names) you can set the Emacs variable `gdb-command-name`; for example,


```
(setq gdb-command-name "mygdb")
```

(preceded by ESC ESC, or typed in the `*scratch*` buffer, or in your `.emacs' file) makes Emacs call the program named "`mygdb`" instead.

In the GDB I/O buffer, you can use these special Emacs commands in addition to the standard Shell mode commands:

C-h m

Describe the features of Emacs' GDB Mode.

M-s

Execute to another source line, like the GDB `step` command; also update the display window to show the current file and location.

M-n

Execute to next source line in this function, skipping all function calls, like the GDB `next` command. Then update the display window to show the current file and location.

M-i

Execute one instruction, like the GDB `stepi` command; update display window accordingly.

M-x gdb-nexti

Execute to next instruction, using the GDB `nexti` command; update display window accordingly.

C-c C-f

Execute until exit from the selected stack frame, like the GDB `finish` command.

M-c

Continue execution of your program, like the GDB `continue` command. *Warning:* In Emacs v19, this command is C-c C-p.

M-u

Go up the number of frames indicated by the numeric argument (see section `Numeric Arguments' in The GNU Emacs Manual), like the GDB `up` command. *Warning:* In Emacs v19, this command is C-c C-u.

M-d

Go down the number of frames indicated by the numeric argument, like the GDB `down` command. *Warning:* In Emacs v19, this command is C-c C-d.

C-x &

Read the number where the cursor is positioned, and insert it at the end of the GDB I/O buffer. For example, if you wish to disassemble code around an address that was displayed earlier, type disassemble; then move the cursor to the address display, and pick up the argument for `disassemble` by typing C-x &. You can customize this further by defining elements of the list `gdb-print-command`; once it is defined, you can format or otherwise process numbers picked up by C-x & before they are inserted. A numeric argument to C-x & indicates that you wish special formatting, and also acts as an index to pick an element of the list. If the list element is a string, the number to be inserted is formatted using the Emacs function `format`; otherwise the

number is passed as an argument to the corresponding list element.

In any source file, the Emacs command C-x SPC (`gdb-break`) tells GDB to set a breakpoint on the source line point is on.

If you accidentally delete the source-display buffer, an easy way to get it back is to type the command `f` in the GDB buffer, to request a frame display; when you run under Emacs, this recreates the source buffer if necessary to show you the context of the current frame.

The source files displayed in Emacs are in ordinary Emacs buffers which are visiting the source files in the usual way. You can edit the files with these buffers if you wish; but keep in mind that GDB communicates with Emacs in terms of line numbers. If you add or delete lines from the text, the line numbers that GDB knows cease to correspond properly with the code.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Reporting Bugs in GDB

Your bug reports play an essential role in making GDB reliable.

Reporting a bug may help you by bringing a solution to your problem, or it may not. But in any case the principal function of a bug report is to help the entire community by making the next version of GDB work better. Bug reports are your contribution to the maintenance of GDB.

In order for a bug report to serve its purpose, you must include the information that enables us to fix the bug.

Have you found a bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the debugger gets a fatal signal, for any input whatever, that is a GDB bug. Reliable debuggers never crash.
- If GDB produces an error message for valid input, that is a bug. (Note that if you're cross debugging, the problem may also be somewhere in the connection to the target.)
- If GDB does not produce an error message for invalid input, that is a bug. However, you should note that your idea of "invalid input" might be our idea of "an extension" or "support for traditional practice".
- If you are an experienced user of debugging tools, your suggestions for improvement of GDB are welcome in any case.

How to report bugs

A number of companies and individuals offer support for GNU products. If you obtained GDB from a support organization, we recommend you contact that organization first.

You can find contact information for many support companies and individuals in the file `etc/SERVICE' in the GNU Emacs distribution.

In any event, we also recommend that you send bug reports for GDB to this addresses:

bug-gdb@prep.ai.mit.edu

Do not send bug reports to `info-gdb', or to `help-gdb', or to any newsgroups. Most users of GDB do not want to receive bug reports. Those that do have arranged to receive `bug-gdb'.

The mailing list `bug-gdb' has a newsgroup `gnu.gdb.bug' which serves as a repeater. The mailing list

and the newsgroup carry exactly the same messages. Often people think of posting bug reports to the newsgroup instead of mailing them. This appears to work, but it has one problem which can be crucial: newsgroup posting often lacks a mail path back to the sender. Thus, if we need to ask for more information, we may be unable to reach you. For this reason, it is better to send bug reports to the mailing list.

As a last resort, send bug reports on paper to:

```
GNU Debugger Bugs
Free Software Foundation Inc.
59 Temple Place - Suite 330
Boston, MA 02111-1307
USA
```

The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to state a fact or leave it out, state it!

Often people omit facts because they think they know what causes the problem and assume that some details do not matter. Thus, you might assume that the name of the variable you use in an example does not matter. Well, probably it does not, but one cannot be sure. Perhaps the bug is a stray memory reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name were different, the contents of that location would fool the debugger into doing the right thing despite the bug. Play it safe and give a specific, complete example. That is the easiest thing for you to do, and the most helpful.

Keep in mind that the purpose of a bug report is to enable us to fix the bug. It may be that the bug has been reported previously, but neither you nor we can know that unless your bug report is complete and self-contained.

Sometimes people give a few sketchy facts and ask, "Does this ring a bell?" Those bug reports are useless, and we urge everyone to *refuse to respond to them* except to chide the sender to report bugs properly.

To enable us to fix the bug, you should include all these things:

- The version of GDB. GDB announces it if you start with no arguments; you can also print it at any time using `show version`. Without this, we will not know whether there is any point in looking for the bug in the current version of GDB.
- The type of machine you are using, and the operating system name and version number.
- What compiler (and its version) was used to compile GDB---e.g. "gcc--2.8.1".
- What compiler (and its version) was used to compile the program you are debugging--e.g. "gcc--2.8.1", or "HP92453-01 A.10.32.03 HP C Compiler". For GCC, you can say `gcc --version` to get this information; for other compilers, see the documentation for those compilers.
- The command arguments you gave the compiler to compile your example and observe the bug. For example, did you use `-O'? To guarantee you will not omit something important, list them all. A copy of the Makefile (or the output from make) is sufficient. If we were to try to guess the

arguments, we would probably guess wrong and then we might not encounter the bug.

- A complete input script, and all necessary source files, that will reproduce the bug.
- A description of what behavior you observe that you believe is incorrect. For example, "It gets a fatal signal." Of course, if the bug is that GDB gets a fatal signal, then we will certainly notice it. But if the bug is incorrect output, we might not notice unless it is glaringly wrong. You might as well not give us a chance to make a mistake. Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of GDB is out of synch, or you have encountered a bug in the C library on your system. (This has happened!) Your copy might crash and ours would not. If you told us to expect a crash, then when ours fails to crash, we would know that the bug was not happening for us. If you had not told us to expect a crash, then we would not be able to draw any conclusion from our observations.
- If you wish to suggest changes to the GDB source, send us context diffs. If you even discuss something in the GDB source, refer to it by context, not by line number. The line numbers in our development sources will not match those in your sources. Your line numbers would convey no useful information to us.

Here are some things that are not necessary:

- A description of the envelope of the bug. Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it. This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. We recommend that you save your time for something else. Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience for us. Errors in the output will be easier to spot, running under the debugger will take less time, and so on. However, simplification is not vital; if you do not want to do this, report the bug anyway and send us the entire test case you used.
- A patch for the bug. A patch for the bug does help us if it is a good one. But do not omit the necessary information, such as the test case, on the assumption that a patch is all we need. We might see problems with your patch and decide to fix the problem another way, or we might not understand it at all. Sometimes with a program as complicated as GDB it is very hard to construct an example that will make the program follow a certain path through the code. If you do not send us the example, we will not be able to construct one, so we will not be able to verify that the bug is fixed. And if we cannot understand what bug you are trying to fix, or why your patch should be an improvement, we will not install it. A test case will help us to understand.
- A guess about what the bug is or what it depends on. Such guesses are usually wrong. Even we cannot guess right about such things without first using the debugger to find the facts.

Go to the first, previous, next, last section, table of contents.

Command Line Editing

This chapter describes the basic features of the GNU command line editing interface.

Introduction to Line Editing

The following paragraphs describe the notation used to represent keystrokes.

The text C-k is read as `Control-K' and describes the character produced when the k key is pressed while the Control key is depressed.

The text M-k is read as `Meta-K' and describes the character produced when the meta key (if you have one) is depressed, and the k key is pressed. If you do not have a meta key, the identical keystroke can be generated by typing ESC *first*, and then typing k. Either process is known as **metafying** the k key.

The text M-C-k is read as `Meta-Control-k' and describes the character produced by **metafying** C-k.

In addition, several keys have their own names. Specifically, DEL, ESC, LFD, SPC, RET, and TAB all stand for themselves when seen in this text, or in an init file (@xref{Readline Init File}).

Readline Interaction

Often during an interactive session you type in a long line of text, only to notice that the first word on the line is misspelled. The Readline library gives you a set of commands for manipulating the text as you type it in, allowing you to just fix your typo, and not forcing you to retype the majority of the line. Using these editing commands, you move the cursor to the place that needs correction, and delete or insert the text of the corrections. Then, when you are satisfied with the line, you simply press RETURN. You do not have to be at the end of the line to press RETURN; the entire line is accepted regardless of the location of the cursor within the line.

Readline Init File Syntax

There are only a few basic constructs allowed in the Readline init file. Blank lines are ignored. Lines beginning with a `#' are comments. Lines beginning with a `$' indicate conditional constructs (see section Conditional Init Constructs). Other lines denote variable settings and key bindings.

Variable Settings

You can modify the run-time behavior of Readline by altering the values of variables in Readline using the `set` command within the init file. Here is how to change from the default Emacs-like key binding to use `vi` line editing commands:

```
set editing-mode vi
```

A great deal of run-time behavior is changeable with the following variables.

`bell-style`

Controls what happens when Readline wants to ring the terminal bell. If set to `none', Readline never rings the bell. If set to `visible', Readline uses a visible bell if one is available. If set to `audible' (the default), Readline attempts to ring the terminal's bell.

`comment-begin`

The string to insert at the beginning of the line when the `insert-comment` command is executed. The default value is `"#"`.

`completion-ignore-case`

If set to `on', Readline performs filename matching and completion in a case-insensitive fashion. The default value is `off'.

`completion-query-items`

The number of possible completions that determines when the user is asked whether he wants to see the list of possibilities. If the number of possible completions is greater than this value, Readline will ask the user whether or not he wishes to view them; otherwise, they are simply listed. The default limit is `100`.

`convert-meta`

If set to `on', Readline will convert characters with the eighth bit set to an ASCII key sequence by stripping the eighth bit and prepending an ESC character, converting them to a meta-prefixed key sequence. The default value is `on'.

`disable-completion`

If set to `On', Readline will inhibit word completion. Completion characters will be inserted into the line as if they had been mapped to `self-insert`. The default is `off'.

`editing-mode`

The `editing-mode` variable controls which default set of key bindings is used. By default, Readline starts up in Emacs editing mode, where the keystrokes are most similar to Emacs. This variable can be set to either `emacs' or `vi'.

`enable-keypad`

When set to `on', Readline will try to enable the application keypad when it is called. Some systems need this to enable the arrow keys. The default is `off'.

`expand-tilde`

If set to `on', tilde expansion is performed when Readline attempts word completion. The default is `off'.

`horizontal-scroll-mode`

This variable can be set to either `on' or `off'. Setting it to `on' means that the text of the lines being edited will scroll horizontally on a single screen line when they are longer than the width of the screen, instead of wrapping onto a new screen line. By default, this variable is set to `off'.

`keymap`

Sets Readline's idea of the current keymap for key binding commands. Acceptable `keymap` names are `emacs`, `emacs-standard`, `emacs-meta`, `emacs-ctlx`, `vi`, `vi-command`, and `vi-insert`. `vi` is equivalent to `vi-command`; `emacs` is equivalent to `emacs-standard`. The default value is `emacs`. The value of the `editing-mode` variable also affects the default keymap.

`mark-directories`

If set to `on', completed directory names have a slash appended. The default is `on'.

`mark-modified-lines`

This variable, when set to `on', causes Readline to display an asterisk (`*') at the start of history lines which have been modified. This variable is `off' by default.

`input-meta`

If set to `on', Readline will enable eight-bit input (it will not strip the eighth bit from the characters it reads), regardless of what the terminal claims it can support. The default value is `off'. The name `meta-flag` is a synonym for this variable.

`output-meta`

If set to `on', Readline will display characters with the eighth bit set directly rather than as a meta-prefixed escape sequence. The default is `off'.

`print-completions-horizontally`

If set to `on', Readline will display completions with matches sorted horizontally in alphabetical order, rather than down the screen. The default is `off'.

`show-all-if-ambiguous`

This alters the default behavior of the completion functions. If set to `on', words which have more than one possible completion cause the matches to be listed immediately instead of ringing the bell. The default value is `off'.

`visible-stats`

If set to `on', a character denoting a file's type is appended to the filename when listing possible completions. The default is `off'.

Key Bindings

The syntax for controlling key bindings in the init file is simple. First you have to know the name of the command that you want to change. The following sections contain tables of the command name, the default keybinding, if any, and a short description of what the command does. Once you know the name of the command, simply place the name of the key you wish to bind the command to, a colon, and then the name of the command on a line in the init file. The name of the key can be expressed in different ways, depending on which is most comfortable for you.

keyname: function-name or macro

keyname is the name of a key spelled out in English. For example:

```
Control-u: universal-argument
Meta-Rubout: backward-kill-word
Control-o: "> output"
```

In the above example, C-u is bound to the function `universal-argument`, and C-o is bound to run the macro expressed on the right hand side (that is, to insert the text `> output' into the line).

"keyseq": function-name or macro

keyseq differs from keyname above in that strings denoting an entire key sequence can be specified, by placing the key sequence in double quotes. Some GNU Emacs style key escapes can be used, as in the following example, but the special character names are not recognized.

```
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
"\e[11~": "Function Key 1"
```

In the above example, C-u is bound to the function `universal-argument` (just as it was in the first example), `C-x C-r' is bound to the function `re-read-init-file`, and `ESC [1 1 ~' is bound to insert the text `Function Key 1'.

The following GNU Emacs style escape sequences are available when specifying key sequences:

`\C-`

control prefix

`\M-`

meta prefix

`\e`

an escape character

`\\`
backslash

`\"`
"

`\'`
'

In addition to the GNU Emacs style escape sequences, a second set of backslash escapes is available:

`\a`
alert (bell)

`\b`
backspace

`\d`
delete

`\f`
form feed

`\n`
newline

`\r`
carriage return

`\t`
horizontal tab

`\v`
vertical tab

`\nnn`
the character whose ASCII code is the octal value nnn (one to three digits)

`\xnnn`
the character whose ASCII code is the hexadecimal value nnn (one to three digits)

When entering the text of a macro, single or double quotes must be used to indicate a macro definition. Unquoted text is assumed to be a function name. In the macro body, the backslash escapes described above are expanded. Backslash will quote any other character in the macro text, including `'" and `'". For example, the following binding will make `C-x \' insert a single `\' into the line:

```
"\C-x\\": "\\"
```

Conditional Init Constructs

Readline implements a facility similar in spirit to the conditional compilation features of the C preprocessor which allows key bindings and variable settings to be performed as the result of tests. There are four parser directives used.

`$if`

The `$if` construct allows bindings to be made based on the editing mode, the terminal being used, or the application using Readline. The text of the test extends to the end of the line; no characters are required to isolate it.

`mode`

The `mode=` form of the `$if` directive is used to test whether Readline is in `emacs` or `vi` mode. This may be used in conjunction with the `set keymap' command, for instance, to set bindings in the `emacs-standard` and `emacs-ctlx` keymaps only if Readline is starting out in `emacs` mode.

`term`

The `term=` form may be used to include terminal-specific key bindings, perhaps to bind the key sequences output by the terminal's function keys. The word on the right side of the `=' is tested against both the full name of the terminal and the portion of the terminal name before the first `-'. This allows `sun` to match both `sun` and `sun-cmd`, for instance.

`application`

The application construct is used to include application-specific settings. Each program using the Readline library sets the application name, and you can test for it. This could be used to bind key sequences to functions useful for a specific program. For instance, the following command adds a key sequence that quotes the current or previous word in Bash:

```
$if Bash
# Quote the current or previous word
"\C-xq": "\eb\"\ef\""
$endif
```

`$endif`

This command, as seen in the previous example, terminates an `$if` command.

`$else`

Commands in this branch of the `$if` directive are executed if the test fails.

`$include`

This directive takes a single filename as an argument and reads commands and bindings from that file.

```
$include /etc/inputrc
```

Sample Init File

Here is an example of an inputrc file. This illustrates key binding, variable assignment, and conditional syntax.

```
# This file controls the behaviour of line input editing for
# programs that use the Gnu Readline library.  Existing programs
# include FTP, Bash, and Gdb.
#
# You can re-read the inputrc file with C-x C-r.
# Lines beginning with '#' are comments.
#
# First, include any systemwide bindings and variable assignments from
# /etc/Inputrc
$include /etc/Inputrc

#
# Set various bindings for emacs mode.

set editing-mode emacs
```


```
$if mode=emacs

Meta-Control-h: backward-kill-word      Text after the function name is ignored

#
# Arrow keys in keypad mode
#
#"\M-OD":        backward-char
#"\M-OC":        forward-char
#"\M-OA":        previous-history
#"\M-OB":        next-history
#
# Arrow keys in ANSI mode
#
"\M-[D":        backward-char
"\M-[C":        forward-char
"\M-[A":        previous-history
"\M-[B":        next-history
#
# Arrow keys in 8 bit keypad mode
#
#"\M-\C-OD":       backward-char
#"\M-\C-OC":       forward-char
#"\M-\C-OA":       previous-history
#"\M-\C-OB":       next-history
#
# Arrow keys in 8 bit ANSI mode
#
#"\M-\C-[D":       backward-char
#"\M-\C-[C":       forward-char
#"\M-\C-[A":       previous-history
#"\M-\C-[B":       next-history

C-q: quoted-insert

$endif

# An old-style binding.  This happens to be the default.
TAB: complete

# Macros that are convenient for shell interaction
$if Bash
# edit the path
"\C-xp": "PATH=${PATH}\e\C-e\C-a\ef\C-f"
# prepare to type a quoted word -- insert open and close double quotes
# and move to just after the open quote
"\C-x\"": "\"\"\C-b"
# insert a backslash (testing backslash escapes in sequences and macros)
"\C-x\\": "\\"
# Quote the current or previous word
```


```
"\C-xq": "\eb\"\ef\""
# Add a binding to refresh the line, which is unbound
"\C-xr": redraw-current-line
# Edit variable on current line.
"\M-\C-v": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
$endif

# use a visible bell if one is available
set bell-style visible

# don't strip characters to 7 bits when reading
set input-meta on

# allow iso-latin1 characters to be inserted rather than converted to
# prefix-meta sequences
set convert-meta off

# display characters with the eighth bit set directly rather than
# as meta-prefixed characters
set output-meta on

# if there are more than 150 possible completions for a word, ask the
# user if he wants to see all of them
set completion-query-items 150

# For FTP
$if Ftp
"\C-xg": "get \M-?"
"\C-xt": "put \M-?"
"\M-.": yank-last-arg
$endif
```

Bindable Readline Commands

This section describes Readline commands that may be bound to key sequences.

Commands For Moving

`beginning-of-line (C-a)`
: Move to the start of the current line.

`end-of-line (C-e)`
: Move to the end of the line.

`forward-char (C-f)`
: Move forward a character.

`backward-char (C-b)`
: Move back a character.

`forward-word (M-f)`
: Move forward to the end of the next word. Words are composed of letters and digits.

`backward-word (M-b)`

Move back to the start of this, or the previous, word. Words are composed of letters and digits.

`clear-screen (C-l)`

Clear the screen and redraw the current line, leaving the current line at the top of the screen.

`redraw-current-line ()`

Refresh the current line. By default, this is unbound.

Commands For Manipulating The History

`accept-line (Newline, Return)`

Accept the line regardless of where the cursor is. If this line is non-empty, add it to the history list. If this line was a history line, then restore the history line to its original state.

`previous-history (C-p)`

Move `up' through the history list.

`next-history (C-n)`

Move `down' through the history list.

`beginning-of-history (M-<)`

Move to the first line in the history.

`end-of-history (M->)`

Move to the end of the input history, i.e., the line currently being entered.

`reverse-search-history (C-r)`

Search backward starting at the current line and moving `up' through the history as necessary. This is an incremental search.

`forward-search-history (C-s)`

Search forward starting at the current line and moving `down' through the the history as necessary. This is an incremental search.

`non-incremental-reverse-search-history (M-p)`

Search backward starting at the current line and moving `up' through the history as necessary using a non-incremental search for a string supplied by the user.

`non-incremental-forward-search-history (M-n)`

Search forward starting at the current line and moving `down' through the the history as necessary using a non-incremental search for a string supplied by the user.

`history-search-forward ()`

Search forward through the history for the string of characters between the start of the current line and the current cursor position (the point). This is a non-incremental search. By default, this command is unbound.

`history-search-backward ()`

Search backward through the history for the string of characters between the start of the current line and the point. This is a non-incremental search. By default, this command is unbound.

`yank-nth-arg (M-C-y)`

Insert the first argument to the previous command (usually the second word on the previous line). With an argument n, insert the nth word from the previous command (the words in the previous command begin with word 0). A negative argument inserts the nth word from the end of the previous command.

`yank-last-arg (M-., M-_)`

TOC

Insert last argument to the previous command (the last word of the previous history entry). With an argum behave exactly like `yank-nth-arg`. Successive calls to `yank-last-arg` move back through the his list, inserting the last argument of each line in turn.

Commands For Changing Text

`delete-char (C-d)`

Delete the character under the cursor. If the cursor is at the beginning of the line, there are no characters in the line, and the last character typed was not bound to `delete-char`, then return `EOF`.

`backward-delete-char (Rubout)`

Delete the character behind the cursor. A numeric argument means to kill the characters instead of deleting them.

`quoted-insert (C-q, C-v)`

Add the next character typed to the line verbatim. This is how to insert key sequences like C-q, for example.

`tab-insert (M-TAB)`

Insert a tab character.

`self-insert (a, b, A, 1, !, ...)`

Insert yourself.

`transpose-chars (C-t)`

Drag the character before the cursor forward over the character at the cursor, moving the cursor forward as well. If the insertion point is at the end of the line, then this transposes the last two characters of the line. Negative arguments don't work.

`transpose-words (M-t)`

Drag the word behind the cursor past the word in front of the cursor moving the cursor over that word as well.

`upcase-word (M-u)`

Uppercase the current (or following) word. With a negative argument, uppercase the previous word, but do not move the cursor.

`downcase-word (M-l)`

Lowercase the current (or following) word. With a negative argument, lowercase the previous word, but do not move the cursor.

`capitalize-word (M-c)`

Capitalize the current (or following) word. With a negative argument, capitalize the previous word, but do not move the cursor.

Killing And Yanking

`kill-line (C-k)`

Kill the text from the current cursor position to the end of the line.

`backward-kill-line (C-x Rubout)`

Kill backward to the beginning of the line.

`unix-line-discard (C-u)`

Kill backward from the cursor to the beginning of the current line. The killed text is saved on the kill-ring.

`kill-whole-line ()`

Kill all characters on the current line, no matter where the cursor is. By default, this is unbound.

`kill-word (M-d)`

Kill from the cursor to the end of the current word, or if between words, to the end of the next word. Word boundaries are the same as `forward-word`.

`backward-kill-word (M-DEL)`

Kill the word behind the cursor. Word boundaries are the same as `backward-word`.

`unix-word-rubout (C-w)`

Kill the word behind the cursor, using white space as a word boundary. The killed text is saved on the kill-ring.

`delete-horizontal-space ()`

Delete all spaces and tabs around point. By default, this is unbound.

`kill-region ()`

Kill the text between the point and the *mark* (saved cursor position). This text is referred to as the region. By default, this command is unbound.

`copy-region-as-kill ()`

Copy the text in the region to the kill buffer, so it can be yanked right away. By default, this command is unbound.

`copy-backward-word ()`

Copy the word before point to the kill buffer. The word boundaries are the same as `backward-word`. By default, this command is unbound.

`copy-forward-word ()`

Copy the word following point to the kill buffer. The word boundaries are the same as `forward-word`. By default, this command is unbound.

`yank (C-y)`

Yank the top of the kill ring into the buffer at the current cursor position.

`yank-pop (M-y)`

Rotate the kill-ring, and yank the new top. You can only do this if the prior command is yank or yank-pop.

Specifying Numeric Arguments

`digit-argument (M-0, M-1, ... M--)`

Add this digit to the argument already accumulating, or start a new argument. M-- starts a negative argument.

`universal-argument ()`

This is another way to specify an argument. If this command is followed by one or more digits, optionally with a leading minus sign, those digits define the argument. If the command is followed by digits, executing `universal-argument` again ends the numeric argument, but is otherwise ignored. As a special case, if this command is immediately followed by a character that is neither a digit or minus sign, the argument count for the next command is multiplied by four. The argument count is initially one, so executing this function the first time makes the argument count four, a second time makes the argument count sixteen, and so on. By default, this is not bound to a key.

Letting Readline Type For You

`complete (TAB)`

Attempt to do completion on the text before the cursor. This is application-specific. Generally, if you are typing a filename argument, you can do filename completion; if you are typing a command, you can do command completion; if you are typing in a symbol to GDB, you can do symbol name completion; if you are typing in a

variable to Bash, you can do variable name completion, and so on.

`possible-completions (M-?)`

List the possible completions of the text before the cursor.

`insert-completions (M-*)`

Insert all completions of the text before point that would have been generated by `possible-completions`.

`menu-complete ()`

Similar to `complete`, but replaces the word to be completed with a single match from the list of possible completions. Repeated execution of `menu-complete` steps through the list of possible completions, inserting each match in turn. At the end of the list of completions, the bell is rung and the original text is restored. An argument of n moves n positions forward in the list of matches; a negative argument may be used to move backward through the list. This command is intended to be bound to `TAB`, but is unbound by default.

Keyboard Macros

`start-kbd-macro (C-x ()`

Begin saving the characters typed into the current keyboard macro.

`end-kbd-macro (C-x ))`

Stop saving the characters typed into the current keyboard macro and save the definition.

`call-last-kbd-macro (C-x e)`

Re-execute the last keyboard macro defined, by making the characters in the macro appear as if typed at the keyboard.

Some Miscellaneous Commands

`re-read-init-file (C-x C-r)`

Read in the contents of the inputrc file, and incorporate any bindings or variable assignments found there.

`abort (C-g)`

Abort the current editing command and ring the terminal's bell (subject to the setting of `bell-style`).

`do-uppercase-version (M-a, M-b, M-x, ...)`

If the metafied character x is lowercase, run the command that is bound to the corresponding uppercase character.

`prefix-meta (ESC)`

Make the next character typed be metafied. This is for keyboards without a meta key. Typing `ESC f' is equivalent to typing `M-f'.

`undo (C-_, C-x C-u)`

Incremental undo, separately remembered for each line.

`revert-line (M-r)`

Undo all changes made to this line. This is like executing the `undo` command enough times to get back to the beginning.

`tilde-expand (M-~)`

Perform tilde expansion on the current word.

`set-mark (C-@)`

Set the mark to the current point. If a numeric argument is supplied, the mark is set to that position.

`exchange-point-and-mark (C-x C-x)`

Swap the point with the mark. The current cursor position is set to the saved position, and the old cursor position is saved as the mark.

`character-search (C-])`

A character is read and point is moved to the next occurrence of that character. A negative count searches for previous occurrences.

`character-search-backward (M-C-])`

A character is read and point is moved to the previous occurrence of that character. A negative count searches for subsequent occurrences.

`insert-comment (M-#)`

The value of the `comment-begin` variable is inserted at the beginning of the current line, and the line is accepted as if a newline had been typed.

`dump-functions ()`

Print all of the functions and their key bindings to the Readline output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an inputrc file. This command is unbound by default.

`dump-variables ()`

Print all of the settable variables and their values to the Readline output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an inputrc file. This command is unbound by default.

`dump-macros ()`

Print all of the Readline key sequences bound to macros and the strings they ouput. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an inputrc file. This command is unbound by default.

Readline vi Mode

While the Readline library does not have a full set of `vi` editing functions, it does contain enough to allow simple editing of the line. The Readline `vi` mode behaves as specified in the POSIX 1003.2 standard.

In order to switch interactively between `emacs` and `vi` editing modes, use the command M-C-j (toggle-editing-mode). The Readline default is `emacs` mode.

When you enter a line in `vi` mode, you are already placed in `insertion' mode, as if you had typed an `i'. Pressing ESC switches you into `command' mode, where you can edit the text of the line with the standard `vi` movement keys, move to previous history lines with `k' and subsequent lines with `j', and so forth.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Using History Interactively

This chapter describes how to use the GNU History Library interactively, from a user's standpoint.

History Interaction

The History library provides a history expansion feature similar to the history expansion in `csh`. The following text describes the syntax you use to manipulate history information.

History expansion takes two parts. In the first part, determine which line from the previous history will be used for substitution. This line is called the **event**. In the second part, select portions of that line for inclusion into the current line. These portions are called **words**. GDB breaks the line into words in the same way that the Bash shell does, so that several English (or Unix) words surrounded by quotes are considered one word.

Event Designators

An **event designator** is a reference to a command line entry in the history list.

`!`
: Start a history subsititution, except when followed by a space, tab, or the end of the line... = or (.

`!!`
: Refer to the previous command. This is a synonym for `!-1`.

`!n`
: Refer to command line n.

`!-n`
: Refer to the command line n lines back.

`!string`
: Refer to the most recent command starting with string.

`!?string[?]`
: Refer to the most recent command containing string.

Word Designators

A : separates the event designator from the **word designator**. It can be omitted if the word designator begins with a ^, $, * or %. Words are numbered from the beginning of the line, with the first word being denoted by a 0 (zero).

`0 (zero)`

The zero'th word. For many applications, this is the command word.

`n`

The n'th word.

`^`

The first argument. that is, word 1.

`$`

The last argument.

`%`

The word matched by the most recent `?string?` search.

`x-y`

A range of words; `-y` Abbreviates `0-y`.

`*`

All of the words, excepting the zero'th. This is a synonym for `1-$`. It is not an error to use * if there is just one word in the event. The empty string is returned in that case.

Modifiers

After the optional word designator, you can add a sequence of one or more of the following **modifiers**, each preceded by a :.

`#`

The entire command line typed so far. This means the current command, not the previous command.

`h`

Remove a trailing pathname component, leaving only the head.

`r`

Remove a trailing suffix of the form `.'suffix, leaving the basename.

`e`

Remove all but the suffix.

`t`

Remove all leading pathname components, leaving the tail.

`p`

Print the new command but do not execute it.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Formatting Documentation

The GDB 4 release includes an already-formatted reference card, ready for printing with PostScript or Ghostscript, in the `gdb' subdirectory of the main source directory(4). If you can use PostScript or Ghostscript with your printer, you can print the reference card immediately with `refcard.ps'.

The release also includes the source for the reference card. You can format it, using TeX, by typing:

```
make refcard.dvi
```

The GDB reference card is designed to print in **landscape** mode on US "letter" size paper; that is, on a sheet 11 inches wide by 8.5 inches high. You will need to specify this form of printing as an option to your DVI output program.

All the documentation for GDB comes as part of the machine-readable distribution. The documentation is written in Texinfo format, which is a documentation system that uses a single source file to produce both on-line information and a printed manual. You can use one of the Info formatting commands to create the on-line version of the documentation and TeX (or `texi2roff`) to typeset the printed version.

GDB includes an already formatted copy of the on-line Info version of this manual in the `gdb' subdirectory. The main Info file is `gdb-4.18/gdb/gdb.info', and it refers to subordinate files matching `gdb.info*' in the same directory. If necessary, you can print out these files, or read them with any editor; but they are easier to read using the `info` subsystem in GNU Emacs or the standalone `info` program, available as part of the GNU Texinfo distribution.

If you want to format these Info files yourself, you need one of the Info formatting programs, such as `texinfo-format-buffer` or `makeinfo`.

If you have `makeinfo` installed, and are in the top level GDB source directory (`gdb-4.18', in the case of version 4.18), you can make the Info file by typing:

```
cd gdb
make gdb.info
```

If you want to typeset and print copies of this manual, you need TeX, a program to print its DVI output files, and `texinfo.tex', the Texinfo definitions file.

TeX is a typesetting program; it does not print files directly, but produces output files called DVI files. To print a typeset document, you need a program to print DVI files. If your system has TeX installed, chances are it has such a program. The precise command to use depends on your system; lpr -d is common; another (for PostScript devices) is dvips. The DVI print command may require a file name without any extension or a `.dvi' extension.

TeX also requires a macro definitions file called `texinfo.tex'. This file tells TeX how to typeset a document written in Texinfo format. On its own, TeX cannot either read or typeset a Texinfo file. `texinfo.tex' is distributed with GDB and is located in the `gdb-version-number/texinfo' directory.

If you have TeX and a DVI printer program installed, you can typeset and print this manual. First switch to the the `gdb' subdirectory of the main source directory (for example, to `gdb-4.18/gdb') and type:

```
make gdb.dvi
```

Then give `gdb.dvi' to your DVI printing program.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Installing GDB

GDB comes with a `configure` script that automates the process of preparing GDB for installation; you can then use `make` to build the `gdb` program. (5)

The GDB distribution includes all the source code you need for GDB in a single directory, whose name is usually composed by appending the version number to `gdb'.

For example, the GDB version 4.18 distribution is in the `` `gdb-4.18' `` directory. That directory contains:

`gdb-4.18/configure (and supporting files)`
: script for configuring GDB and all its supporting libraries

`gdb-4.18/gdb`
: the source specific to GDB itself

`gdb-4.18/bfd`
: source for the Binary File Descriptor library

`gdb-4.18/include`
: GNU include files

`gdb-4.18/libiberty`
: source for the `-liberty' free software library

`gdb-4.18/opcodes`
: source for the library of opcode tables and disassemblers

`gdb-4.18/readline`
: source for the GNU command-line interface

`gdb-4.18/glob`
: source for the GNU filename pattern-matching subroutine

`gdb-4.18/mmalloc`
: source for the GNU memory-mapped malloc package

The simplest way to configure and build GDB is to run `configure` from the `` `gdb-version-number' `` source directory, which in this example is the `` `gdb-4.18' `` directory.

First switch to the `` `gdb-version-number' `` source directory if you are not already in it; then run `configure`. Pass the identifier for the platform on which GDB will run as an argument.

For example:

```
cd gdb-4.18
```


```
./configure host
make
```

where host is an identifier such as `sun4' or `decstation', that identifies the platform where GDB will run. (You can often leave off host; `configure` tries to guess the correct value by examining your system.)

Running `configure host' and then running `make` builds the `bfd', `readline', `mmalloc', and `libiberty' libraries, then `gdb` itself. The configured source files, and the binaries, are left in the corresponding source directories.

`configure` is a Bourne-shell (`/bin/sh`) script; if your system does not recognize this automatically when you run a different shell, you may need to run `sh` on it explicitly:

```
sh configure host
```

If you run `configure` from a directory that contains source directories for multiple libraries or programs, such as the `gdb-4.18' source directory for version 4.18, `configure` creates configuration files for every directory level underneath (unless you tell it not to, with the `--norecursion' option).

You can run the `configure` script from any of the subordinate directories in the GDB distribution if you only want to configure that subdirectory, but be sure to specify a path to it.

For example, with version 4.18, type the following to configure only the `bfd` subdirectory:

```
cd gdb-4.18/bfd
../configure host
```

You can install `gdb` anywhere; it has no hardwired paths. However, you should make sure that the shell on your path (named by the `SHELL' environment variable) is publicly readable. Remember that GDB uses the shell to start your program--some systems refuse to let GDB debug child processes whose programs are not readable.

Compiling GDB in another directory

If you want to run GDB versions for several host or target machines, you need a different `gdb` compiled for each combination of host and target. `configure` is designed to make this easy by allowing you to generate each configuration in a separate subdirectory, rather than in the source directory. If your `make` program handles the `VPATH' feature (GNU `make` does), running `make` in each of these directories builds the `gdb` program specified there.

To build `gdb` in a separate directory, run `configure` with the `--srcdir' option to specify where to find the source. (You also need to specify a path to find `configure` itself from your working directory. If the path to `configure` would be the same as the argument to `--srcdir', you can leave out the `--srcdir' option; it is assumed.)

For example, with version 4.18, you can build GDB in a separate directory for a Sun 4 like this:


```
cd gdb-4.18
mkdir ../gdb-sun4
cd ../gdb-sun4
../gdb-4.18/configure sun4
make
```

When `configure` builds a configuration using a remote source directory, it creates a tree for the binaries with the same structure (and using the same names) as the tree under the source directory. In the example, you'd find the Sun 4 library `` `libiberty.a' `` in the directory `` `gdb-sun4/libiberty' ``, and GDB itself in `` `gdb-sun4/gdb' ``.

One popular reason to build several GDB configurations in separate directories is to configure GDB for cross-compiling (where GDB runs on one machine--the **host**---while debugging programs that run on another machine--the **target**). You specify a cross-debugging target by giving the `--target=target' option to `configure`.

When you run `make` to build a program or library, you must run it in a configured directory--whatever directory you were in when you called `configure` (or one of its subdirectories).

The `Makefile` that `configure` generates in each source directory also runs recursively. If you type `make` in a source directory such as `` `gdb-4.18' `` (or in a separate configured directory configured with `--srcdir=dirname/gdb-4.18'), you will build all the required libraries, and then build GDB.

When you have multiple hosts or targets configured in separate directories, you can run `make` on them in parallel (for example, if they are NFS-mounted on each of the hosts); they will not interfere with each other.

Specifying names for hosts and targets

The specifications used for hosts and targets in the `configure` script are based on a three-part naming scheme, but some short predefined aliases are also supported. The full naming scheme encodes three pieces of information in the following pattern:

```
architecture-vendor-os
```

For example, you can use the alias `sun4` as a host argument, or as the value for target in a `--target=target` option. The equivalent full name is `sparc-sun-sunos4'.

The `configure` script accompanying GDB does not provide any query facility to list all supported host and target names or aliases. `configure` calls the Bourne shell script `config.sub` to map abbreviations to full names; you can read the script, if you wish, or you can use it to test your guesses on abbreviations--for example:

```
% sh config.sub i386-linux
i386-pc-linux-gnu
% sh config.sub alpha-linux
```


```
alpha-unknown-linux-gnu
% sh config.sub hp9k700
hppa1.1-hp-hpux
% sh config.sub sun4
sparc-sun-sunos4.1.1
% sh config.sub sun3
m68k-sun-sunos4.1.1
% sh config.sub i986v
Invalid configuration `i986v': machine `i986v' not recognized
```

`config.sub` is also distributed in the GDB source directory (`gdb-4.18', for version 4.18).

`configure` options

Here is a summary of the `configure` options and arguments that are most often useful for building GDB. `configure` also has several other options not listed here. See Info file `configure.info', node `What Configure Does', for a full explanation of `configure`.

```
configure [--help]
          [--prefix=dir]
          [--exec-prefix=dir]
          [--srcdir=dirname]
          [--norecursion] [--rm]
          [--target=target]
          host
```

You may introduce options with a single `-' rather than `--' if you prefer; but you may abbreviate option names if you use `--'.

`--help`
: Display a quick summary of how to invoke `configure`.

`--prefix=dir`
: Configure the source to install programs and files under directory `dir'.

`--exec-prefix=dir`
: Configure the source to install programs under directory `dir'.

`--srcdir=dirname`
: **Warning: using this option requires GNU `make`, or another `make` that implements the `VPATH` feature.**
Use this option to make configurations in directories separate from the GDB source directories. Among other things, you can use this to build (or maintain) several configurations simultaneously, in separate directories. `configure` writes configuration specific files in the current directory, but arranges for them to use the source in the directory dirname. `configure` creates directories under the working directory in parallel to the source directories below dirname.

`--norecursion`

Configure only the directory level where `configure` is executed; do not propagate configura to subdirectories.

`--target=target`

Configure GDB for cross-debugging programs running on the specified target. Without this option, GDB is configured to debug programs that run on the same machine (host) as GDB itself. There is no convenient way to generate a list of all available targets.

`host ...`

Configure GDB to run on the specified host. There is no convenient way to generate a list of all available hosts.

There are many other options available as well, but they are generally needed for special purposes only.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Index

#

$

.

/

:

@

a

b

c

d

e

f

g

h

i

n

o

p

q

r

S

t

u

v

w

X

Z

{

Go to the first, previous, next, last section, table of contents.

Debugging with GDB

The GNU Source-Level Debugger

Seventh Edition, for GDB version 4.18

February 1999

Richard M. Stallman and Roland H. Pesch

---

(1)

`b' cannot be used because these format letters are also used with the `x` command, where `b' stands for "byte"; see section Examining memory.

(2)

This is a way of removing one word from the stack, on machines where stacks grow downward in memory (most machines, nowadays). This assumes that the innermost stack frame is selected; setting `$sp` is not allowed when other stack frames are selected. To pop entire frames off the stack, regardless of machine architecture, use `return`; see section Returning from a function.

(3)

If you choose a port number that conflicts with another service, `gdbserver` prints an error message and exits.

(4)

In `` `gdb-4.18/gdb/refcard.ps' `` of the version 4.18 release.

(5)

If you have a more recent version of GDB than 4.18, look at the `` `README' `` file in the sources; we may have improved the installation procedures since publishing this manual.

This document was generated on 20 May 1999 using the texi2html translator version 1.51a.

gdbm

Go to the next section.

Copying Conditions.

This library is free; this means that everyone is free to use it and free to redistribute it on a free basis. GNU `dbm` (`gdbm`) is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of `gdbm` that they might get from you.

Specifically, we want to make sure that you have the right to give away copies `gdbm`, that you receive source code or else can get it if you want it, that you can change these functions or use pieces of them in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies `gdbm`, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for anything in the `gdbm` distribution. If these functions are modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

`gdbm` is currently distributed under the terms of the GNU General Public License, Version 2. (*NOT* under the GNU General Library Public License.) A copy the GNU General Public License is included with the distribution of `gdbm`.

Go to the next section.

Go to the previous, next section.

Introduction to GNU dbm.

GNU `dbm` (`gdbm`)is a library of database functions that use extendible hashing and works similar to the standard UNIX `dbm` functions. These routines are provided to a programmer needing to create and manipulate a hashed database. (`gdbm` is *NOT* a complete database package for an end user.)

The basic use of `gdbm` is to store key/data pairs in a data file. Each key must be unique and each key is paired with only one data item. The keys can not be directly accessed in sorted order. The basic unit of data in `gdbm` is the structure:

```
typedef struct {
           char *dptr;
           int  dsize;
         } datum;
```

This structure allows for arbitrary sized keys and data items.

The key/data pairs are stored in a `gdbm` disk file, called a `gdbm` database. An application must open a `gdbm` database to be able manipulate the keys and data contained in the database. `gdbm` allows an application to have multiple databases open at the same time. When an application opens a `gdbm` database, it is designated as a `reader` or a `writer`. A `gdbm` database opened by at most one writer at a time. However, many readers may open the database open simultaneously. Readers and writers can not open the `gdbm` database at the same time.

Go to the previous, next section.

Go to the previous, next section.

List of functions.

The following is a quick list of the functions contained in the `gdbm` library. The include file `gdbm.h`, that can be included by the user, contains a definition of these functions.

```
#include <gdbm.h>

GDBM_FILE gdbm_open(name, block_size, flags, mode, fatal_func);
void gdbm_close(dbf);
int gdbm_store(dbf, key, content, flag);
datum gdbm_fetch(dbf, key);
int gdbm_delete(dbf, key);
datum gdbm_firstkey(dbf);
datum gdbm_nextkey(dbf, key);
int gdbm_reorganize(dbf);
void gdbm_sync(dbf);
int gdbm_exists(dbf, key);
char *gdbm_strerror(errno);
int gdbm_setopt(dbf, option, value, size)
```

The `gdbm.h` include file is often in the `` `/usr/local/gnu/include' `` directory. (The actual location of `gdbm.h` depends on your local installation of `gdbm`.)

Go to the previous, next section.

Go to the previous, next section.

Opening the database.

Initialize gdbm system. If the file has a size of zero bytes, a file initialization procedure is performed, setting up the initial structure in the file.

The procedure for opening a gdbm file is:

```
GDBM_FILE dbf;

dbf = gdbm_open(name, block_size, flags, mode, fatal_func);
```

The parameters are:

char *name

The name of the file (the complete name, gdbm does not append any characters to this name).

int block_size

It is used during initialization to determine the size of various constructs. It is the size of a single transfer from disk to memory. This parameter is ignored if the file has been previously initialized. The minimum size is 512. If the value is less than 512, the file system blocksize is used, otherwise the value of block_size is used.

int flags

If flags is set to GDBM_READER, the user wants to just read the database and any call to gdbm_store or gdbm_delete will fail. Many readers can access the database at the same time. If flags is set to GDBM_WRITER, the user wants both read and write access to the database and requires exclusive access. If flags is set to GDBM_WRCREAT, the user wants both read and write access to the database and if the database does not exist, create a new one. If flags is set to GDBM_NEWDB, the user want a new database created, regardless of whether one existed, and wants read and write access to the new database. For all writers (GDBM_WRITER, GDBM_WRCREAT and GDBM_NEWDB) the value GDBM_FAST can be added to the flags field using logical or. This option causes gdbm to write the database without any disk file synchronization. This allows faster writes, but may produce an inconsistent database in the event of abnormal termination of the writer. Any error detected will cause a return value of NULL and an appropriate value will be in gdbm_errno (see Variables). If no errors occur, a pointer to the gdbm file descriptor will be returned.

int mode

File mode (see chmod(2) and open(2) if the file is created).

void (*fatal_func) ()

A function for gdbm to call if it detects a fatal error. The only parameter of this function is a string. If the value of NULL is provided, gdbm will use a default function.

The return value, dbf, is the pointer needed by all other functions to access that gdbm file. If the return

is the NULL pointer, `gdbm_open` was not successful. The errors can be found in `gdbm_errno` for `gdbm` errors and in `errno` for file system errors (for error codes, see `gdbm.h`).

In all of the following calls, the parameter `dbf` refers to the pointer returned from `gdbm_open`.

Go to the previous, next section.

Go to the previous, next section.

Closing the database.

It is important that every file opened is also closed. This is needed to update the reader/writer count on the file. This is done by:

```
gdbm_close(dbf);
```

The parameter is:

GDBM_FILE dbf

The pointer returned by `gdbm_open`.

Closes the `gdbm` file and frees all memory associated with the file `dbf`.

Go to the previous, next section.

Go to the previous, next section.

Inserting and replacing records in the database.

The function `gdbm_store` inserts or replaces records in the database.

```
ret = gdbm_store(dbf, key, content, flag);
```

The parameters are:

GDBM_FILE dbf

The pointer returned by `gdbm_open`.

datum key

The `key` data.

datum content

The data to be associated with the key.

int flag

Defines the action to take when the key is already in the database. The value GDBM_REPLACE (defined in `gdbm.h`) asks that the old data be replaced by the new `content`. The value GDBM_INSERT asks that an error be returned and no action taken if the `key` already exists.

The values returned in `ret` are:

-1

The item was not stored in the database because the caller was not an official writer or either `key` or `content` have a NULL dptr field. Both `key` and `content` must have the dptr field be a non-NULL value. Since a NULL dptr field is used by other functions to indicate an error, a NULL field cannot be valid data.

+1

The item was not stored because the argument `flag` was GDBM_INSERT and the `key` was already in the database.

0

No error. `content` is keyed by `key`. The file on disk is updated to reflect the structure of the new database before returning from this function.

If you store data for a `key` that is already in the data base, `gdbm` replaces the old data with the new data if called with GDBM_REPLACE. You do not get two data items for the same `key` and you do not get an error from `gdbm_store`.

The size in `gdbm` is not restricted like `dbm` or `ndbm`. Your data can be as large as you want.

Go to the previous, next section.

Go to the previous, next section.

Searching for records in the database.

Looks up a given `key` and returns the information associated with that key. The pointer in the structure that is returned is a pointer to dynamically allocated memory block. To search for some data:

```
content = gdbm_fetch(dbf, key);
```

The parameters are:

GDBM_FILE dbf

The pointer returned by `gdbm_open`.

datum key

The `key` data.

The datum returned in `content` is a pointer to the data found. If the dptr is NULL, no data was found. If dptr is not NULL, then it points to data allocated by malloc. `gdbm` does not automatically free this data. The user must free this storage when done using it. This eliminates the need to copy the result to save it for later use (you just save the pointer).

You may also search for a particular key without retrieving it, using:

```
ret = gdbm_exists(dbf, key);
```

The parameters are:

GDBM_FILE dbf

The pointer returned by `gdbm_open`.

datum key

The `key` data.

Unlike `gdbm_fetch`, this routine does not allocate any memory, and simply returns true or false, depending on whether the `key` exists, or not.

Go to the previous, next section.

Go to the previous, next section.

Removing records from the database.

To remove some data from the database:

```
ret = gdbm_delete(dbf, key);
```

The parameters are:

GDBM_FILE dbf

The pointer returned by `gdbm_open`.

datum key

The `key` data.

The ret value is -1 if the item is not present or the requester is a reader. The ret value is 0 if there was a successful delete.

`gdbm_delete` removes the keyed item and the `key` from the database `dbf`. The file on disk is updated to reflect the structure of the new database before returning from this function.

Go to the previous, next section.

Go to the previous, next section.

TOC

Sequential access to records.

The next two functions allow for accessing all items in the database. This access is not `key` sequential, but it is guaranteed to visit every `key` in the database once. The order has to do with the hash values. `gdbm_firstkey` starts the visit of all keys in the database. `gdbm_nextkey` finds and reads the next entry in the hash structure for `dbf`.

```
key = gdbm_firstkey(dbf);

nextkey = gdbm_nextkey(dbf, key);
```

The parameters are:

GDBM_FILE dbf

The pointer returned by `gdbm_open`.

datum `key`

datum nextkey

The `key` data.

The return values are both datum. If `key`.dptr or nextkey.dptr is NULL, there is no first `key` or next `key`. Again notice that dptr points to data allocated by malloc and `gdbm` will not free it for you.

These functions were intended to visit the database in read-only algorithms, for instance, to validate the database or similar operations.

File `visiting` is based on a `hash table`. `gdbm_delete` re-arranges the hash table to make sure that any collisions in the table do not leave some item `un-findable`. The original key order is NOT guaranteed to remain unchanged in ALL instances. It is possible that some key will not be visited if a loop like the following is executed:

```
key = gdbm_firstkey ( dbf );
while ( key.dptr ) {
   nextkey = gdbm_nextkey ( dbf, key );
   if ( some condition ) {
      gdbm_delete ( dbf, key );
      free ( key.dptr );
   }
   key = nextkey;
}
```

Go to the previous, next section.

Go to the previous, next section.

Database reorganization.

The following function should be used very seldom.

```
ret = gdbm_reorganize(dbf);
```

The parameter is:

GDBM_FILE dbf

The pointer returned by `gdbm_open`.

If you have had a lot of deletions and would like to shrink the space used by the `gdbm` file, this function will reorganize the database. `gdbm` will not shorten the length of a `gdbm` file (deleted file space will be reused) except by using this reorganization.

This reorganization requires creating a new file and inserting all the elements in the old file `dbf` into the new file. The new file is then renamed to the same name as the old file and `dbf` is updated to contain all the correct information about the new file. If an error is detected, the return value is negative. The value zero is returned after a successful reorganization.

Go to the previous, next section.

TOC

Go to the previous, next section.

Database Synchronization

If your database was opened with the GDBM_FAST flag, `gdbm` does not wait for writes to the disk to complete before continuing. This allows faster writing of databases at the risk of having a corrupted database if the application terminates in an abnormal fashion. The following function allows the programmer to make sure the disk version of the database has been completely updated with all changes to the current time.

```
gdbm_sync(dbf);
```

The parameter is:

GDBM_FILE dbf

> The pointer returned by `gdbm_open`.

This would usually be called after a complete set of changes have been made to the database and before some long waiting time. `gdbm_close` automatically calls the equivalent of `gdbm_sync` so no call is needed if the database is to be closed immediately after the set of changes have been made.

Go to the previous, next section.

Go to the previous, next section.

Error strings.

To convert a `gdbm` error code into English text, use this routine:

```
ret = gdbm_strerror(errno)
```

The parameter is:

gdbm_error errno

The `gdbm` error code, usually `gdbm_errno`.

The appropiate phrase for reading by humans is returned.

Go to the previous, next section.

Go to the previous, next section.

Seting options.

`Gdbm` now supports the ability to set certain options on an already open database.

```
ret = gdbm_setopt(dbf, option, value, size)
```

The parameters are:

GDBM_FILE dbf

The pointer returned by `gdbm_open`.

int option

The option to be set.

int *value

A pointer to the value to which `option` will be set.

int size

The length of the data pointed to by `value`.

The only legal option currently is GDBM_CACHESIZE, which sets the size of the internal bucket cache. This option may only be set once on each GDBM_FILE descriptor, and is set automatically to 100 upon the first access to the database.

The return value will be -1 upon failure, or 0 upon success. The global variable `gdbm_errno` will be set upon failure.

For instance, to set a database to use a cache of 10, after opening it with `gdbm_open`, but prior to accessing it in any way, the following code could be used:

```
int value = 10;
ret = gdbm_setopt(dbf, GDBM_CACHESIZE, &value, sizeof(int));
```

Go to the previous, next section.

Go to the previous, next section.

Two useful variables.

The following two variables are variables that may need to be used:

gdbm_error gdbm_errno

: The variable that contains more information about `gdbm` errors (`gdbm.h` has the definitions of the error values).

char * gdbm_version

: The string containing the version information.

Go to the previous, next section.

Go to the previous, next section.

Compatibility with standard dbm and ndbm.

GNU `dbm` files are not `sparse`. You can copy them with the UNIX `cp` command and they will not expand in the copying process.

There is a compatibility mode for use with programs that already use UNIX `dbm` and UNIX `ndbm`.

GNU `dbm` has compatibility functions for `dbm`. For `dbm` compatibility functions, you need the include file `dbm.h`.

In this compatibility mode, no `gdbm` file pointer is required by the user, and Only one file may be opened at a time. All users in compatibility mode are assumed to be writers. If the `gdbm` file is a read only, it will fail as a writer, but will also try to open it as a reader. All returned pointers in datum structures point to data that `gdbm` WILL free. They should be treated as static pointers (as standard UNIX `dbm` does). The compatibility function names are the same as the UNIX `dbm` function names. Their definitions follow:

```
int dbminit(name);
int store(key, content);
datum fetch(key);
int delete(key);
datum firstkey();
datum nextkey(key);
int dbmclose();
```

Standard UNIX `dbm` and GNU `dbm` do not have the same data format in the file. You cannot access a standard UNIX `dbm` file with GNU `dbm`! If you want to use an old database with GNU `dbm`, you must use the `conv2gdbm` program.

Also, GNU `dbm` has compatibility functions for `ndbm`. For `ndbm` compatibility functions, you need the include file `ndbm.h`.

Again, just like `ndbm`, any returned datum can be assumed to be static storage. You do not have to free that memory, the `ndbm` compatibility functions will do it for you.

The functions are:

```
DBM *dbm_open(name, flags, mode);
void dbm_close(file);
datum dbm_fetch(file, key);
int dbm_store(file, key, content, flags);
int dbm_delete(file, key);
```


```
datum dbm_firstkey(file);
datum dbm_nextkey(file);
int dbm_error(file);
int dbm_clearerr(file);
int dbm_dirfno(file);
int dbm_pagfno(file);
int dbm_rdonly(file);
```

If you want to compile an old C program that used UNIX `dbm` or `ndbm` and want to use `gdbm` files, execute the following `cc` command:

```
cc ... -L /usr/local/lib -lgdbm
```

Go to the previous, next section.

Go to the previous, next section.

Converting dbm files to gdbm format.

The program `conv2gdbm` has been provided to help you convert from `dbm` databases to `gdbm`. The usage is:

```
conv2gdbm [-q] [-b block_size] dbm_file [gdbm_file]
```

The options are:

-q
: Causes `conv2gdbm` to work quietly.

block_size
: Is the same as in `gdbm_open`.

dbm_file
: Is the name of the `dbm` file without the `.pag` or `.dir` extensions.

gdbm_file
: Is the complete file name. If not included, the `gdbm` file name is the same as the `dbm` file name without any extensions. That is `conv2gdbm dbmfile` converts the files `dbmfile.pag` and `dbmfile.dir` into a `gdbm` file called `dbmfile`.

Go to the previous, next section.

Go to the previous section.

Problems and bugs.

If you have problems with GNU `dbm` or think you've found a bug, please report it. Before reporting a bug, make sure you've actually found a real bug. Carefully reread the documentation and see if it really says you can do what you're trying to do. If it's not clear whether you should be able to do something or not, report that too; it's a bug in the documentation!

Before reporting a bug or trying to fix it yourself, try to isolate it to the smallest possible input file that reproduces the problem. Then send us the input file and the exact results `gdbm` gave you. Also say what you expected to occur; this will help us decide whether the problem was really in the documentation.

Once you've got a precise problem, send e-mail to:

```
Internet: `bug-gnu-utils@prep.ai.mit.edu'.
UUCP: `mit-eddie!prep.ai.mit.edu!bug-gnu-utils'.
```

Please include the version number of GNU `dbm` you are using. You can get this information by printing the variable `gdbm_version` (see Variables).

Non-bug suggestions are always welcome as well. If you have questions about things that are unclear in the documentation or are just obscure features, please report them too.

You may contact the author by:

```
   e-mail:  phil@cs.wwu.edu
 us-mail:  Philip A. Nelson
           Computer Science Department
           Western Washington University
           Bellingham, WA 98226
```

Go to the previous section.

Comparing and Merging Files

`diff`, `diff3`, `sdiff`, `cmp`, and `patch`

Edition 1.1, for `diff` 2.1 and `patch` 2.0.12g8

January 1993

by David MacKenzie, Paul Eggert, and Richard Stallman

Comparing and Merging Files

Overview

Computer users often find occasion to ask how two files differ. Perhaps one file is a newer version of the other file. Or maybe the two files started out as identical copies but were changed by different people.

You can use the `diff` command to show differences between two files, or each corresponding file in two directories. `diff` outputs differences between files line by line in any of several formats, selectable by command line options. This set of differences is often called a diff or patch. For files that are identical, `diff` normally produces no output; for binary (non-text) files, `diff` normally reports only that they are different.

You can use the `cmp` command to show the offsets and line numbers where two files differ. `cmp` can also show all the characters that differ between the two files, side by side. Another way to compare two files character by character is the Emacs command M-x compare-windows. See section 'Other Window' in The GNU Emacs Manual, for more information on that command.

You can use the `diff3` command to show differences among three files. When two people have made independent changes to a common original, `diff3` can report the differences between the original and the two changed versions, and can produce a merged file that contains both persons' changes together with warnings about conflicts.

You can use the `sdiff` command to merge two files interactively.

You can use the set of differences produced by `diff` to distribute updates to text files (such as program source code) to other people. This method is especially useful when the differences are small compared to the complete files. Given `diff` output, you can use the `patch` program to update, or patch, a copy of the file. If you think of `diff` as subtracting one file from another to produce their difference, you can think of `patch` as adding the difference to one file to reproduce the other.

This manual first concentrates on making diffs, and later shows how to use diffs to update files.

GNU `diff` was written by Mike Haertel, David Hayes, Richard Stallman, Len Tower, and Paul Eggert. Wayne Davison designed and implemented the unified output format. The basic algorithm is described in "An O(ND) Difference Algorithm and its Variations", Eugene Myers, Algorithmica Vol. 1 No. 2, 1986, p. 251; and in "A File Comparison Program", W. Miller and E. Myers, Software Practice and Experience Vol. 15 No. 11, 1985, p. 1025. GNU `diff3` was written by Randy Smith. GNU `sdiff` was written by Thomas Lord. GNU `cmp` was written by Torbjorn Granlund and David MacKenzie.

`patch` was written mainly by Larry Wall; the GNU enhancements were written mainly by Wayne Davison and David MacKenzie. Parts of this manual are adapted from a manual page written by Larry Wall, with his permission.

What Comparison Means

There are several ways to think about the differences between two files. One way to think of the differences is as a series of lines that were deleted from, inserted in, or changed in one file to produce the other file. `diff` compares two files line by line, finds groups of lines that differ, and reports each group of differing lines. It can report the differing lines in several formats, which have different purposes.

GNU `diff` can show whether files are different without detailing the differences. It also provides ways to suppress certain kinds of differences that are not important to you. Most commonly, such differences are changes in the amount of whitespace between words or lines. `diff` also provides ways to suppress differences in alphabetic case or in lines that match a regular expression that you provide. These options can accumulate; for example, you can ignore changes in both whitespace and alphabetic case.

Another way to think of the differences between two files is as a sequence of pairs of characters that can be either identical or different. `cmp` reports the differences between two files character by character, instead of line by line. As a result, it is more useful than `diff` for comparing binary files. For text files, `cmp` is useful mainly when you want to know only whether two files are identical. For this purpose, it is better than `diff` because it is much faster.

To illustrate the effect that considering changes character by character can have compared with considering them line by line, think of what happens if a single newline character is added to the beginning of a file. If that file is then compared with an otherwise identical file that lacks the newline at the beginning, `diff` will report that a blank line has been added to the file, while `cmp` will report that almost every character of the two files differs.

`diff3` normally compares three input files line by line, finds groups of lines that differ, and reports each group of differing lines. Its output is designed to make it easy to inspect two different sets of changes to the same file.

Hunks

When comparing two files, `diff` finds sequences of lines common to both files, interspersed with groups of differing lines called hunks. Comparing two identical files yields one sequence of common lines and no hunks, because no lines differ. Comparing two entirely different files yields no common lines and one large hunk that contains all lines of both files. In general, there are many ways to match up lines between two given files. `diff` tries to minimize the total hunk size by finding large sequences of common lines interspersed with small hunks of differing lines.

For example, suppose the file `` `F' `` contains the three lines `a', `b', `c', and the file `` `G' `` contains the same three lines in reverse order `c', `b', `a'. If `diff` finds the line `c' as common, then the command `diff F G' produces this output:

```
1,2d0
< a
< b
3a2,3
```


```
> b
> a
```

But if `diff` notices the common line `b' instead, it produces this output:

```
1c1
< a
---
> c
3c3
< c
---
> a
```

It is also possible to find `a' as the common line. `diff` does not always find an optimal matching between the files; it takes shortcuts to run faster. But its output is usually close to the shortest possible. You can adjust this tradeoff with the `--minimal' option (see section `diff` Performance Tradeoffs).

Suppressing Differences in Blank and Tab Spacing

The `-b' and `--ignore-space-change' options ignore blanks and tabs at line end, and to consider all other sequences of one or more blank and tab characters to be equivalent. With these options, `diff` considers the following two lines to be equivalent, where `$' denotes the line end:

```
Here lyeth  muche rychnesse  in lytell space.   -- John Heywood$
Here lyeth muche rychnesse in lytell space. -- John Heywood   $
```

The `-w' and `--ignore-all-space' options are stronger than `-b'. They ignore difference even if one file has whitespace where the other file has none, and they ignore all whitespace characters, not just blanks and tabs. (The whitespace characters include backspace, tab, vertical tab, formfeed, carriage return, space, and no-break space.) With these options, `diff` considers the following two lines to be equivalent, where `$' denotes the line end and `^M' denotes a carriage return:

```
Here lyeth  muche  rychnesse in lytell space.--  John Heywood$
  He relyeth much erychnes  seinly tells pace.  --John Heywood   ^M$
```

Suppressing Differences in Blank Lines

The `-B' and `--ignore-blank-lines' options ignore insertions or deletions of blank lines. These options normally affect only lines that are completely empty; they do not affect lines that look empty but contain space or tab characters. With these options, for example, a file containing

```
1.  A point is that which has no part.

2.  A line is breadthless length.
```


```
-- Euclid, The Elements, I
```

is considered identical to a file containing

```
1.  A point is that which has no part.
2.  A line is breadthless length.


-- Euclid, The Elements, I
```

Suppressing Case Differences

GNU `diff` can treat lowercase letters as equivalent to their uppercase counterparts, so that, for example, it considers `Funky Stuff', `funky STUFF', and `fUNKy stuFf' to all be the same. To request this, use the `-i' or `--ignore-case' option.

Suppressing Lines Matching a Regular Expression

To ignore insertions and deletions of lines that match a regular expression, use the `-I regexp' or `--ignore-matching-lines=regexp' option. You should escape regular expressions that contain shell metacharacters to prevent the shell from expanding them. For example, `diff -I '^[0-9]" ignores all changes to lines beginning with a digit.

However, `-I' only ignores the insertion or deletion of lines that contain the regular expression if every changed line in the hunk--every insertion and every deletion--matches the regular expression. In other words, for each nonignorable change, `diff` prints the complete set of changes in its vicinity, including the ignorable ones.

You can specify more than one regular expression for lines to ignore by using more than one `-I' option. `diff` tries to match each line against each regular expression, starting with the last one given.

Summarizing Which Files Differ

When you only want to find out whether files are different, and you don't care what the differences are, you can use the summary output format. In this format, instead of showing the differences between the files, `diff` simply reports whether files differ. The `-q' and `--brief' options select this output format.

This format is especially useful when comparing the contents of two directories. It is also much faster than doing the normal line by line comparisons, because `diff` can stop analyzing the files as soon as it knows that there are any differences.

You can also get a brief indication of whether two files differ by using `cmp`. For files that are identical, `cmp` produces no output. When the files differ, by default, `cmp` outputs the byte offset and line number where the first difference occurs. You can use the `-s' option to suppress that information, so that `cmp` produces no output and reports whether the files differ using only its exit status (see section Invoking `cmp`).

Unlike `diff`, `cmp` cannot compare directories; it can only compare two files.

Binary Files and Forcing Text Comparisons

If `diff` thinks that either of the two files it is comparing is binary (a non-text file), it normally treats that pair of files much as if the summary output format had been selected (see section Summarizing Which Files Differ), and reports only that the binary files are different. This is because line by line comparisons are usually not meaningful for binary files.

`diff` determines whether a file is text or binary by checking the first few bytes in the file; the exact number of bytes is system dependent, but it is typically several thousand. If every character in that part of the file is non-null, `diff` considers the file to be text; otherwise it considers the file to be binary.

Sometimes you might want to force `diff` to consider files to be text. For example, you might be comparing text files that contain null characters; `diff` would erroneously decide that those are non-text files. Or you might be comparing documents that are in a format used by a word processing system that uses null characters to indicate special formatting. You can force `diff` to consider all files to be text files, and compare them line by line, by using the `-a' or `--text' option. If the files you compare using this option do not in fact contain text, they will probably contain few newline characters, and the `diff` output will consist of hunks showing differences between long lines of whatever characters the files contain.

You can also force `diff` to consider all files to be binary files, and report only whether they differ (but not how). Use the `--brief' option for this.

If you want to compare two files byte by byte, you can use the `cmp` program with the `-l' option to show the values of each differing byte in the two files. With GNU `cmp`, you can also use the `-c' option to show the ASCII representation of those bytes. See section Invoking `cmp`, for more information.

If `diff3` thinks that any of the files it is comparing is binary (a non-text file), it normally reports an error, because such comparisons are usually not useful. `diff3` uses the same test as `diff` to decide whether a file is binary. As with `diff`, if the input files contain a few non-text characters but otherwise are like text files, you can force `diff3` to consider all files to be text files and compare them line by line by using the `-a' or `--text' options.

`diff` Output Formats

`diff` has several mutually exclusive options for output format. The following sections describe each format, illustrating how `diff` reports the differences between two sample input files.

Two Sample Input Files

Here are two sample files that we will use in numerous examples to illustrate the output of `diff` and how various options can change it.

This is the file `lao':


```
The Way that can be told of is not the eternal Way;
The name that can be named is not the eternal name.
The Nameless is the origin of Heaven and Earth;
The Named is the mother of all things.
Therefore let there always be non-being,
  so we may see their subtlety,
And let there always be being,
  so we may see their outcome.
The two are the same,
But after they are produced,
  they have different names.
```

This is the file `tzu':

```
The Nameless is the origin of Heaven and Earth;
The named is the mother of all things.

Therefore let there always be non-being,
  so we may see their subtlety,
And let there always be being,
  so we may see their outcome.
The two are the same,
But after they are produced,
  they have different names.
They both may be called deep and profound.
Deeper and more profound,
The door of all subtleties!
```

In this example, the first hunk contains just the first two lines of `lao', the second hunk contains the fourth line of `lao' opposing the second and third lines of `tzu', and the last hunk contains just the last three lines of `tzu'.

Showing Differences Without Context

The "normal" `diff` output format shows each hunk of differences without any surrounding context. Sometimes such output is the clearest way to see how lines have changed, without the clutter of nearby unchanged lines (although you can get similar results with the context or unified formats by using 0 lines of context). However, this format is no longer widely used for sending out patches; for that purpose, the context format (see section Context Format) and the unified format (see section Unified Format) are superior. Normal format is the default for compatibility with older versions of `diff` and the POSIX standard.

Detailed Description of Normal Format

The normal output format consists of one or more hunks of differences; each hunk shows one area where the files differ. Normal format hunks look like this:

```
change-command
< from-file-line
< from-file-line...
---
> to-file-line
> to-file-line...
```

There are three types of change commands. Each consists of a line number or comma-separated range of lines in the first file, a single character indicating the kind of change to make, and a line number or comma-separated range of lines in the second file. All line numbers are the original line numbers in each file. The types of change commands are:

`lar'

Add the lines in range r of the second file after line l of the first file. For example, `8a12,15' means append lines 12--15 of file 2 after line 8 of file 1; or, if changing file 2 into file 1, delete lines 12--15 of file 2.

`fct'

Replace the lines in range f of the first file with lines in range t of the second file. This is like a combined add and delete, but more compact. For example, `5,7c8,10' means change lines 5--7 of file 1 to read as lines 8--10 of file 2; or, if changing file 2 into file 1, change lines 8--10 of file 2 to read as lines 5--7 of file 1.

`rdl'

Delete the lines in range r from the first file; line l is where they would have appeared in the second file had they not been deleted. For example, `5,7d3' means delete lines 5--7 of file 1; or, if changing file 2 into file 1, append lines 5--7 of file 1 after line 3 of file 2.

An Example of Normal Format

Here is the output of the command `diff lao tzu' (see section Two Sample Input Files, for the complete contents of the two files). Notice that it shows only the lines that are different between the two files.

```
1,2d0
< The Way that can be told of is not the eternal Way;
< The name that can be named is not the eternal name.
4c2,3
< The Named is the mother of all things.
---
> The named is the mother of all things.
>
11a11,13
```


```
> They both may be called deep and profound.
> Deeper and more profound,
> The door of all subtleties!
```

Showing Differences in Their Context

Usually, when you are looking at the differences between files, you will also want to see the parts of the files near the lines that differ, to help you understand exactly what has changed. These nearby parts of the files are called the context.

GNU `diff` provides two output formats that show context around the differing lines: context format and unified format. It can optionally show in which function or section of the file the differing lines are found.

If you are distributing new versions of files to other people in the form of `diff` output, you should use one of the output formats that show context so that they can apply the diffs even if they have made small changes of their own to the files. `patch` can apply the diffs in this case by searching in the files for the lines of context around the differing lines; if those lines are actually a few lines away from where the diff says they are, `patch` can adjust the line numbers accordingly and still apply the diff correctly. See section Applying Imperfect Patches, for more information on using `patch` to apply imperfect diffs.

Context Format

The context output format shows several lines of context around the lines that differ. It is the standard format for distributing updates to source code.

To select this output format, use the `-C lines', `--context[=lines]', or `-c' option. The argument lines that some of these options take is the number of lines of context to show. If you do not specify lines, it defaults to three.

Detailed Description of Context Format

The context output format starts with a two-line header, which looks like this:

```
*** from-file from-file-modification-time
--- to-file to-file-modification time
```

You can change the header's content with the `-L label' or `--label=label' option; see section Showing Alternate File Names.

Next come one or more hunks of differences; each hunk shows one area where the files differ. Context format hunks look like this:

```
***************
*** from-file-line-range ****
  from-file-line
  from-file-line...
--- to-file-line-range ----
```


```
  to-file-line
  to-file-line...
```

The lines of context around the lines that differ start with two space characters. The lines that differ between the two files start with one of the following indicator characters, followed by a space character:

`!'

A line that is part of a group of one or more lines that changed between the two files. There is a corresponding group of lines marked with `!' in the part of this hunk for the other file.

`+'

An "inserted" line in the second file that corresponds to nothing in the first file.

`-'

A "deleted" line in the first file that corresponds to nothing in the second file.

If all of the changes in a hunk are insertions, the lines of from-file are omitted. If all of the changes are deletions, the lines of to-file are omitted.

An Example of Context Format

Here is the output of `diff -c lao tzu' (see section Two Sample Input Files, for the complete contents of the two files). Notice that up to three lines that are not different are shown around each line that is different; they are the context lines. Also notice that the first two hunks have run together, because their contents overlap.

```
*** lao Sat Jan 26 23:30:39 1991
--- tzu Sat Jan 26 23:30:50 1991
***************
*** 1,7 ****
- The Way that can be told of is not the eternal Way;
- The name that can be named is not the eternal name.
  The Nameless is the origin of Heaven and Earth;
! The Named is the mother of all things.
  Therefore let there always be non-being,
    so we may see their subtlety,
  And let there always be being,
--- 1,6 ----
  The Nameless is the origin of Heaven and Earth;
! The named is the mother of all things.
!
  Therefore let there always be non-being,
    so we may see their subtlety,
  And let there always be being,
***************
*** 9,11 ****
--- 8,13 ----
  The two are the same,
```


```
  But after they are produced,
    they have different names.
+ They both may be called deep and profound.
+ Deeper and more profound,
+ The door of all subtleties!
```

An Example of Context Format with Less Context

Here is the output of `diff --context=1 lao tzu' (see section Two Sample Input Files, for the complete contents of the two files). Notice that at most one context line is reported here.

```
*** lao Sat Jan 26 23:30:39 1991
--- tzu Sat Jan 26 23:30:50 1991
***************
*** 1,5 ****
- The Way that can be told of is not the eternal Way;
- The name that can be named is not the eternal name.
  The Nameless is the origin of Heaven and Earth;
! The Named is the mother of all things.
  Therefore let there always be non-being,
--- 1,4 ----
  The Nameless is the origin of Heaven and Earth;
! The named is the mother of all things.
!
  Therefore let there always be non-being,
***************
*** 11 ****
--- 10,13 ----
    they have different names.
+ They both may be called deep and profound.
+ Deeper and more profound,
+ The door of all subtleties!
```

Unified Format

The unified output format is a variation on the context format that is more compact because it omits redundant context lines. To select this output format, use the `-U lines', `--unified[=lines]', or `-u' option. The argument lines is the number of lines of context to show. When it is not given, it defaults to three.

At present, only GNU `diff` can produce this format and only GNU `patch` can automatically apply diffs in this format.

Detailed Description of Unified Format

The unified output format starts with a two-line header, which looks like this:


```
--- from-file from-file-modification-time
+++ to-file to-file-modification-time
```

You can change the header's content with the `-L label' or `--label=label' option; see See section Showing Alternate File Names.

Next come one or more hunks of differences; each hunk shows one area where the files differ. Unified format hunks look like this:

```
@@ from-file-range to-file-range @@
 line-from-either-file
 line-from-either-file...
```

The lines common to both files begin with a space character. The lines that actually differ between the two files have one of the following indicator characters in the left column:

`+'
: A line was added here to the first file.

`-'
: A line was removed here from the first file.

An Example of Unified Format

Here is the output of the command `diff -u lao tzu' (see section Two Sample Input Files, for the complete contents of the two files):

```
--- lao Sat Jan 26 23:30:39 1991
+++ tzu Sat Jan 26 23:30:50 1991
@@ -1,7 +1,6 @@
-The Way that can be told of is not the eternal Way;
-The name that can be named is not the eternal name.
 The Nameless is the origin of Heaven and Earth;
-The Named is the mother of all things.
+The named is the mother of all things.
+
 Therefore let there always be non-being,
   so we may see their subtlety,
 And let there always be being,
@@ -9,3 +8,6 @@
 The two are the same,
 But after they are produced,
   they have different names.
+They both may be called deep and profound.
+Deeper and more profound,
+The door of all subtleties!
```

Showing Which Sections Differences Are in

Sometimes you might want to know which part of the files each change falls in. If the files are source code, this could mean which function was changed. If the files are documents, it could mean which chapter or appendix was changed. GNU `diff` can show this by displaying the nearest section heading line that precedes the differing lines. Which lines are "section headings" is determined by a regular expression.

Showing Lines That Match Regular Expressions

To show in which sections differences occur for files that are not source code for C or similar languages, use the `-F regexp' or `--show-function-line=regexp' option. `diff` considers lines that match the argument regexp to be the beginning of a section of the file. Here are suggested regular expressions for some common languages:

`^[A-Za-z_]'
: C, C++, Prolog

`^('
: Lisp

`^@\(chapter\|appendix\|unnumbered\|chapheading\)'
: Texinfo

This option does not automatically select an output format; in order to use it, you must select the context format (see section Context Format) or unified format (see section Unified Format). In other output formats it has no effect.

The `-F' and `--show-function-line' options find the nearest unchanged line that precedes each hunk of differences and matches the given regular expression. Then they add that line to the end of the line of asterisks in the context format, or to the `@@' line in unified format. If no matching line exists, they leave the output for that hunk unchanged. If that line is more than 40 characters long, they output only the first 40 characters. You can specify more than one regular expression for such lines; `diff` tries to match each line against each regular expression, starting with the last one given. This means that you can use `-p' and `-F' together, if you wish.

Showing C Function Headings

To show in which functions differences occur for C and similar languages, you can use the `-p' or `--show-c-function' option. This option automatically selects the context output format (see section Context Format), with the default number of lines of context. You can override that number with `-C lines' later in the command line. You can override both the format and the number with `-U lines' later in the command line.

The `-p' and `--show-c-function' options are equivalent to `-c -F'^[_a-zA-Z$]'' (see section Showing Lines That Match Regular Expressions). GNU `diff` provides them for the sake of convenience.

Showing Alternate File Names

If you are comparing two files that have meaningless or uninformative names, you might want `diff` to show alternate names in the header of the context and unified output formats. To do this, use the `-L label' or `--label=label' option. The first time you give this option, its argument replaces the name and date of the first file in the header; the second time, its argument replaces the name and date of the second file. If you give this option more than twice, `diff` reports an error. The `-L' option does not affect the file names in the `pr` header when the `-l' or `--paginate' option is used (see section Paginating `diff` Output).

Here are the first two lines of the output from `diff -C2 -Loriginal -Lmodified lao tzu':

```
*** original
--- modified
```

Showing Differences Side by Side

`diff` can produce a side by side difference listing of two files. The files are listed in two columns with a gutter between them. The gutter contains one of the following markers:

white space
: The corresponding lines are in common. That is, either the lines are identical, or the difference is ignored because of one of the `--ignore' options (see section Suppressing Differences in Blank and Tab Spacing).

`|'
: The corresponding lines differ, and they are either both complete or both incomplete.

`<'
: The files differ and only the first file contains the line.

`>'
: The files differ and only the second file contains the line.

`('
: Only the first file contains the line, but the difference is ignored.

`)'
: Only the second file contains the line, but the difference is ignored.

`\'
: The corresponding lines differ, and only the first line is incomplete.

`/'
: The corresponding lines differ, and only the second line is incomplete.

Normally, an output line is incomplete if and only if the lines that it contains are incomplete; See section Incomplete Lines. However, when an output line represents two differing lines, one might be incomplete while the other is not. In this case, the output line is complete, but its the gutter is marked `\' if the first line is incomplete, `/' if the second line is.

Side by side format is sometimes easiest to read, but it has limitations. It generates much wider output than usual, and truncates lines that are too long to fit. Also, it relies on lining up output more heavily than usual, so its output looks particularly bad if you use varying width fonts, nonstandard tab stops, or nonprinting characters.

You can use the `sdiff` command to interactively merge side by side differences. See section Interactive Merging with `sdiff`, for more information on merging files.

Controlling Side by Side Format

The `-y' or `--side-by-side' option selects side by side format. Because side by side output lines contain two input lines, they are wider than usual. They are normally 130 columns, which can fit onto a traditional printer line. You can set the length of output lines with the `-W columns' or `--width=columns' option. The output line is split into two halves of equal length, separated by a small gutter to mark differences; the right half is aligned to a tab stop so that tabs line up. Input lines that are too long to fit in half of an output line are truncated for output.

The `--left-column' option prints only the left column of two common lines. The `--suppress-common-lines' option suppresses common lines entirely.

An Example of Side by Side Format

Here is the output of the command `diff -y -W 72 lao tzu' (see section Two Sample Input Files, for the complete contents of the two files).

```
The Way that can be told of is n   <
The name that can be named is no   <
The Nameless is the origin of He           The Nameless is the origin of He
The Named is the mother of all t   |       The named is the mother of all t
                                   >
Therefore let there always be no           Therefore let there always be no
  so we may see their subtlety,              so we may see their subtlety,
And let there always be being,             And let there always be being,
  so we may see their outcome.               so we may see their outcome.
The two are the same,                      The two are the same,
But after they are produced,               But after they are produced,
  they have different names.                 they have different names.
                                   >       They both may be called deep and
                                   >       Deeper and more profound,
                                   >       The door of all subtleties!
```

Making Edit Scripts

Several output modes produce command scripts for editing from-file to produce to-file.

ed Scripts

`diff` can produce commands that direct the `ed` text editor to change the first file into the second file. Long ago, this was the only output mode that was suitable for editing one file into another automatically; today, with `patch`, it is almost obsolete. Use the `-e' or `--ed' option to select this output format.

Like the normal format (see section Showing Differences Without Context), this output format does not show any context; unlike the normal format, it does not include the information necessary to apply the diff in reverse (to produce the first file if all you have is the second file and the diff).

If the file `d' contains the output of `diff -e old new', then the command `(cat d && echo w) | ed - old' edits `old' to make it a copy of `new'. More generally, if `d1', `d2', ..., `dN' contain the outputs of `diff -e old new1', `diff -e new1 new2', ..., `diff -e newN-1 newN', respectively, then the command `(cat d1 d2 ... dN && echo w) | ed - old' edits `old' to make it a copy of `newN'.

Detailed Description of ed Format

The `ed` output format consists of one or more hunks of differences. The changes closest to the ends of the files come first so that commands that change the number of lines do not affect how `ed` interprets line numbers in succeeding commands. `ed` format hunks look like this:

```
change-command
to-file-line
to-file-line...
.
```

Because `ed` uses a single period on a line to indicate the end of input, GNU `diff` protects lines of changes that contain a single period on a line by writing two periods instead, then writing a subsequent `ed` command to change the two periods into one. The `ed` format cannot represent an incomplete line, so if the second file ends in a changed incomplete line, `diff` reports an error and then pretends that a newline was appended.

There are three types of change commands. Each consists of a line number or comma-separated range of lines in the first file and a single character indicating the kind of change to make. All line numbers are the original line numbers in the file. The types of change commands are:

`la'

Add text from the second file after line l in the first file. For example, `8a' means to add the following lines after line 8 of file 1.

`rc'

Replace the lines in range r in the first file with the following lines. Like a combined add and delete, but more compact. For example, `5,7c' means change lines 5--7 of file 1 to read as the text file 2.

`rd'

Delete the lines in range r from the first file. For example, `5,7d' means delete lines 5--7 of file 1.

Example ed Script

Here is the output of `diff -e lao tzu' (see section Two Sample Input Files, for the complete contents of the two files):

```
11a
They both may be called deep and profound.
Deeper and more profound,
The door of all subtleties!
.
4c
The named is the mother of all things.

.
1,2d
```

Forward ed Scripts

`diff` can produce output that is like an `ed` script, but with hunks in forward (front to back) order. The format of the commands is also changed slightly: command characters precede the lines they modify, spaces separate line numbers in ranges, and no attempt is made to disambiguate hunk lines consisting of a single period. Like `ed` format, forward `ed` format cannot represent incomplete lines.

Forward `ed` format is not very useful, because neither `ed` nor `patch` can apply diffs in this format. It exists mainly for compatibility with older versions of `diff`. Use the `-f' or `--forward-ed' option to select it.

RCS Scripts

The RCS output format is designed specifically for use by the Revision Control System, which is a set of free programs used for organizing different versions and systems of files. Use the `-n' or `--rcs' option to select this output format. It is like the forward `ed` format (see section Forward `ed` Scripts), but it can represent arbitrary changes to the contents of a file because it avoids the forward `ed` format's problems with lines consisting of a single period and with incomplete lines. Instead of ending text sections with a line consisting of a single period, each command specifies the number of lines it affects; a combination of the `a' and `d' commands are used instead of `c'. Also, if the second file ends in a changed incomplete line, then the output also ends in an incomplete line.

Here is the output of `diff -n lao tzu' (see section Two Sample Input Files, for the complete contents of the two files):

```
d1 2
d4 1
a4 2
```


```
The named is the mother of all things.

a11 3
They both may be called deep and profound.
Deeper and more profound,
The door of all subtleties!
```

Merging Files with If-then-else

You can use `diff` to merge two files of C source code. The output of `diff` in this format contains all the lines of both files. Lines common to both files are output just once; the differing parts are separated by the C preprocessor directives `#ifdef name` or `#ifndef name`, `#else`, and `#endif`. When compiling the output, you select which version to use by either defining or leaving undefined the macro name.

To merge two files, use `diff` with the `-D name' or `--ifdef=name' option. The argument name is the C preprocessor identifier to use in the `#ifdef` and `#ifndef` directives.

For example, if you change an instance of `wait (&s)` to `waitpid (-1, &s, 0)` and then merge the old and new files with the `--ifdef=HAVE_WAITPID' option, then the affected part of your code might look like this:

```
    do {
#ifndef HAVE_WAITPID
        if ((w = wait (&s)) < 0  &&  errno != EINTR)
#else /* HAVE_WAITPID */
        if ((w = waitpid (-1, &s, 0)) < 0  &&  errno != EINTR)
#endif /* HAVE_WAITPID */
            return w;
    } while (w != child);
```

You can specify formats for languages other than C by using line group formats and line formats, as described in the next sections.

Line Group Formats

Line group formats let you specify formats suitable for many applications that allow if-then-else input, including programming languages and text formatting languages. A line group format specifies the output format for a contiguous group of similar lines.

For example, the following command compares the TeX files `old' and `new', and outputs a merged file in which old regions are surrounded by `\begin{em}'-`\end{em}' lines, and new regions are surrounded by `\begin{bf}'-`\end{bf}' lines.

```
diff \
   --old-group-format='\begin{em}
%<\end{em}
' \
```


```
   --new-group-format='\begin{bf}
%>\end{bf}
' \
   old new
```

The following command is equivalent to the above example, but it is a little more verbose, because it spells out the default line group formats.

```
diff \
   --old-group-format='\begin{em}
%<\end{em}
' \
   --new-group-format='\begin{bf}
%>\end{bf}
' \
   --unchanged-group-format='%=' \
   --changed-group-format='\begin{em}
%<\end{em}
\begin{bf}
%>\end{bf}
' \
   old new
```

To specify a line group format, use `diff` with one of the options listed below. You can specify up to four line group formats, one for each kind of line group. You should quote format, because it typically contains shell metacharacters.

`--old-group-format=format'

These line groups are hunks containing only lines from the first file. The default old group format is the same as the changed group format if it is specified; otherwise it a format that outputs the line group as-is.

`--new-group-format=format'

These line groups are hunks containing only lines from the second file. The default new group format is same as the the changed group format if it is specified; otherwise it is a format that outputs the line group as-is.

`--changed-group-format=format'

These line groups are hunks containing lines from both files. The default changed group format is the concatenation of the old and new group formats.

`--unchanged-group-format=format'

These line groups contain lines common to both files. The default unchanged group format is a format that outputs the line group as-is.

In a line group format, ordinary characters represent themselves; conversion specifications start with `%' and have one of the following forms.

`%<'

stands for the lines from the first file, including the trailing newline. Each line is formatted according

to the old line format (see section Line Formats).

`%>'

stands for the lines from the second file, including the trailing newline. Each line is formatted according to the new line format.

`%='

stands for the lines common to both files, including the trailing newline. Each line is formatted according to the unchanged line format.

`%0'

stands for a null character.

`%%'

stands for `%'.

Line Formats

Line formats control how each line taken from an input file is output as part of a line group in if-then-else format.

For example, the following command outputs text with a one-column change indicator to the left of the text. The first column of output is `-' for deleted lines, `|' for added lines, and a space for unchanged lines. The formats contain newline characters where newlines are desired on output.

```
diff \
   --old-line-format='-%l
' \
   --new-line-format='|%l
' \
   --unchanged-line-format=' %l
' \
   old new
```

To specify a line format, use one of the following options. You should quote format, since it often contains shell metacharacters.

`--old-line-format=format'

formats lines just from the first file.

`--new-line-format=format'

formats lines just from the second file.

`--unchanged-line-format=format'

formats lines common to both files.

In a line format, ordinary characters represent themselves; conversion specifications start with `%' and have one of the following forms.

`%l'

stands for the the contents of the line, not counting its trailing newline (if any). This format ignores

whether the line is incomplete; See section Incomplete Lines.

`%L'

stands for the the contents of the line, including its trailing newline (if any). If a line is incomplete, this format preserves its incompleteness.

`%0'

stands for a null character.

`%%'

stands for `%'.

The default line format is `%l' followed by a newline character.

If the input contains tab characters and it is important that they line up on output, you should ensure that `%l' or `%L' in a line format is just after a tab stop (e.g. by preceding `%l' or `%L' with a tab character), or you should use the `-t' or `--expand-tabs' option.

Detailed Description of If-then-else Format

For lines common to both files, `diff` uses the unchanged line group format. For each hunk of differences in the merged output format, if the hunk contains only lines from the first file, `diff` uses the old line group format; if the hunk contains only lines from the second file, `diff` uses the new group format; otherwise, `diff` uses the changed group format.

The old, new, and unchanged line formats specify the output format of lines from the first file, lines from the second file, and lines common to both files, respectively.

The option `--ifdef=name' is equivalent to the following sequence of options using shell syntax:

```
--old-group-format='#ifndef name
%<#endif /* not name */
' \
--new-group-format='#ifdef name
%>#endif /* name */
' \
--unchanged-group-format='%=' \
--changed-group-format='#ifndef name
%<#else /* name */
%>#endif /* name */
'
```

You should carefully check the `diff` output for proper nesting. For example, when using the the `-D name' or `--ifdef=name' option, you should check that if the differing lines contain any of the C preprocessor directives `#ifdef', `#ifndef', `#else', `#elif', or `#endif', they are nested properly and match. If they don't, you must make corrections manually. It is a good idea to carefully check the resulting code anyway to make sure that it really does what you want it to; depending on how the input files were produced, the output might contain duplicate or otherwise incorrect code.

The `patch` `-D name' option behaves just like the `diff` `-D name' option, except it operates on a file and

a diff to produce a merged file; See section Options to `patch`.

An Example of If-then-else Format

Here is the output of `diff -DTWO lao tzu' (see section Two Sample Input Files, for the complete contents of the two files):

```
#ifndef TWO
The Way that can be told of is not the eternal Way;
The name that can be named is not the eternal name.
#endif /* not TWO */
The Nameless is the origin of Heaven and Earth;
#ifndef TWO
The Named is the mother of all things.
#else /* TWO */
The named is the mother of all things.

#endif /* TWO */
Therefore let there always be non-being,
  so we may see their subtlety,
And let there always be being,
  so we may see their outcome.
The two are the same,
But after they are produced,
  they have different names.
#ifdef TWO
They both may be called deep and profound.
Deeper and more profound,
The door of all subtleties!
#endif /* TWO */
```

Comparing Directories

You can use `diff` to compare some or all of the files in two directory trees. When both file name arguments to `diff` are directories, it compares each file that is contained in both directories, examining file names in alphabetical order. Normally `diff` is silent about pairs of files that contain no differences, but if you use the `-s' or `--report-identical-files' option, it reports pairs of identical files. Normally `diff` reports subdirectories common to both directories without comparing subdirectories' files, but if you use the `-r' or `--recursive' option, it compares every corresponding pair of files in the directory trees, as many levels deep as they go.

For file names that are in only one of the directories, `diff` normally does not show the contents of the file that exists; it reports only that the file exists in that directory and not in the other. You can make `diff` act as though the file existed but was empty in the other directory, so that it outputs the entire contents of the file that actually exists. (It is output as either an insertion or a deletion, depending on whether it is in the

first or the second directory given.) To do this, use the `-N' or `--new-file' option.

If the older directory contains one or more large files that are not in the newer directory, you can make the patch smaller by using the `-P' or `--unidirectional-new-file' option instead of `-N'. This option is like `-N' except that it only inserts the contents of files that appear in the second directory but not the first (that is, files that were added). At the top of the patch, write instructions for the user applying the patch to remove the files that were deleted before applying the patch. See section Tips for Making Patch Distributions, for more discussion of making patches for distribution.

To ignore some files while comparing directories, use the `-x pattern' or `--exclude=pattern' option. This option ignores any files or subdirectories whose base names match the shell pattern pattern. Unlike in the shell, a period at the start of the base of a file name matches a wildcard at the start of a pattern. You should enclose pattern in quotes so that the shell does not expand it. For example, the option `-x '*.[ao]" ignores any file whose name ends with `.a' or `.o'.

This option accumulates if you specify it more than once. For example, using the options `-x 'RCS' -x '*,v" ignores any file or subdirectory whose base name is `RCS' or ends with `,v'.

If you need to give this option many times, you can instead put the patterns in a file, one pattern per line, and use the `-X file' or `--exclude-from=file' option.

If you have been comparing two directories and stopped partway through, later you might want to continue where you left off. You can do this by using the `-S file' or `--starting-file=file' option. This compares only the file file and all alphabetically later files in the topmost directory level.

Making `diff` Output Prettier

`diff` provides several ways to adjust the appearance of its output. These adjustments can be applied to any output format.

Preserving Tabstop Alignment

The lines of text in some of the `diff` output formats are preceded by one or two characters that indicate whether the text is inserted, deleted, or changed. The addition of those characters can cause tabs to move to the next tabstop, throwing off the alignment of columns in the line. GNU `diff` provides two ways to make tab-aligned columns line up correctly.

The first way is to have `diff` convert all tabs into the correct number of spaces before outputting them; select this method with the `-t' or `--expand-tabs' option. `diff` assumes that tabstops are set every 8 columns. To use this form of output with `patch`, you must give `patch` the `-l' or `--ignore-whitespace' option (see section Applying Patches with Changed Whitespace, for more information).

The other method for making tabs line up correctly is to add a tab character instead of a space after the indicator character at the beginning of the line. This ensures that all following tab characters are in the same position relative to tabstops that they were in the original files, so that the output is aligned correctly. Its disadvantage is that it can make long lines too long to fit on one line of the screen or the paper. It also does not work with the unified output format, which does not have a space character after the change type

indicator character. Select this method with the `-T' or `--initial-tab' option.

Paginating `diff` Output

It can be convenient to have long output page-numbered and time-stamped. The `-l' and `--paginate' options do this by sending the `diff` output through the `pr` program. Here is what the page header might look like for `diff -lc lao tzu':

```
Mar 11 13:37 1991  diff -lc lao tzu Page 1
```

`diff` Performance Tradeoffs

GNU `diff` runs quite efficiently; however, in some circumstances you can cause it to run faster or produce a more compact set of changes. There are two ways that you can affect the performance of GNU `diff` by changing the way it compares files.

Performance has more than one dimension. These options improve one aspect of performance at the cost of another, or they improve performance in some cases while hurting it in others.

The way that GNU `diff` determines which lines have changed always comes up with a near-minimal set of differences. Usually it is good enough for practical purposes. If the `diff` output is large, you might want `diff` to use a modified algorithm that sometimes produces a smaller set of differences. The `-d' or `--minimal' option does this; however, it can also cause `diff` to run more slowly than usual, so it is not the default behavior.

When the files you are comparing are large and have small groups of changes scattered throughout them, you can use the `-H' or `--speed-large-files' option to make a different modification to the algorithm that `diff` uses. If the input files have a constant small density of changes, this option speeds up the comparisons without changing the output. If not, `diff` might produce a larger set of differences; however, the output will still be correct.

Comparing Three Files

Use the program `diff3` to compare three files and show any differences among them. (`diff3` can also merge files; see section Merging From a Common Ancestor).

The "normal" `diff3` output format shows each hunk of differences without surrounding context. Hunks are labeled depending on whether they are two-way or three-way, and lines are annotated by their location in the input files.

See section Invoking `diff3`, for more information on how to run `diff3`.

A Third Sample Input File

Here is a third sample file that will be used in examples to illustrate the output of `diff3` and how various options can change it. The first two files are the same that we used for `diff` (see section Two Sample Input Files). This is the third sample file, called `` `tao' ``:

```
The Way that can be told of is not the eternal Way;
The name that can be named is not the eternal name.
The Nameless is the origin of Heaven and Earth;
The named is the mother of all things.

Therefore let there always be non-being,
  so we may see their subtlety,
And let there always be being,
  so we may see their result.
The two are the same,
But after they are produced,
  they have different names.

  -- The Way of Lao-Tzu, tr. Wing-tsit Chan
```

Detailed Description of `diff3` Normal Format

Each hunk begins with a line marked `` `====' ``. Three-way hunks have plain `` `====' `` lines, and two-way hunks have `1', `2', or `3' appended to specify which of the three input files differ in that hunk. The hunks contain copies of two or three sets of input lines each preceded by one or two commands identifying where the lines came from. Two spaces precede each copy of an input line to distinguish it from the commands. Commands take the following forms:

`file:la'

> This hunk appears after line l of file file, and contains no lines in that file. To edit this file to yield the other files, one must append hunk lines taken from the other files. For example, `1:11a' means that the hunk follows line 11 in the first file and contains no lines from that file.

`file:rc'

> This hunk contains the lines in the range r of file file. The range r is a comma-separated pair of line numbers, or just one number if the range is a singleton. To edit this file to yield the other files, one must change the specified lines to be the lines taken from the other files. For example, `2:11,13c' means that the hunk contains lines 11 through 13 from the second file.

If the last line in a set of input lines is incomplete (see section Incomplete Lines), it is distinguished on output from a full line by a following line that starts with `\'.

`diff3` Hunks

Groups of lines that differ in two or three of the input files are called diff3 hunks, by analogy with `diff` hunks (see section Hunks). If all three input files differ in a `diff3` hunk, the hunk is called a three-way hunk; if just two input files differ, it is a two-way hunk.

As with `diff`, several solutions are possible. When comparing the files `A', `B', and `C', `diff3` normally finds `diff3` hunks by merging the two-way hunks output by the two commands `diff A B' and `diff A C'. This does not necessarily minimize the size of the output, but exceptions should be rare.

For example, suppose `F' contains the three lines `a', `b', `f', `G' contains the lines `g', `b', `g', and `H' contains the lines `a', `b', `h'. `diff3 F G H' might output the following:

```
====2
1:1c
3:1c
  a
2:1c
  g
====
1:3c
  f
2:3c
  g
3:3c
  h
```

because it found a two-way hunk containing `a' in the first and third files and `g' in the second file, then the single line `b' common to all three files, then a three-way hunk containing the last line of each file.

An Example of `diff3` Normal Format

Here is the output of the command `diff3 lao tzu tao' (see section A Third Sample Input File, for the complete contents of the files). Notice that it shows only the lines that are different among the three files.

```
====2
1:1,2c
3:1,2c
  The Way that can be told of is not the eternal Way;
  The name that can be named is not the eternal name.
2:0a
====1
1:4c
  The Named is the mother of all things.
2:2,3c
```


```
3:4,5c
  The named is the mother of all things.

====3
1:8c
2:7c
    so we may see their outcome.
3:9c
    so we may see their result.
====
1:11a
2:11,13c
  They both may be called deep and profound.
  Deeper and more profound,
  The door of all subtleties!
3:13,14c

    -- The Way of Lao-Tzu, tr. Wing-tsit Chan
```

Merging From a Common Ancestor

When two people have made changes to copies of the same file, `diff3` can produce a merged output that contains both sets of changes together with warnings about conflicts.

One might imagine programs with names like `diff4` and `diff5` to compare more than three files simultaneously, but in practice the need rarely arises. You can use `diff3` to merge three or more sets of changes to a file by merging two change sets at a time.

`diff3` can incorporate changes from two modified versions into a common preceding version. This lets you merge the sets of changes represented by the two newer files. Specify the common ancestor version as the second argument and the two newer versions as the first and third arguments, like this:

```
diff3 mine older yours
```

You can remember the order of the arguments by noting that they are in alphabetical order.

You can think of this as subtracting older from yours and adding the result to mine, or as merging into mine the changes that would turn older into yours. This merging is well-defined as long as mine and older match in the neighborhood of each such change. This fails to be true when all three input files differ or when only older differs; we call this a conflict. When all three input files differ, we call the conflict an overlap.

`diff3` gives you several ways to handle overlaps and conflicts. You can omit overlaps or conflicts, or select only overlaps, or mark conflicts with special `<<<<<<<' and `>>>>>>>' lines.

`diff3` can output the merge results as an `ed` script that that can be applied to the first file to yield the merged output. However, it is usually better to have `diff3` generate the merged output directly; this bypasses some problems with `ed`.

Selecting Which Changes to Incorporate

You can select all unmerged changes from older to yours for merging into mine with the `-e' or `--ed' option. You can select only the nonoverlapping unmerged changes with `-3' or `--easy-only', and you can select only the overlapping changes with `-x' or `--overlap-only'.

The `-e', `-3' and `-x' options select only unmerged changes, i.e. changes where mine and yours differ; they ignore changes from older to yours where mine and yours are identical, because they assume that such changes have already been merged. If this assumption is not a safe one, you can use the `-A' or `--show-all' option (see section Marking Conflicts).

Here is the output of the command `diff3` with each of these three options (see section A Third Sample Input File, for the complete contents of the files). Notice that `-e' outputs the union of the disjoint sets of changes output by `-3' and `-x'.

Output of `diff3 -e lao tzu tao':

```
11a

  -- The Way of Lao-Tzu, tr. Wing-tsit Chan
.
8c
  so we may see their result.
.
```

Output of `diff3 -3 lao tzu tao':

```
8c
  so we may see their result.
.
```

Output of `diff3 -x lao tzu tao':

```
11a

  -- The Way of Lao-Tzu, tr. Wing-tsit Chan
.
```

Marking Conflicts

`diff3` can mark conflicts in the merged output by bracketing them with special marker lines. A conflict that comes from two files A and B is marked as follows:

```
<<<<<<< A
lines from A
```


```
=======
lines from B
>>>>>>> B
```

A conflict that comes from three files A, B and C is marked as follows:

```
<<<<<<< A
lines from A
||||||| B
lines from B
=======
lines from C
>>>>>>> C
```

The `-A' or `--show-all' option acts like the `-e' option, except that it brackets conflicts, and it outputs all changes from older to yours, not just the unmerged changes. Thus, given the sample input files (see section A Third Sample Input File), `diff3 -A lao tzu tao' puts brackets around the conflict where only `tzu' differs:

```
<<<<<<< tzu
=======
The Way that can be told of is not the eternal Way;
The name that can be named is not the eternal name.
>>>>>>> tao
```

And it outputs the three-way conflict as follows:

```
<<<<<<< lao
||||||| tzu
They both may be called deep and profound.
Deeper and more profound,
The door of all subtleties!
=======

  -- The Way of Lao-Tzu, tr. Wing-tsit Chan
>>>>>>> tao
```

The `-E' or `--show-overlap' option outputs less information than the `-A' or `--show-all' option, because it outputs only unmerged changes, and it never outputs the contents of the second file. Thus the `-E' option acts like the `-e' option, except that it brackets the first and third files from three-way overlapping changes. Similarly, `-X' acts like `-x', except it brackets all its (necessarily overlapping) changes. For example, for the three-way overlapping change above, the `-E' and `-X' options output the following:

```
<<<<<<< lao
=======

  -- The Way of Lao-Tzu, tr. Wing-tsit Chan
```


```
>>>>>>> tao
```

If you are comparing files that have meaningless or uninformative names, you can use the `-L label' or `--label=label' option to show alternate names in the `<<<<<<<', `|||||||' and `>>>>>>>' brackets. This option can be given up to three times, once for each input file. Thus `diff3 -A -L X -L Y -L Z A B C' acts like `diff3 -A A B C', except that the output looks like it came from files named `X', `Y' and `Z' rather than from files named `A', `B' and `C'.

Generating the Merged Output Directly

With the `-m' or `--merge' option, `diff3` outputs the merged file directly. This is more efficient than using `ed` to generate it, and works even with non-text files that `ed` would reject. If you specify `-m' without an `ed` script option, `-A' (`--show-all') is assumed.

For example, the command `diff3 -m lao tzu tao' (see section A Third Sample Input File for a copy of the input files) would output the following:

```
<<<<<<< tzu
=======
The Way that can be told of is not the eternal Way;
The name that can be named is not the eternal name.
>>>>>>> tao
The Nameless is the origin of Heaven and Earth;
The Named is the mother of all things.
Therefore let there always be non-being,
  so we may see their subtlety,
And let there always be being,
  so we may see their result.
The two are the same,
But after they are produced,
  they have different names.
<<<<<<< lao
||||||| tzu
They both may be called deep and profound.
Deeper and more profound,
The door of all subtleties!
=======

  -- The Way of Lao-Tzu, tr. Wing-tsit Chan
>>>>>>> tao
```

How `diff3` Merges Incomplete Lines

With `-m', incomplete lines (see section Incomplete Lines) are simply copied to the output as they are found; if the merged output ends in an conflict and one of the input files ends in an incomplete line,

succeeding `|||||||', `=======' or `>>>>>>>' brackets appear somewhere other than the start of a line because they are appended to the incomplete line.

Without `-m', if an `ed` script option is specified and an incomplete line is found, `diff3` generates a warning and acts as if a newline had been present.

Saving the Changed File

Traditional Unix `diff3` generates an `ed` script without the trailing `w' and and `q' commands that save the changes. System V `diff3` generates these extra commands. GNU `diff3` normally behaves like traditional Unix `diff3`, but with the `-i' option it behaves like System V `diff3` and appends the `w' and `q' commands.

The `-i' option requires one of the `ed` script options `-AeExX3', and is incompatible with the merged output option `-m'.

Interactive Merging with `sdiff`

With `sdiff`, you can merge two files interactively based on a side-by-side `-y' format comparison (see section Showing Differences Side by Side). Use `-o file' or `--output=file' to specify where to put the merged text. See section Invoking `sdiff`, for more details on the options to `sdiff`.

Another way to merge files interactively is to use the Emacs Lisp package `emerge`. See section 'emerge' in The GNU Emacs Manual, for more information.

Specifying `diff` Options to `sdiff`

The following `sdiff` options have the same meaning as for `diff`. See section Options to `diff`, for the use of these options.

```
-a -b -d -i -t -v
-B -H -I regexp

--ignore-blank-lines  --ignore-case
--ignore-matching-lines=regexp  --ignore-space-change
--left-column  --minimal  --speed-large-files
--suppress-common-lines  --expand-tabs
--text  --version  --width=columns
```

For historical reasons, `sdiff` has alternate names for some options. The `-l' option is equivalent to the `--left-column' option, and similarly `-s' is equivalent to `--suppress-common-lines'. The meaning of the `sdiff` `-w' and `-W' options is interchanged from that of `diff`: with `sdiff`, `-w columns' is equivalent to `--width=columns', and `-W' is equivalent to `--ignore-all-space'. `sdiff` without the `-o' option is equivalent to `diff` with the `-y' or `--side-by-side' option (see section Showing Differences Side by Side).

Merge Commands

Groups of common lines, with a blank gutter, are copied from the first file to the output. After each group of differing lines, `sdiff` prompts with `%' and pauses, waiting for one of the following commands. Follow each command with RET.

`e'

Discard both versions. Invoke a text editor on an empty temporary file, then copy the resulting file to the output.

`eb'

Concatenate the two versions, edit the result in a temporary file, then copy the edited result to the output.

`el'

Edit a copy of the left version, then copy the result to the output.

`er'

Edit a copy of the right version, then copy the result to the output.

`l'

Copy the left version to the output.

`q'

Quit.

`r'

Copy the right version to the output.

`s'

Silently copy common lines.

`v'

Verbosely copy common lines. This is the default.

The text editor invoked is specified by the `EDITOR` environment variable if it is set. The default is system-dependent.

Merging with `patch`

`patch` takes comparison output produced by `diff` and applies the differences to a copy of the original file, producing a patched version. With `patch`, you can distribute just the changes to a set of files instead of distributing the entire file set; your correspondents can apply `patch` to update their copy of the files with your changes. `patch` automatically determines the diff format, skips any leading or trailing headers, and uses the headers to determine which file to patch. This lets your correspondents feed an article or message containing a difference listing directly to `patch`.

`patch` detects and warns about common problems like forward patches. It saves the original version of the files it patches, and saves any patches that it could not apply. It can also maintain a `patchlevel.h` file

to ensures that your correspondents apply diffs in the proper order.

`patch` accepts a series of diffs in its standard input, usually separated by headers that specify which file to patch. It applies `diff` hunks (see section Hunks) one by one. If a hunk does not exactly match the original file, `patch` uses heuristics to try to patch the file as well as it can. If no approximate match can be found, `patch` rejects the hunk and skips to the next hunk. `patch` normally replaces each file f with its new version, saving the original file in `f.orig', and putting reject hunks (if any) into `f.rej'.

See section Invoking `patch`, for detailed information on the options to `patch`. See section Backup File Names, for more information on how `patch` names backup files. See section Reject File Names, for more information on where `patch` puts reject hunks.

Selecting the `patch` Input Format

`patch` normally determines which `diff` format the patch file uses by examining its contents. For patch files that contain particularly confusing leading text, you might need to use one of the following options to force `patch` to interpret the patch file as a certain format of diff. The output formats listed here are the only ones that `patch` can understand.

`-c'

`--context'

context diff.

`-e'

`--ed'

`ed` script.

`-n'

`--normal'

normal diff.

`-u'

`--unified'

unified diff.

Applying Imperfect Patches

`patch` tries to skip any leading text in the patch file, apply the diff, and then skip any trailing text. Thus you can feed a news article or mail message directly to `patch`, and it should work. If the entire diff is indented by a constant amount of whitespace, `patch` automatically ignores the indentation.

However, certain other types of imperfect input require user intervention.

Applying Patches with Changed Whitespace

Sometimes mailers, editors, or other programs change spaces into tabs, or vice versa. If this happens to a patch file or an input file, the files might look the same, but `patch` will not be able to match them properly. If this problem occurs, use the `-l' or `--ignore-whitespace' option, which makes `patch` compare whitespace loosely so that any sequence of whitespace in the patch file matches any sequence of whitespace in the input files. Non-whitespace characters must still match exactly. Each line of the context must still match a line in the input file.

Applying Reversed Patches

Sometimes people run `diff` with the new file first instead of second. This creates a diff that is "reversed". To apply such patches, give `patch` the `-R' or `--reverse' option. `patch` then attempts to swap each hunk around before applying it. Rejects come out in the swapped format. The `-R' option does not work with `ed` scripts because there is too little information in them to reconstruct the reverse operation.

Often `patch` can guess that the patch is reversed. If the first hunk of a patch fails, `patch` reverses the hunk to see if it can apply it that way. If it can, `patch` asks you if you want to have the `-R' option set; if it can't, `patch` continues to apply the patch normally. This method cannot detect a reversed patch if it is a normal diff and the first command is an append (which should have been a delete) since appends always succeed, because a null context matches anywhere. But most patches add or change lines rather than delete them, so most reversed normal diffs begin with a delete, which fails, and `patch` notices.

If you apply a patch that you have already applied, `patch` thinks it is a reversed patch and offers to un-apply the patch. This could be construed as a feature. If you did this inadvertently and you don't want to un-apply the patch, just answer `n' to this offer and to the subsequent "apply anyway" question--or type C-c to kill the `patch` process.

Helping `patch` Find Inexact Matches

For context diffs, and to a lesser extent normal diffs, `patch` can detect when the line numbers mentioned in the patch are incorrect, and it attempts to find the correct place to apply each hunk of the patch. As a first guess, it takes the line number mentioned in the hunk, plus or minus any offset used in applying the previous hunk. If that is not the correct place, `patch` scans both forward and backward for a set of lines matching the context given in the hunk.

First `patch` looks for a place where all lines of the context match. If it cannot find such a place, and it is reading a context or unified diff, and the maximum fuzz factor is set to 1 or more, then `patch` makes another scan, ignoring the first and last line of context. If that fails, and the maximum fuzz factor is set to 2 or more, it makes another scan, ignoring the first two and last two lines of context are ignored. It continues similarly if the maximum fuzz factor is larger.

The `-F lines' or `--fuzz=lines' option sets the maximum fuzz factor to lines. This option only applies to context and unified diffs; it ignores up to lines lines while looking for the place to install a hunk. Note that a larger fuzz factor increases the odds of making a faulty patch. The default fuzz factor is 2; it may not be set to more than the number of lines of context in the diff, ordinarily 3.

If `patch` cannot find a place to install a hunk of the patch, it writes the hunk out to a reject file (see section

Reject File Names, for information on how reject files are named). It writes out rejected hunks in context format no matter what form the input patch is in. If the input is a normal or `ed` diff, many of the contexts are simply null. The line numbers on the hunks in the reject file may be different from those in the patch file: they show the approximate location where `patch` thinks the failed hunks belong in the new file rather than in the old one.

As it completes each hunk, `patch` tells you whether the hunk succeeded or failed, and if it failed, on which line (in the new file) `patch` thinks the hunk should go. If this is different from the line number specified in the diff, it tells you the offset. A single large offset *may* indicate that `patch` installed a hunk in the wrong place. `patch` also tells you if it used a fuzz factor to make the match, in which case you should also be slightly suspicious.

`patch` cannot tell if the line numbers are off in an `ed` script, and can only detect wrong line numbers in a normal diff when it finds a change or delete command. It may have the same problem with a context diff using a fuzz factor equal to or greater than the number of lines of context shown in the diff (typically 3). In these cases, you should probably look at a context diff between your original and patched input files to see if the changes make sense. Compiling without errors is a pretty good indication that the patch worked, but not a guarantee.

`patch` usually produces the correct results, even when it must make many guesses. However, the results are guaranteed only when the patch is applied to an exact copy of the file that the patch was generated from.

Removing Empty Files

Sometimes when comparing two directories, the first directory contains a file that the second directory does not. If you give `diff` the `-N' or `--new-file' option, it outputs a diff that deletes the contents of this file. By default, `patch` leaves an empty file after applying such a diff. The `-E' or `--remove-empty-files' option to `patch` deletes output files that are empty after applying the diff.

Multiple Patches in a File

If the patch file contains more than one patch, `patch` tries to apply each of them as if they came from separate patch files. This means that it determines the name of the file to patch for each patch, and that it examines the leading text before each patch for file names and prerequisite revision level (see section Tips for Making Patch Distributions, for more on that topic).

For the second and subsequent patches in the patch file, you can give options and another original file name by separating their argument lists with a `+'. However, the argument list for a second or subsequent patch may not specify a new patch file, since that does not make sense.

For example, to tell `patch` to strip the first three slashes from the name of the first patch in the patch file and none from subsequent patches, and to use `` `code.c' `` as the first input file, you can use:

```
patch -p3 code.c + -p0 < patchfile
```

The `-S' or `--skip' option ignores the current patch from the patch file, but continue looking for the next

patch in the file. Thus, to ignore the first and third patches in the patch file, you can use:

```
patch -S + + -S + < patch file
```

Messages and Questions from `patch`

`patch` can produce a variety of messages, especially if it has trouble decoding its input. In a few situations where it's not sure how to proceed, `patch` normally prompts you for more information from the keyboard. There are options to suppress printing non-fatal messages and stopping for keyboard input.

The message `Hmm...' indicates that `patch` is reading text in the patch file, attempting to determine whether there is a patch in that text, and if so, what kind of patch it is.

You can inhibit all terminal output from `patch`, unless an error occurs, by using the `-s', `--quiet', or `--silent' option.

There are two ways you can prevent `patch` from asking you any questions. The `-f' or `--force' option assumes that you know what you are doing. It assumes the following:

- skip patches for which it can't find a file to patch;
- patch files even though they have the wrong version for the `Prereq:' line in the patch;
- assume that patches are not reversed even if they look like they are.

The `-t' or `--batch' option is similar to `-f', in that it suppresses questions, but it makes somewhat different assumptions:

- skip patches for which it can't find a file to patch (the same as `-f');
- skip patches for which the file has the wrong version for the `Prereq:' line in the patch;
- assume that patches are reversed if they look like they are.

`patch` exits with a non-zero status if it creates any reject files. When applying a set of patches in a loop, you should check the exit status, so you don't apply a later patch to a partially patched file.

Tips for Making Patch Distributions

Here are some things you should keep in mind if you are going to distribute patches for updating a software package.

Make sure you have specified the file names correctly, either in a context diff header or with an `Index:' line. If you are patching files in a subdirectory, be sure to tell the patch user to specify a `-p' or `--strip' option as needed. Take care to not send out reversed patches, since these make people wonder whether they have already applied the patch.

To save people from partially applying a patch before other patches that should have gone before it, you can make the first patch in the patch file update a file with a name like `patchlevel.h' or `version.c', which contains a patch level or version number. If the input file contains the wrong version number, `patch` will complain immediately.

An even clearer way to prevent this problem is to put a `Prereq:' line before the patch. If the leading text of the patch file contains a line that starts with `Prereq:', `patch` takes the next word from that line (normally a version number) and checks whether the next input file contains that word, preceded and followed by either whitespace or a newline. If not, `patch` prompts you for confirmation before proceeding. This makes it difficult to accidentally apply patches in the wrong order.

Since `patch` does not handle incomplete lines properly, make sure that all the source files in your program end with a newline whenever you release a version.

To create a patch that changes an older version of a package into a newer version, first make a copy of the older version in a scratch directory. Typically you do that by unpacking a `tar` or `shar` archive of the older version.

You might be able to reduce the size of the patch by renaming or removing some files before making the patch. If the older version of the package contains any files that the newer version does not, or if any files have been renamed between the two versions, make a list of `rm` and `mv` commands for the user to execute in the old version directory before applying the patch. Then run those commands yourself in the scratch directory.

If there are any files that you don't need to include in the patch because they can easily be rebuilt from other files (for example, `` `TAGS' `` and output from `yacc` and `makeinfo`), replace the versions in the scratch directory with the newer versions, using `rm` and `ln` or `cp`.

Now you can create the patch. The de-facto standard `diff` format for patch distributions is context format with two lines of context, produced by giving `diff` the `-C 2' option. Give `diff` the `-N' option in case the newer version of the package contains any files that the older one does not. Make sure to specify the scratch directory first and the newer directory second.

Add to the top of the patch a note telling the user any `rm` and `mv` commands to run before applying the patch. Then you can remove the scratch directory.

Invoking cmp

The `cmp` command compares two files, and if they differ, tells the first byte and line number where they differ. Its arguments are as follows:

```
cmp options... from-file [to-file]
```

The file name `-' is always the standard input. `cmp` also uses the standard input if one file name is omitted.

An exit status of 0 means no differences were found, 1 means some differences were found, and 2 means trouble.

Options to `cmp`

Below is a summary of all of the options that GNU `cmp` accepts. Most options have two equivalent names, one of which is a single letter preceded by `-', and the other of which is a long name preceded by `--'. Multiple single letter options (unless they take an argument) can be combined into a single command line word: `-cl' is equivalent to `-c -l'.

`-c'

Print the differing characters. Display control characters as a `^' followed by a letter of the alphabet and precede characters that have the high bit set with `M-' (which stands for "meta").

`-l'

Print the (decimal) offsets and (octal) values of all differing bytes.

`--print-chars'

Print the differing characters. Display control characters as a `^' followed by a letter of the alphabet and precede characters that have the high bit set with `M-' (which stands for "meta").

`--quiet'

`-s'

`--silent'

Do not print anything; only return an exit status indicating whether the files differ.

`--verbose'

Print the (decimal) offsets and (octal) values of all differing bytes.

Invoking `diff`

The format for running the `diff` command is:

```
diff options... from-file to-file
```

In the simplest case, `diff` compares the contents of the two files from-file and to-file. A file name of `-' stands for text read from the standard input. As a special case, `diff - -' compares a copy of standard input to itself.

If from-file is a directory and to-file is not, `diff` compares the file in from-file whose file name is that of to-file, and vice versa. The non-directory file must not be `-'.

If both from-file and to-file are directories, `diff` compares corresponding files in both directories, in alphabetical order; this comparison is not recursive unless the `-r' or `--recursive' option is given. `diff` never compares the actual contents of a directory as if it were a file. The file that is fully specified may not be standard input, because standard input is nameless and the notion of "file with the same name" does not apply.

`diff` options begin with `-', so normally from-file and to-file may not begin with `-'. However, `--' as an argument by itself treats the remaining arguments as file names even if they begin with `-'.

An exit status of 0 means no differences were found, 1 means some differences were found, and 2 mea trouble.

Options to `diff`

Below is a summary of all of the options that GNU `diff` accepts. Most options have two equivalent names, one of which is a single letter preceded by `-', and the other of which is a long name preceded by `--'. Multiple single letter options (unless they take an argument) can be combined into a single command line word: `-ac' is equivalent to `-a -c'. Long named options can be abbreviated to any unique prefix of their name. Brackets ([and]) indicate that an option takes an optional argument.

`-lines'

Show lines (an integer) lines of context. This option does not specify an output format by itself; it has no effect unless it is combined with `-c' (see section Context Format) or `-u' (see section Unified Format). This option is obsolete.

`-a'

Treat all files as text and compare them line-by-line, even if they do not seem to be text. See section Binary Files and Forcing Text Comparisons.

`-b'

Ignore changes in amount of blank and tab whitespace. See section Suppressing Differences in Blank and Tab Spacing.

`-B'

Ignore changes that just insert or delete blank lines. See section Suppressing Differences in Blank Lines.

`--brief'

Report only whether the files differ, not the details of the differences. See section Summarizing Which Files Differ.

`-c'

Use the context output format. See section Context Format.

`-C lines'

`--context[=lines]'

Use the context output format, showing lines (an integer) lines of context, or three if lines is not given. See section Context Format.

`--changed-group-format=format'

Use format to output a line group containing differing lines from both files in if-then-else format. See section Line Group Formats.

`-d'

Change the algorithm perhaps find a smaller set of changes. This makes `diff` slower (sometimes much slower). See section `diff` Performance Tradeoffs.

`-D name'

Make merged `#ifdef' format output, conditional on the preprocessor macro name. See section Merging Files with If-then-else.

`-e'

`--ed'

Make output that is a valid `ed` script. See section `ed` Scripts.

`--exclude=pattern'

When comparing directories, ignore files and subdirectories whose basenames match pattern. See section Comparing Directories.

`--exclude-from=file'

When comparing directories, ignore files and subdirectories whose basenames match any pattern contained in file. See section Comparing Directories.

`--expand-tabs'

Expand tabs to spaces in the output, to preserve the alignment of tabs in the input files. See section Preserving Tabstop Alignment.

`-f'

Make output that looks vaguely like an `ed` script but has changes in the order they appear in the file. See section Forward `ed` Scripts.

`-F regexp'

In context and unified format, for each hunk of differences, show some of the last preceding line that matches regexp. See section Showing Lines That Match Regular Expressions.

`--forward-ed'

Make output that looks vaguely like an `ed` script but has changes in the order they appear in the file. See section Forward `ed` Scripts.

`-h'

This option currently has no effect; it is present for Unix compatibility.

`-H'

Use heuristics to speed handling of large files that have numerous scattered small changes. See section `diff` Performance Tradeoffs.

`-i'

Ignore changes in case; consider upper- and lower-case letters equivalent. See section Suppressing Case Differences.

`-I regexp'

Ignore changes that just insert or delete lines that match regexp. See section Suppressing Lines Matching a Regular Expression.

`--ifdef=name'

Make merged if-then-else output using format. See section Merging Files with If-then-else.

`--ignore-all-space'

Ignore whitespace when comparing lines. See section Suppressing Differences in Blank and Tab

Spacing.

`--ignore-blank-lines'

Ignore changes that just insert or delete blank lines. See section Suppressing Differences in Blank Lines.

`--ignore-case'

Ignore changes in case; consider upper- and lower-case to be the same. See section Suppressing Case Differences.

`--ignore-matching-lines=regexp'

Ignore changes that just insert or delete lines that match regexp. See section Suppressing Lines Matching a Regular Expression.

`--ignore-space-change'

Ignore changes in amount of blank and tab whitespace. See section Suppressing Differences in Blank and Tab Spacing.

`--initial-tab'

Output a tab rather than a space before the text of a line in normal or context format. This causes the alignment of tabs in the line to look normal. See section Preserving Tabstop Alignment.

`-l'

Pass the output through `pr` to paginate it. See section Paginating `diff` Output.

`-L label'

Use label instead of the file name in the context format (see section Context Format) and unified format (see section Unified Format) headers. See section RCS Scripts.

`--label=label'

Use label instead of the file name in the context format (see section Context Format) and unified format (see section Unified Format) headers.

`--left-column'

Print only the left column of two common lines in side by side format. See section Controlling Side by Side Format.

`--minimal'

Change the algorithm to perhaps find a smaller set of changes. This makes `diff` slower (sometimes much slower). See section `diff` Performance Tradeoffs.

`-n'

Output RCS-format diffs; like `-f' except that each command specifies the number of lines affected. See section RCS Scripts.

`-N'

`--new-file'

In directory comparison, if a file is found in only one directory, treat it as present but empty in the other directory. See section Comparing Directories.

`--new-group-format=format'

Use format to output a group of lines taken from just the second file in if-then-else format. See section Line Group Formats.

`--new-line-format=format'

Use format to output a line taken from just the second file in if-then-else format. See section Line Formats.

`--old-group-format=format'

Use format to output a group of lines taken from just the first file in if-then-else format. See section Line Group Formats.

`--old-line-format=format'

Use format to output a line taken from just the first file in if-then-else format. See section Line Formats.

`-p'

Show which C function each change is in. See section Showing C Function Headings.

`-P'

When comparing directories, if a file appears only in the second directory of the two, treat it as present but empty in the other.

`--paginate'

Pass the output through `pr` to paginate it. See section Paginating `diff` Output.

`-q'

Report only whether the files differ, not the details of the differences. See section Summarizing Which Files Differ.

`-r'

When comparing directories, recursively compare any subdirectories found. See section Comparing Directories.

`--rcs'

Output RCS-format diffs; like `-f' except that each command specifies the number of lines affected. See section RCS Scripts.

`--recursive'

When comparing directories, recursively compare any subdirectories found. See section Comparing Directories.

`--report-identical-files'

Report when two files are the same. See section Comparing Directories.

`-s'

Report when two files are the same. See section Comparing Directories.

`-S file'

When comparing directories, start with the file file. This is used for resuming an aborted comparison.

See section Comparing Directories.

`--sdiff-merge-assist'

Print extra information to help `sdiff`. `sdiff` uses this option when it runs `diff`. This option is not intended for users to use directly.

`--show-c-function'

Show which C function each change is in. See section Showing C Function Headings.

`--show-function-line=regexp'

In context and unified format, for each hunk of differences, show some of the last preceding line that matches regexp. See section Showing Lines That Match Regular Expressions.

`--side-by-side'

Use the side by side output format. See section Controlling Side by Side Format.

`--speed-large-files'

Use heuristics to speed handling of large files that have numerous scattered small changes. See section `diff` Performance Tradeoffs.

`--starting-file=file'

When comparing directories, start with the file file. This is used for resuming an aborted comparison. See section Comparing Directories.

`--suppress-common-lines'

Do not print common lines in side by side format. See section Controlling Side by Side Format.

`-t'

Expand tabs to spaces in the output, to preserve the alignment of tabs in the input files. See section Preserving Tabstop Alignment.

`-T'

Output a tab rather than a space before the text of a line in normal or context format. This causes the alignment of tabs in the line to look normal. See section Preserving Tabstop Alignment.

`--text'

Treat all files as text and compare them line-by-line, even if they do not appear to be text. See section Binary Files and Forcing Text Comparisons.

`-u'

Use the unified output format. See section Unified Format.

`--unchanged-group-format=format'

Use format to output a group of common lines taken from both files in if-then-else format. See section Line Group Formats.

`--unchanged-line-format=format'

Use format to output a line common to both files in if-then-else format. See section Line Formats.

`--unidirectional-new-file'

When comparing directories, if a file appears only in the second directory of the two, treat it as present but empty in the other. See section Comparing Directories.

`-U lines'

`--unified[=lines]'

Use the unified output format, showing lines (an integer) lines of context, or three if lines is not given. See section Unified Format.

`-v'

`--version'

Output the version number of `diff`.

`-w'

Ignore horizontal whitespace when comparing lines. See section Suppressing Differences in Blank and Tab Spacing.

`-W columns'

`--width=columns'

Use an output width of columns in side by side format. See section Controlling Side by Side Format.

`-x pattern'

When comparing directories, ignore files and subdirectories whose basenames match pattern. See section Comparing Directories.

`-X file'

When comparing directories, ignore files and subdirectories whose basenames match any pattern contained in file. See section Comparing Directories.

`-y'

Use the side by side output format. See section Controlling Side by Side Format.

Invoking `diff3`

The `diff3` command compares three files and outputs descriptions of their differences. Its arguments are as follows:

```
diff3 options... mine older yours
```

The files to compare are mine, older, and yours. At most one of these three file names may be `-', which tells `diff3` to read the standard input for that file.

An exit status of 0 means `diff3` was successful, 1 means some conflicts were found, and 2 means trouble.

Options to `diff3`

Below is a summary of all of the options that GNU `diff3` accepts. Multiple single letter options (unless they take an argument) can be combined into a single command line argument.

`-a'

Treat all files as text and compare them line-by-line, even if they do not appear to be text. See section Binary Files and Forcing Text Comparisons.

`-A'

Incorporate all changes from older to yours into mine, surrounding all conflicts with bracket lines. See section Marking Conflicts.

`-e'

Generate an `ed` script that incorporates all the changes from older to yours into mine. See section Selecting Which Changes to Incorporate.

`-E'

Like `-e', except bracket lines from overlapping changes' first and third files. See section Marking Conflicts. With `-e', an overlapping change looks like this:

```
<<<<<<< mine
lines from mine
=======
lines from yours
>>>>>>> yours
```

`--ed'

Generate an `ed` script that incorporates all the changes from older to yours into mine. See section Selecting Which Changes to Incorporate.

`--easy-only'

Like `-e', except output only the nonoverlapping changes. See section Selecting Which Changes to Incorporate.

`-i'

Generate `w' and `q' commands at the end of the `ed` script for System V compatibility. This option must be combined with one of the `-AeExX3' options, and may not be combined with `-m'. See section Saving the Changed File.

`-L label'

`--label=label'

Use the label label for the brackets output by the `-A', `-E' and `-X' options. This option may be given up to three times, one for each input file. The default labels are the names of the input files. Thus `diff3 -L X -L Y -L Z -m A B C' acts like `diff3 -m A B C', except that the output looks like it came from files named `X', `Y' and `Z' rather than from files named `A', `B' and `C'. See section Marking Conflicts.

`-m'

`--merge'

Apply the edit script to the first file and send the result to standard output. Unlike piping the output from `diff3` to `ed`, this works even for binary files and incomplete lines. `-A' is assumed if no edit script option is specified. See section Generating the Merged Output Directly.

`--overlap-only'

Like `-e', except output only the overlapping changes. See section Selecting Which Changes to Incorporate.

`--show-all'

Incorporate all unmerged changes from older to yours into mine, surrounding all overlapping changes with bracket lines. See section Marking Conflicts.

`--show-overlap'

Like `-e', except bracket lines from overlapping changes' first and third files. See section Marking Conflicts.

`--text'

Treat all files as text and compare them line-by-line, even if they do not appear to be text. See section Binary Files and Forcing Text Comparisons.

`-v'

`--version'

Output the version number of `diff3`.

`-x'

Like `-e', except output only the overlapping changes. See section Selecting Which Changes to Incorporate.

`-X'

Like `-E', except output only the overlapping changes. In other words, like `-x', except bracket changes as in `-E'. See section Marking Conflicts.

`-3'

Like `-e', except output only the nonoverlapping changes. See section Selecting Which Changes to Incorporate.

Invoking `patch`

Normally `patch` is invoked like this:

```
patch <patchfile
```

The full format for invoking `patch` is:

```
patch options... [origfile [patchfile]] [+ options... [origfile]]...
```

If you do not specify patchfile, or if patchfile is `-', `patch` reads the patch (that is, the `diff` output) from the standard input.

You can specify one or more of the original files as orig arguments; each one and options for interpreting it is separated from the others with a `+'. See section Multiple Patches in a File, for more information.

If you do not specify an input file on the command line, `patch` tries to figure out from the leading text

(any text in the patch that comes before the `diff` output) which file to edit. In the header of a context or unified diff, `patch` looks in lines beginning with `***', `---', or `+++'; among those, it chooses the shortest name of an existing file. Otherwise, if there is an `Index:' line in the leading text, `patch` tries to use the file name from that line. If `patch` cannot figure out the name of an existing file from the leading text, it prompts you for the name of the file to patch.

If the input file does not exist or is read-only, and a suitable RCS or SCCS file exists, `patch` attempts to check out or get the file before proceeding.

By default, `patch` replaces the original input file with the patched version, after renaming the original file into a backup file (see section Backup File Names, for a description of how `patch` names backup files). You can also specify where to put the output with the `-o output-file' or `--output=output-file' option.

Applying Patches in Other Directories

The `-d directory' or `--directory=directory' option to `patch` makes directory directory the current directory for interpreting both file names in the patch file, and file names given as arguments to other options (such as `-B' and `-o'). For example, while in a news reading program, you can patch a file in the `/usr/src/emacs' directory directly from the article containing the patch like this:

```
| patch -d /usr/src/emacs
```

Sometimes the file names given in a patch contain leading directories, but you keep your files in a directory different from the one given in the patch. In those cases, you can use the `-p[number]' or `--strip[=number]' option to set the file name strip count to number. The strip count tells `patch` how many slashes, along with the directory names between them, to strip from the front of file names. `-p' with no number given is equivalent to `-p0'. By default, `patch` strips off all leading paths, leaving just the base file names, except that when a file name given in the patch is a relative path and all of its leading directories already exist, `patch` does not strip off the leading path. (A relative path is one that does not start with a slash.)

`patch` looks for each file (after any slashes have been stripped) in the current directory, or if you used the `-d directory' option, in that directory.

For example, suppose the file name in the patch file is `/gnu/src/emacs/etc/NEWS'. Using `-p' or `-p0' gives the entire file name unmodified, `-p1' gives `gnu/src/emacs/etc/NEWS' (no leading slash), `-p4' gives `etc/NEWS', and not specifying `-p' at all gives `NEWS'.

Backup File Names

Normally, `patch` renames an original input file into a backup file by appending to its name the extension `.orig', or `~' on systems that do not support long file names. The `-b backup-suffix' or `--suffix=backup-suffix' option uses backup-suffix as the backup extension instead.

Alternately, you can specify the extension for backup files with the `SIMPLE_BACKUP_SUFFIX` environment variable, which the options override.

`patch` can also create numbered backup files the way GNU Emacs does. With this method, instead of

having a single backup of each file, `patch` makes a new backup file name each time it patches a file. I example, the backups of a file named `` `sink' `` would be called, successively, `` `sink.~1~' ``, `` `sink.~2~' ``, `` `sink.~3~' ``, etc.

The `-V backup-style' or `--version-control=backup-style' option takes as an argument a method for creating backup file names. You can alternately control the type of backups that `patch` makes with the `VERSION_CONTROL` environment variable, which the `-V' option overrides. The value of the `VERSION_CONTROL` environment variable and the argument to the `-V' option are like the GNU Emacs `version-control` variable (see section Backup File Names, for more information on backup versions in Emacs). They also recognize synonyms that are more descriptive. The valid values are listed below; unique abbreviations are acceptable.

`t'

`numbered'

Always make numbered backups.

`nil'

`existing'

Make numbered backups of files that already have them, simple backups of the others. This is the default.

`never'

`simple'

Always make simple backups.

Alternately, you can tell `patch` to prepend a prefix, such as a directory name, to produce backup file names. The `-B backup-prefix' or `--prefix=backup-prefix' option makes backup files by prepending backup-prefix to them. If you use this option, `patch` ignores any `-b' option that you give.

If the backup file already exists, `patch` creates a new backup file name by changing the first lowercase letter in the last component of the file name into uppercase. If there are no more lowercase letters in the name, it removes the first character from the name. It repeats this process until it comes up with a backup file name that does not already exist.

If you specify the output file with the `-o' option, that file is the one that is backed up, not the input file.

Reject File Names

The names for reject files (files containing patches that `patch` could not find a place to apply) are normally the name of the output file with `.rej' appended (or `#' on systems that do not support long file names).

Alternatively, you can tell `patch` to place all of the rejected patches in a single file. The `-r reject-file' or `--reject-file=reject-file' option uses reject-file as the reject file name.

Options to `patch`

Here is a summary of all of the options that `patch` accepts. Older versions of `patch` do not accept long-named options or the `-t', `-E', or `-V' options.

Multiple single-letter options that do not take an argument can be combined into a single command line argument (with only one dash). Brackets ([and]) indicate that an option takes an optional argument.

`-b backup-suffix'

Use backup-suffix as the backup extension instead of `.orig' or `~'. See section Backup File Names.

`-B backup-prefix'

Use backup-prefix as a prefix to the backup file name. If this option is specified, any `-b' option is ignored. See section Backup File Names.

`--batch'

Do not ask any questions. See section Messages and Questions from `patch`.

`-c'

`--context'

Interpret the patch file as a context diff. See section Selecting the `patch` Input Format.

`-d directory'

`--directory=directory'

Makes directory directory the current directory for interpreting both file names in the patch file, and file names given as arguments to other options. See section Applying Patches in Other Directories.

`-D name'

Make merged if-then-else output using format. See section Merging Files with If-then-else.

`--debug=number'

Set internal debugging flags. Of interest only to `patch` patchers.

`-e'

`--ed'

Interpret the patch file as an `ed` script. See section Selecting the `patch` Input Format.

`-E'

Remove output files that are empty after the patches have been applied. See section Removing Empty Files.

`-f'

Assume that the user knows exactly what he or she is doing, and do not ask any questions. See section Messages and Questions from `patch`.

`-F lines'

Set the maximum fuzz factor to lines. See section Helping `patch` Find Inexact Matches.

`--force'

Assume that the user knows exactly what he or she is doing, and do not ask any questions. See

section Messages and Questions from `patch`.

`--forward'

Ignore patches that `patch` thinks are reversed or already applied. See also `-R'. See section Applying Reversed Patches.

`--fuzz=lines'

Set the maximum fuzz factor to lines. See section Helping `patch` Find Inexact Matches.

`--ifdef=name'

Make merged if-then-else output using format. See section Merging Files with If-then-else.

`--ignore-whitespace'

`-l'

Let any sequence of whitespace in the patch file match any sequence of whitespace in the input file. See section Applying Patches with Changed Whitespace.

`-n'

`--normal'

Interpret the patch file as a normal diff. See section Selecting the `patch` Input Format.

`-N'

Ignore patches that `patch` thinks are reversed or already applied. See also `-R'. See section Applying Reversed Patches.

`-o output-file'

`--output=output-file'

Use output-file as the output file name. See section Options to `patch`.

`-p[number]'

Set the file name strip count to number. See section Applying Patches in Other Directories.

`--prefix=backup-prefix'

Use backup-prefix as a prefix to the backup file name. If this option is specified, any `-b' option is ignored. See section Backup File Names.

`--quiet'

Work silently unless an error occurs. See section Messages and Questions from `patch`.

`-r reject-file'

Use reject-file as the reject file name. See section Reject File Names.

`-R'

Assume that this patch was created with the old and new files swapped. See section Applying Reversed Patches.

`--reject-file=reject-file'

Use reject-file as the reject file name. See section Reject File Names.

`--remove-empty-files'

Remove output files that are empty after the patches have been applied. See section Removing

Empty Files.

`--reverse'

Assume that this patch was created with the old and new files swapped. See section Applying Reversed Patches.

`-s'

Work silently unless an error occurs. See section Messages and Questions from `patch`.

`-S'

Ignore this patch from the patch file, but continue looking for the next patch in the file. See section Multiple Patches in a File.

`--silent'

Work silently unless an error occurs. See section Messages and Questions from `patch`.

`--skip'

Ignore this patch from the patch file, but continue looking for the next patch in the file. See section Multiple Patches in a File.

`--strip[=number]'

Set the file name strip count to number. See section Applying Patches in Other Directories.

`--suffix=backup-suffix'

Use backup-suffix as the backup extension instead of `.orig' or `~'. See section Backup File Names.

`-t'

Do not ask any questions. See section Messages and Questions from `patch`.

`-u'

`--unified'

Interpret the patch file as a unified diff. See section Selecting the `patch` Input Format.

`-v'

Output the revision header and patch level of `patch`.

`-V backup-style'

Select the kind of backups to make. See section Backup File Names.

`--version'

Output the revision header and patch level of `patch`.

`--version=control=backup-style'

Select the kind of backups to make. See section Backup File Names.

`-x number'

Set internal debugging flags. Of interest only to `patch` patchers.

Invoking `sdiff`

The `sdiff` command merges two files and interactively outputs the results. Its arguments are as follows:

```
sdiff -o outfile options... from-file to-file
```

This merges from-file with to-file, with output to outfile. If from-file is a directory and to-file is not, `sdiff` compares the file in from-file whose file name is that of to-file, and vice versa. from-file and to-file may not both be directories.

`sdiff` options begin with `-', so normally from-file and to-file may not begin with `-'. However, `--' as an argument by itself treats the remaining arguments as file names even if they begin with `-'. You may not use `-' as an input file.

An exit status of 0 means no differences were found, 1 means some differences were found, and 2 means trouble.

`sdiff` without `-o' (or `--output') produces a side-by-side difference. This usage is obsolete; use `diff --side-by-side' instead.

Options to `sdiff`

Below is a summary of all of the options that GNU `sdiff` accepts. Each option has two equivalent names, one of which is a single letter preceded by `-', and the other of which is a long name preceded by `--'. Multiple single letter options (unless they take an argument) can be combined into a single command line argument. Long named options can be abbreviated to any unique prefix of their name.

`-a'

Treat all files as text and compare them line-by-line, even if they do not appear to be text. See section Binary Files and Forcing Text Comparisons.

`-b'

Ignore changes in amount of blank and tab whitespace. See section Suppressing Differences in Blank and Tab Spacing.

`-B'

Ignore changes that just insert or delete blank lines. See section Suppressing Differences in Blank Lines.

`-d'

Change the algorithm to perhaps find a smaller set of changes. This makes `sdiff` slower (sometimes much slower). See section `diff` Performance Tradeoffs.

`-H'

Use heuristics to speed handling of large files that have numerous scattered small changes. See section `diff` Performance Tradeoffs.

`--expand-tabs'

Expand tabs to spaces in the output, to preserve the alignment of tabs in the input files. See section Preserving Tabstop Alignment.

`-i'

Ignore changes in case; consider upper- and lower-case to be the same. See section Suppressing Case Differences.

`-I regexp'

Ignore changes that just insert or delete lines that match regexp. See section Suppressing Lines Matching a Regular Expression.

`--ignore-all-space'

Ignore whitespace when comparing lines. See section Suppressing Differences in Blank and Tab Spacing.

`--ignore-blank-lines'

Ignore changes that just insert or delete blank lines. See section Suppressing Differences in Blank Lines.

`--ignore-case'

Ignore changes in case; consider upper- and lower-case to be the same. See section Suppressing Case Differences.

`--ignore-matching-lines=regexp'

Ignore changes that just insert or delete lines that match regexp. See section Suppressing Lines Matching a Regular Expression.

`--ignore-space-change'

Ignore changes in amount of blank and tab whitespace. See section Suppressing Differences in Blank and Tab Spacing.

`-l'

`--left-column'

Print only the left column of two common lines. See section Controlling Side by Side Format.

`--minimal'

Change the algorithm to perhaps find a smaller set of changes. This makes `sdiff` slower (sometimes much slower). See section `diff` Performance Tradeoffs.

`-o file'

`--output=file'

Put merged output into file. This option is required for merging.

`-s'

`--suppress-common-lines'

Do not print common lines. See section Controlling Side by Side Format.

`--speed-large-files'

Use heuristics to speed handling of large files that have numerous scattered small changes. See

section `diff` Performance Tradeoffs.

`-t'

Expand tabs to spaces in the output, to preserve the alignment of tabs in the input files. See section Preserving Tabstop Alignment.

`--text'

Treat all files as text and compare them line-by-line, even if they do not appear to be text. See section Binary Files and Forcing Text Comparisons.

`-v'

`--version'

Output the version number of `sdiff`.

`-w columns'

`--width=columns'

Use an output width of columns. See section Controlling Side by Side Format. Note that for historical reasons, this option is `-W' in `diff`, `-w' in `sdiff`.

`-W'

Ignore horizontal whitespace when comparing lines. See section Suppressing Differences in Blank and Tab Spacing. Note that for historical reasons, this option is `-w' in `diff`, `-W' in `sdiff`.

Incomplete Lines

When an input file ends in a non-newline character, its last line is called an incomplete line because its last character is not a newline. All other lines are called full lines and end in a newline character. Incomplete lines do not match full lines unless differences in blank lines are ignored (see section Suppressing Differences in Blank Lines).

An incomplete line is normally distinguished on output from a full line by a following line that starts with `\'. However, the RCS format (see section RCS Scripts) outputs the incomplete line as-is, without any trailing newline or following line. The side by side format normally represents incomplete lines as-is, but in some cases uses a `\' or `/' gutter marker; See section Showing Differences Side by Side. The if-then-else line format preserves a line's incompleteness with `%L', and discards the newline with `%l'; See section Line Formats. Finally, with the `ed` and forward `ed` output formats (see section `diff` Output Formats) `diff` cannot represent an incomplete line, so it pretends there was a newline and reports an error.

For example, suppose `F' and `G' are one-byte files that contain just `f' and `g', respectively. Then `diff F G' outputs

```
1c1
< f
\ No newline at end of file
---
> g
```


```
\ No newline at end of file
```

(The exact message may differ in non-English locales.) `diff -n F G' outputs the following without a trailing newline:

```
d1 1
a1 1
g
```

`diff -e F G' reports two errors and outputs the following:

```
1c
g
.
```

Future Projects

Here are some ideas for improving GNU `diff` and `patch`. The GNU project has identified some improvements as potential programming projects for volunteers. You can also help by reporting any bugs that you find.

If you are a programmer and would like to contribute something to the GNU project, please consider volunteering for one of these projects. If you are seriously contemplating work, please write to `gnu@prep.ai.mit.edu' to coordinate with other volunteers.

Suggested Projects for Improving GNU `diff` and `patch`

One should be able to use GNU `diff` to generate a patch from any pair of directory trees, and given the patch and a copy of one such tree, use `patch` to generate a faithful copy of the other. Unfortunately, some changes to directory trees cannot be expressed using current patch formats; also, `patch` does not handle some of the existing formats. These shortcomings motivate the following suggested projects.

Handling Changes to the Directory Structure

`diff` and `patch` do not handle some changes to directory structure. For example, suppose one directory tree contains a directory named `D' with some subsidiary files, and another contains a file with the same name `D'. `diff -r' does not output enough information for `patch` to transform the the directory subtree into the file.

There should be a way to specify that a file has been deleted without having to include its entire contents in the patch file. There should also be a way to tell `patch` that a file was renamed, even if there is no way for `diff` to generate such information.

These problems can be fixed by extending the `diff` output format to represent changes in directory

structure, and extending `patch` to understand these extensions.

Files that are Neither Directories Nor Regular Files

Some files are neither directories nor regular files: they are unusual files like symbolic links, device special files, named pipes, and sockets. Currently, `diff` treats all these files like regular files. However, this means that `patch` cannot represent changes to such files. For example, if you change which file a symbolic link points to, `diff` outputs the difference between the two files, instead of the change to the symbolic link.

`diff` should optionally report changes to special files specially, and `patch` should be extended to understand these extensions.

File Names that Contain Unusual Characters

When a file name contains an unusual character like a newline or whitespace, `diff -r' generates a patch that `patch` cannot parse. The problem is with format of `diff` output, not just with `patch`, because with odd enough file names one can cause `diff` to generate a patch that is syntactically correct but patches the wrong files. The format of `diff` output should be extended to handle all possible file names.

Arbitrary Limits

GNU `diff` can analyze files with arbitrarily long lines and files that end in incomplete lines. However, `patch` cannot patch such files. The `patch` internal limits on line lengths should be removed, and `patch` should be extended to parse `diff` reports of incomplete lines.

Handling Files that Do Not Fit in Memory

`diff` operates by reading both files into memory. This method fails if the files are too large, and `diff` should have a fallback.

One way to do this is to scan the files sequentially to compute hash codes of the lines and put the lines in equivalence classes based only on hash code. Then compare the files normally. This does produce some false matches.

Then scan the two files sequentially again, checking each match to see whether it is real. When a match is not real, mark both the "matching" lines as changed. Then build an edit script as usual.

The output routines would have to be changed to scan the files sequentially looking for the text to print.

Ignoring Certain Changes

It would be nice to have a feature for specifying two strings, one in from-file and one in to-file, which should be considered to match. Thus, if the two strings are `foo' and `bar', then if two lines differ only in that `foo' in file 1 corresponds to `bar' in file 2, the lines are treated as identical.

It is not clear how general this feature can or should be, or what syntax should be used for it.

Reporting Bugs

If you think you have found a bug in GNU `cmp`, `diff`, `diff3`, `sdiff`, or `patch`, please report it by electronic mail to `bug-gnu-utils@prep.ai.mit.edu'. Send as precise a description of the problem as you can, including sample input files that produce the bug, if applicable.

Because Larry Wall has not released a new version of `patch` since mid 1988 and the GNU version of `patch` has been changed since then, please send bug reports for `patch` by electronic mail to both `bug-gnu-utils@prep.ai.mit.edu' and `lwall@netlabs.com'.

Concept Index

c

d

e

f

h

i

l

m

n

o

p

r

s

t

u

w

GNU MP

The GNU Multiple Precision Arithmetic Library

Edition 1.3.2

May 1993

by Torbj@"orn Granlund

GNU MP

TOC

Published by the Free Software Foundation
675 Massachusetts Avenue,
Cambridge, MA 02139 USA

GNU MP Copying Conditions

This library is free; this means that everyone is free to use it and free to redistribute it on a free basis. The library is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of this library that they might get from you.

Specifically, we want to make sure that you have the right to give away copies of the library, that you receive source code or else can get it if you want it, that you can change this library or use pieces of it in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of the GMP library, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for the GMP library. If it is modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

The precise conditions of the license for the GMP library are found in the General Public License that accompany the source code.

Introduction to MP

GNU MP is a portable library for arbitrary precision integer and rational number arithmetic.(1) It aims to provide the fastest possible arithmetic for all applications that need more than two words of integer precision.

Most often, applications tend to use just a few words of precision; but some applications may need thousands of words. GNU MP is designed to give good performance for both kinds of applications, by choosing algorithms based on the sizes of the operands.

There are five groups of functions in the MP library:

1. Functions for signed integer arithmetic, with names beginning with `mpz_`.
2. Functions for rational number arithmetic, with names beginning with `mpq_`.
3. Functions compatible with Berkeley MP, such as `itom`, `madd`, and `mult`.
4. Fast low-level functions that operate on natural numbers. These are used by the functions in the preceding groups, and you can also call them directly from very time-critical user programs. These functions' names begin with `mpn_`.
5. Miscellaneous functions.

As a general rule, all MP functions expect output arguments before input arguments. This notation is based on an analogy with the assignment operator. (The BSD MP compatibility functions disobey this rule, having the output argument(s) last.) Multi-precision numbers, whether output or input, are always passed as addresses to the declared type.

Nomenclature and Data Types

In this manual, integer means a multiple precision integer, as used in the MP package. The C data type for such integers is `MP_INT`. For example:

```
MP_INT sum;

struct foo { MP_INT x, y; };

MP_INT vec[20];
```

Rational number means a multiple precision fraction. The C data type for these fractions is `MP_RAT`. For example:

```
MP_RAT quotient;
```

A limb means the part of a multi-precision number that fits in a single word. (We chose this word because a limb of the human body is analogous to a digit, only larger, and containing several digits.) Normally a limb contains 32 bits.

Thanks

I would like to thank Gunnar Sjoedin and Hans Riesel for their help with mathematical problems, Richard Stallman for his help with design issues and for revising this manual, Brian Beuning and Doug Lea for their testing of various versions of the library, and Joachim Hollman for his many valuable suggestions.

Special thanks to Brian Beuning, he has shaked out many bugs from early versions of the code!

John Amanatides of York University in Canada contributed the function `mpz_probab_prime_p`.

Initialization

Before you can use a variable or object of type `MP_INT` or `MP_RAT`, you must initialize it. This fills in the components that point to dynamically allocated space for the limbs of the number.

When you are finished using the object, you should clear out the object. This frees the dynamic space that it points to, so the space can be used again.

Once you have initialized the object, you need not be concerned about allocating additional space. The functions in the MP package automatically allocate additional space when the object does not already have enough space. They do not, however, reduce the space in use when a smaller number is stored in the object. Most of the time, this policy is best, since it avoids frequent re-allocation. If you want to reduce the space in an object to the minimum needed, you can do `_mpz_realloc (&object, mpz_size (&object))`.

The functions to initialize numbers are `mpz_init` (for `MP_INT`) and `mpq_init` (for `MP_RAT`).

`mpz_init` allocates space for the limbs, and stores a pointer to that space in the `MP_INT` object. It also stores the value 0 in the object.

In the same manner, `mpq_init` allocates space for the numerator and denominator limbs, and stores pointers to these spaces in the `MP_RAT` object.

To clear out a number object, use `mpz_clear` and `mpq_clear`, respectively.

Here is an example of use:

```
{
  MP_INT temp;
  mpz_init (&temp);

  ... store and read values in temp zero or more times ...

  mpz_clear (&temp):
}
```

You might be tempted to copy an integer from one object to another like this:


```
MP_INT x, y;

x = y;
```

Although valid C, **this is an error.** Rather than copying the integer value from `y` to `x` it will make the two variables share storage. Subsequent assignments to one variable would change the other mysteriously. And if you were to clear out both variables subsequently, you would confuse `malloc` and cause your program to crash.

To copy the value properly, you must use the function `mpz_set`. (see section Integer Assignment Functions)

Integer Functions

This chapter describes the MP functions for performing integer arithmetic.

The integer functions use arguments and values of type pointer-to-`MP_INT` (see section Nomenclature and Data Types). The type `MP_INT` is a structure, but applications should not refer directly to its components. Include the header `` `gmp.h' `` to get the definition of `MP_INT`.

Initializing Integer Objects

Most of the functions for integer arithmetic assume that the output is stored in an object already initialized. For example, `mpz_add` stores the result of addition (see section Integer Arithmetic Functions). Thus, you must initialize the object before storing the first value in it. You can do this separately by calling the function `mpz_init`.

Function: void **mpz_init** *(MP_INT *integer)*

Initialize integer with limb space and set the initial numeric value to 0. Each variable should normally only be initialized once, or at least cleared out (using `mpz_clear`) between each initialization.

Here is an example of using `mpz_init`:

```
{
  MP_INT integ;
  mpz_init (&integ);
  ...
  mpz_add (&integ, ...);
  ...
  mpz_sub (&integ, ...);

  /* Unless you are now exiting the program, do ... */
  mpz_clear (&integ);
}
```

As you can see, you can store new values any number of times, once an object is initialized.

Function: void **mpz_clear** *(MP_INT *integer)*

Free the limb space occupied by integer. Make sure to call this function for all `MP_INT` variables when you are done with them.

Function: void * **_mpz_realloc** *(MP_INT *integer, mp_size new_alloc)*

Change the limb space allocation to new_alloc limbs. This function is not normally called from user code, but it can be used to give memory back to the heap, or to increase the space of a variable to avoid repeated automatic re-allocation.

Function: void **mpz_array_init** *(MP_INT integer_array[], size_t array_size, mp_size fixed_num_limbs)*

Allocate **fixed** limb space for all array_size integers in integer_array. The fixed allocation for each integer in the array is fixed_num_limbs. This function is useful for decreasing the working set for some algorithms that use large integer arrays. If the fixed space will be insufficient for storing the result of a subsequent calculation, the result is unpredictable.

There is no way to de-allocate the storage allocated by this function. Don't call `mpz_clear`!

Integer Assignment Functions

These functions assign new values to already initialized integers (see section Initializing Integer Objects).

Function: void **mpz_set** *(MP_INT *dest_integer, MP_INT *src_integer)*

Assign dest_integer from src_integer.

Function: void **mpz_set_ui** *(MP_INT *integer, unsigned long int initial_value)*

Set the value of integer from initial_value.

Function: void **mpz_set_si** *(MP_INT *integer, signed long int initial_value)*

Set the value of integer from initial_value.

Function: int **mpz_set_str** *(MP_INT *integer, char *initial_value, int base)*

Set the value of integer from initial_value, a '\0'-terminated C string in base base. White space is allowed in the string, and is simply ignored. The base may vary from 2 to 36. If base is 0, the actual base is determined from the leading characters: if the first two characters are `0x' or `0X', hexadecimal is assumed, otherwise if the first character is `0', octal is assumed, otherwise decimal is assumed.

This function returns 0 if the entire string up to the '\0' is a valid number in base base. Otherwise it returns -1.

Combined Initialization and Assignment Functions

For your convenience, MP provides a parallel series of initialize-and-set arithmetic functions which initialize the output and then store the value there. These functions' names have the form `mpz_init_set...`

Here is an example of using one:


```
{
  MP_INT integ;
  mpz_init_set_str (&integ, "3141592653589793238462643383279502884", 10);
  ...
  mpz_sub (&integ, ...);

  mpz_clear (&integ);
}
```

Once the integer has been initialized by any of the `mpz_init_set...` functions, it can be used as the source or destination operand for the ordinary integer functions. Don't use an initialize-and-set function on a variable already initialized!

Function: void **mpz_init_set** *(MP_INT *dest_integer, MP_INT *src_integer)*

Initialize dest_integer with limb space and set the initial numeric value from src_integer.

Function: void **mpz_init_set_ui** *(MP_INT *dest_integer, unsigned long int src_ulong)*

Initialize dest_integer with limb space and set the initial numeric value from src_ulong.

Function: void **mpz_init_set_si** *(MP_INT *dest_integer, signed long int src_slong)*

Initialize dest_integer with limb space and set the initial numeric value from src_slong.

Function: int **mpz_init_set_str** *(MP_INT *dest_integer, char *src_cstring, int base)*

Initialize dest_integer with limb space and set the initial numeric value from src_cstring, a '\0'-terminated C string in base base. The base may vary from 2 to 36. There may be white space in the string.

If the string is a correct base base number, the function returns 0; if an error occurs it returns -1. dest_integer is initialized even if an error occurs. (I.e., you have to call mpz_clear for it.)

Conversion Functions

Function: unsigned long int **mpz_get_ui** *(MP_INT *src_integer)*

Return the least significant limb from src_integer. This function together with `mpz_div_2exp(..., src_integer, CHAR_BIT*sizeof(unsigned long int))` can be used to extract the limbs of an integer efficiently.

Function: signed long int **mpz_get_si** *(MP_INT *src_integer)*

If src_integer fits into a `signed long int` return the value of src_integer. Otherwise return the least significant bits of src_integer, with the same sign as src_integer.

Function: char * **mpz_get_str** *(char *string, int base, MP_INT *integer)*

Convert integer to a '\0'-terminated C string in string, using base base. The base may vary from 2 to 36. If string is NULL, space for the string is allocated using the default allocation function.

If string is not NULL, it should point to a block of storage enough large for the result. To find out the right

amount of space to provide for string, use `mpz_sizeinbase (integer, base) + 2`. The "+ 2" for a possible minus sign, and for the terminating null character. (see section Miscellaneous Functions).

This function returns a pointer to the result string.

Integer Arithmetic Functions

Function: void **mpz_add** *(MP_INT *sum, MP_INT *addend1, MP_INT *addend2)*

Function: void **mpz_add_ui** *(MP_INT *sum, MP_INT *addend1, unsigned long int addend2)*

Set sum to addend1 + addend2.

Function: void **mpz_sub** *(MP_INT *difference, MP_INT *minuend, MP_INT *subtrahend)*

Function: void **mpz_sub_ui** *(MP_INT *difference, MP_INT *minuend, unsigned long int subtrahend)*

Set difference to minuend - subtrahend.

Function: void **mpz_mul** *(MP_INT *product, MP_INT *multiplicator, MP_INT *multiplicand)*

Function: void **mpz_mul_ui** *(MP_INT *product, MP_INT *multiplicator, unsigned long int multiplicand)*

Set product to multiplicator times multiplicand.

Division is undefined if the divisor is zero, and passing a zero divisor to the divide or modulo functions, as well passing a zero mod argument to the powm functions, will make these functions intentionally divide by zero. This gives the user the possibility to handle arithmetic exceptions in these functions in the same manner as other arithmetic exceptions.

Function: void **mpz_div** *(MP_INT *quotient, MP_INT *dividend, MP_INT *divisor)*

Function: void **mpz_div_ui** *(MP_INT *quotient, MP_INT *dividend, unsigned long int divisor)*

Set quotient to dividend / divisor. The quotient is rounded towards 0.

Function: void **mpz_mod** *(MP_INT *remainder, MP_INT *divdend, MP_INT *divisor)*

Function: void **mpz_mod_ui** *(MP_INT *remainder, MP_INT *divdend, unsigned long int divisor)*

Divide dividend and divisor and put the remainder in remainder. The remainder has the same sign as the dividend, and its absolute value is less than the absolute value of the divisor.

Function: void **mpz_divmod** *(MP_INT *quotient, MP_INT *remainder, MP_INT *dividend, MP_INT *divisor)*

Function: void **mpz_divmod_ui** *(MP_INT *quotient, MP_INT *remainder, MP_INT *dividend, unsigned long int divisor)*

Divide dividend and divisor and put the quotient in quotient and the remainder in remainder. The quotient is rounded towards 0. The remainder has the same sign as the dividend, and its absolute value is less than the absolute value of the divisor.

If quotient and remainder are the same variable, the results are not defined.

TOC

Function: void **mpz_mdiv** *(MP_INT *quotient, MP_INT *dividend, MP_INT *divisor)*

Function: void **mpz_mdiv_ui** *(MP_INT *quotient, MP_INT *dividend, unsigned long int divisor)*

Set quotient to dividend / divisor. The quotient is rounded towards -infinity.

Function: void **mpz_mmod** *(MP_INT *remainder, MP_INT *divdend, MP_INT *divisor)*

Function: unsigned long int **mpz_mmod_ui** *(MP_INT *remainder, MP_INT *divdend, unsigned long int divisor)*

Divide dividend and divisor and put the remainder in remainder. The remainder is always positive, and its value is less than the value of the divisor.

For `mpz_mmod_ui` the remainder is returned, and if remainder is not NULL, also stored there.

Function: void **mpz_mdivmod** *(MP_INT *quotient, MP_INT *remainder, MP_INT *dividend, MP_INT *divisor)*

Function: unsigned long int **mpz_mdivmod_ui** *(MP_INT *quotient, MP_INT *remainder, MP_INT *dividend, unsigned long int divisor)*

Divide dividend and divisor and put the quotient in quotient and the remainder in remainder. The quotient is rounded towards -infinity. The remainder is always positive, and its value is less than the value of the divisor.

For `mpz_mdivmod_ui` the remainder is small enough to fit in an `unsigned long int`, and is therefore returned. If remainder is not NULL, the remainder is also stored there.

If quotient and remainder are the same variable, the results are not defined.

Function: void **mpz_sqrt** *(MP_INT *root, MP_INT *operand)*

Set root to the square root of operand. The result is rounded towards zero.

Function: void **mpz_sqrtrem** *(MP_INT *root, MP_INT *remainder, MP_INT *operand)*

Set root to the square root of operand, as with `mpz_sqrt`. Set remainder to (i.e. zero if operand is a perfect square).

If root and remainder are the same variable, the results are not defined.

Function: int **mpz_perfect_square_p** *(MP_INT *square)*

Return non-zero if square is perfect, i.e. if the square root of square is integral. Return zero otherwise.

Function: int **mpz_probab_prime_p** *(MP_INT *n, int reps)*

An implementation of the probabilistic primality test found in Knuth's Seminumerical Algorithms book. If the function `mpz_probab_prime_p(n, reps)` returns 0 then n is not prime. If it returns 1, then n is `probably' prime. The probability of a false positive is (1/4)**reps, where reps is the number of internal passes of the probabilistic algorithm. Knuth indicates that 25 passes are reasonable.

Function: void **mpz_powm** *(MP_INT *res, MP_INT *base, MP_INT *exp, MP_INT *mod)*

Function: void **mpz_powm_ui** *(MP_INT *res, MP_INT *base, unsigned long int exp, MP_INT *mod)*

Set res to (base raised to exp) modulo mod. If exp is negative, the result is undefined.

Function: void **mpz_pow_ui** *(MP_INT *res, MP_INT *base, unsigned long int exp)*

Set res to base raised to exp.

Function: void **mpz_fac_ui** *(MP_INT *res, unsigned long int n)*

Set res n!, the factorial of n.

Function: void **mpz_gcd** *(MP_INT *res, MP_INT *operand1, MP_INT *operand2)*

Set res to the greatest common divisor of operand1 and operand2.

Function: void **mpz_gcdext** *(MP_INT *g, MP_INT *s, MP_INT *t, MP_INT *a, MP_INT *b)*

Compute g, s, and t, such that as + bt = g = `gcd` (a, b). If t is NULL, that argument is not computed.

Function: void **mpz_neg** *(MP_INT *negated_operand, MP_INT *operand)*

Set negated_operand to -operand.

Function: void **mpz_abs** *(MP_INT *positive_operand, MP_INT *signed_operand)*

Set positive_operand to the absolute value of signed_operand.

Function: int **mpz_cmp** *(MP_INT *operand1, MP_INT *operand2)*

Function: int **mpz_cmp_ui** *(MP_INT *operand1, unsigned long int operand2)*

Function: int **mpz_cmp_si** *(MP_INT *operand1, signed long int operand2)*

Compare operand1 and operand2. Return a positive value if operand1 > operand2, zero if operand1 = operand2, and a negative value if operand1 < operand2.

Function: void **mpz_mul_2exp** *(MP_INT *product, MP_INT *multiplicator, unsigned long int exponent_of_2)*

Set product to multiplicator times 2 raised to exponent_of_2. This operation can also be defined as a left shift, exponent_of_2 steps.

Function: void **mpz_div_2exp** *(MP_INT *quotient, MP_INT *dividend, unsigned long int exponent_of_2)*

Set quotient to dividend divided by 2 raised to exponent_of_2. This operation can also be defined as a right shift, exponent_of_2 steps, but unlike the >> operator in C, the result is rounded towards 0.

Function: void **mpz_mod_2exp** *(MP_INT *remainder, MP_INT *dividend, unsigned long int exponent_of_2)*

Set remainder to dividend mod (2 raised to exponent_of_2). The sign of remainder will have the same sign as dividend.

This operation can also be defined as a masking of the exponent_of_2 least significant bits.

TOC

Logical Functions

Function: void **mpz_and** *(MP_INT *conjunction, MP_INT *operand1, MP_INT *operand2)*

Set conjunction to operand1 logical-and operand2.

Function: void **mpz_ior** *(MP_INT *disjunction, MP_INT *operand1, MP_INT *operand2)*

Set disjunction to operand1 inclusive-or operand2.

Function: void **mpz_xor** *(MP_INT *disjunction, MP_INT *operand1, MP_INT *operand2)*

Set disjunction to operand1 exclusive-or operand2.

This function is missing in the current release.

Function: void **mpz_com** *(MP_INT *complemented_operand, MP_INT *operand)*

Set complemented_operand to the one's complement of operand.

Input and Output Functions

Functions that perform input from a standard I/O stream, and functions for output conversion.

Function: void **mpz_inp_raw** *(MP_INT *integer, FILE *stream)*

Input from standard I/O stream stream in the format written by `mpz_out_raw`, and put the result in integer.

Function: void **mpz_inp_str** *(MP_INT *integer, FILE *stream, int base)*

Input a string in base base from standard I/O stream stream, and put the read integer in integer. The base may vary from 2 to 36. If base is 0, the actual base is determined from the leading characters: if the first two characters are `0x' or `0X', hexadecimal is assumed, otherwise if the first character is `0', octal is assumed, otherwise decimal is assumed.

Function: void **mpz_out_raw** *(FILE *stream, MP_INT *integer)*

Output integer on standard I/O stream stream, in raw binary format. The integer is written in a portable format, with 4 bytes of size information, and that many bytes of limbs. Both the size and the limbs are written in decreasing significance order.

Function: void **mpz_out_str** *(FILE *stream, int base, MP_INT *integer)*

Output integer on standard I/O stream stream, as a string of digits in base base. The base may vary from 2 to 36.

Rational Number Functions

All rational arithmetic functions canonicalize the result, so that the denominator and the numerator have no common factors. Zero has the unique representation 0/1.

The set of functions is quite small. Maybe it will be extended in a future release.

Function: void **mpq_init** *(MP_RAT *dest_rational)*

Initialize dest_rational with limb space and set the initial numeric value to 0/1. Each variable should normally only be initialized once, or at least cleared out (using the function `mpq_clear`) between each initialization.

Function: void **mpq_clear** *(MP_RAT *rational_number)*

Free the limb space occupied by rational_number. Make sure to call this function for all `MP_RAT` variables when you are done with them.

Function: void **mpq_set** *(MP_RAT *dest_rational, MP_RAT *src_rational)*

Assign dest_rational from src_rational.

Function: void **mpq_set_ui** *(MP_RAT *rational_number, unsigned long int numerator, unsigned long int denominator)*

Set the value of rational_number to numerator/denominator. If numerator and denominator have common factors, they are divided out before rational_number is assigned.

Function: void **mpq_set_si** *(MP_RAT *rational_number, signed long int numerator, unsigned long int denominator)*

Like `mpq_set_ui`, but numerator is signed.

Function: void **mpq_add** *(MP_RAT *sum, MP_RAT *addend1, MP_RAT *addend2)*

Set sum to addend1 + addend2.

Function: void **mpq_sub** *(MP_RAT *difference, MP_RAT *minuend, MP_RAT *subtrahend)*

Set difference to minuend - subtrahend.

Function: void **mpq_mul** *(MP_RAT *product, MP_RAT *multiplicator, MP_RAT *multiplicand)*

Set product to multiplicator * multiplicand

Function: void **mpq_div** *(MP_RAT *quotient, MP_RAT *dividend, MP_RAT *divisor)*

Set quotient to dividend / divisor.

Function: void **mpq_neg** *(MP_RAT *negated_operand, MP_RAT *operand)*

Set negated_operand to -operand.

Function: int **mpq_cmp** *(MP_RAT *operand1, MP_RAT *operand2)*

Compare operand1 and operand2. Return a positive value if operand1 > operand2, zero if operand1 = operand2, and a negative value if operand1 < operand2.

Function: void **mpq_inv** *(MP_RAT *inverted_number, MP_RAT *number)*

Invert number by swapping the numerator and denominator. If the new denominator becomes zero, this routine will divide by zero.

Function: void **mpq_set_num** *(MP_RAT *rational_number, MP_INT *numerator)*

Make numerator become the numerator of rational_number by copying.

Function: void **mpq_set_den** *(MP_RAT *rational_number, MP_INT *denominator)*

Make denominator become the denominator of rational_number by copying. If denominator < 0 the denominator of rational_number is set to the absolute value of denominator, and the sign of the numerator of rational_number is changed.

Function: void **mpq_get_num** *(MP_INT *numerator, MP_RAT *rational_number)*

Copy the numerator of rational_number to the integer numerator, to prepare for integer operations on the numerator.

Function: void **mpq_get_den** *(MP_INT *denominator, MP_RAT *rational_number)*

Copy the denominator of rational_number to the integer denominator, to prepare for integer operations on the denominator.

Low-level Functions

The next release of the GNU MP library (2.0) will include changes to some mpn functions. Programs that use these functions according to the descriptions below will therefore not work with the next release.

The low-level function layer is designed to be as fast as possible, **not** to provide a coherent calling interface. The different functions have similar interfaces, but there are variations that might be confusing. These functions do as little as possible apart from the real multiple precision computation, so that no time is spent on things that not all callers need.

A source operand is specified by a pointer to the least significant limb and a limb count. A destination operand is specified by just a pointer. It is the responsability of the caller to ensure that the destination has enough space for storing the result.

With this way of specifying source operands, it is possible to perform computations on subranges of an argument, and store the result into a subrange of a destination.

All these functions require that the operands are normalized in the sense that the most significant limb must be non-zero. (A future release of might drop this requirement.)

The low-level layer is the base for the implementation of the `mpz_` and `mpq_` layers.

The code below adds the number beginning at src1_ptr and the number beginning at src2_ptr and writes the

sum at dest_ptr. A constraint for `mpn_add` is that src1_size must not be smaller that src2_size.

`mpn_add (dest_ptr, src1_ptr, src1_size, src2_ptr, src2_size)`

In the description below, a source operand is identified by the pointer to the least significant limb, and the limb count in braces.

Function: mp_size **mpn_add** *(mp_ptr dest_ptr, mp_srcptr src1_ptr, mp_size src1_size, mp_srcptr src2_ptr, mp_size src2_size)*

Add {src1_ptr, src1_size} and {src2_ptr, src2_size}, and write the src1_size least significant limbs of the result to dest_ptr. Carry-out, either 0 or 1, is returned.

This function requires that src1_size is greater than or equal to src2_size.

Function: mp_size **mpn_sub** *(mp_ptr dest_ptr, mp_srcptr src1_ptr, mp_size src1_size, mp_srcptr src2_ptr, mp_size src2_size)*

Subtarct {src2_ptr, src2_size} from {src1_ptr, src1_size}, and write the result to dest_ptr.

Return 1 if the minuend < the subtrahend. Otherwise, return the negative difference between the number of words in the result and the minuend. I.e. return 0 if the result has src1_size words, -1 if it has src1_size - 1 words, etc.

This function requires that src1_size is greater than or equal to src2_size.

Function: mp_size **mpn_mul** *(mp_ptr dest_ptr, mp_srcptr src1_ptr, mp_size src1_size, mp_srcptr src2_ptr, mp_size src2_size)*

Multiply {src1_ptr, src1_size} and {src2_ptr, src2_size}, and write the result to dest_ptr. The exact size of the result is returned.

The destination has to have space for src1_size + src1_size limbs, even if the result might be one limb smaller.

This function requires that src1_size is greater than or equal to src2_size. The destination must be distinct from either input operands.

Function: mp_size **mpn_div** *(mp_ptr dest_ptr, mp_ptr src1_ptr, mp_size src1_size, mp_srcptr src2_ptr, mp_size src2_size)*

Divide {src1_ptr, src1_size} by {src2_ptr, src2_size}, and write the quotient to dest_ptr, and the remainder to src1_ptr.

Return 0 if the quotient size is at most (src1_size - src2_size), and 1 if the quotient size is at most (src1_size - src2_size + 1). The caller has to check the most significant limb to find out the exact size.

The most significant bit of the most significant limb of the divisor has to be set.

This function requires that src1_size is greater than or equal to src2_size. The quotient, pointed to by dest_ptr, must be distinct from either input operands.

Function: mp_limb **mpn_lshift** *(mp_ptr dest_ptr, mp_srcptr src_ptr, mp_size src_size, unsigned long int count)*

Shift {src_ptr, src_size} count bits to the left, and write the src_size least significant limbs of the result t dest_ptr. count might be in the range 1 to n - 1, on an n-bit machine. The limb shifted out is returned.

Overlapping of the destination space and the source space is allowed in this function, provdied dest_ptr >= src_ptr.

Function: mp_size **mpn_rshift** *(mp_ptr dest_ptr, mp_srcptr src_ptr, mp_size src_size, unsigned long int count)*

Shift {src_ptr, src_size} count bits to the right, and write the src_size least significant limbs of the result to dest_ptr. count might be in the range 1 to n - 1, on an n-bit machine. The size of the result is returned.

Overlaping of the destination space and the source space is allowed in this function, provdied dest_ptr <= src_ptr.

Function: mp_size **mpn_rshiftci** *(mp_ptr dest_ptr, mp_srcptr src_ptr, mp_size src_size, unsigned long int count, mp_limb inlimb)*

Like mpn_rshift, but use inlimb to feed the least significant end of the destination.

Function: int **mpn_cmp** *(mp_srcptr src1_ptr, mp_srcptr src2_ptr, mp_size size)*

Compare {src1_ptr, size} and {src2_ptr, size} and return a positive value if src1 > src2, 0 of they are equal, and a negative value if src1 < src2.

Berkeley MP Compatible Functions

These functions are intended to be fully compatible with the Berkeley MP library which is available on many BSD derived U*ix systems.

The original Berkeley MP library has a usage restriction: you cannot use the same variable as both source and destination in a single function call. The compatible functions in GNU MP do not share this restriction--inputs and outputs may overlap.

It is not recommended that new programs are written using these functions. Apart from the incomplete set of functions, the interface for initializing `MINT` objects is more error prone, and the `pow` function collides with `pow` in `` `libm.a' ``.

Include the header `` `mp.h' `` to get the definition of the necessary types and functions. If you are on a BSD derived system, make sure to include GNU `` `mp.h' `` if you are going to link the GNU `` `libmp.a' `` to you program. This means that you probably need to give the -I<dir> option to the compiler, where <dir> is the directory where you have GNU `` `mp.h' ``.

Function: MINT * **itom** *(signed short int initial_value)*

Allocate an integer consisting of a `MINT` object and dynamic limb space. Initialize the integer to initial_value. Return a pointer to the `MINT` object.

Function: MINT * **xtom** *(char *initial_value)*

Allocate an integer consisting of a `MINT` object and dynamic limb space. Initialize the integer from

initial_value, a hexadecimal, '\0'-terminate C string. Return a pointer to the `MINT` object.

Function: void **move** *(MINT *src, MINT *dest)*

Set dest to src by copying. Both variables must be previously initialized.

Function: void **madd** *(MINT *src_1, MINT *src_2, MINT *destination)*

Add src_1 and src_2 and put the sum in destination.

Function: void **msub** *(MINT *src_1, MINT *src_2, MINT *destination)*

Subtract src_2 from src_1 and put the difference in destination.

Function: void **mult** *(MINT *src_1, MINT *src_2, MINT *destination)*

Multiply src_1 and src_2 and put the product in destination.

Function: void **mdiv** *(MINT *dividend, MINT *divisor, MINT *quotient, MINT *remainder)*

Function: void **sdiv** *(MINT *dividend, signed short int divisor, MINT *quotient, signed short int *remainder)*

Set quotient to dividend / divisor, and remainder to dividend mod divisor. The quotient is rounded towards zero; the remainder has the same sign as the dividend.

Some implementations of this function return a remainder whose sign is inverted if the divisor is negative. Such a definition makes little sense from a mathematical point of view. GNU MP might be considered incompatible with the traditional MP in this respect.

Function: void **msqrt** *(MINT *operand, MINT *root, MINT *remainder)*

Set root to the square root of operand, as with `mpz_sqrt`. Set remainder to (i.e. zero if operand is a perfect square).

Function: void **pow** *(MINT *base, MINT *exp, MINT *mod, MINT *dest)*

Set dest to (base raised to exp) modulo mod.

Function: void **rpow** *(MINT *base, signed short int exp, MINT *dest)*

Set dest to base raised to exp.

Function: void **gcd** *(MINT *operand1, MINT *operand2, MINT *res)*

Set res to the greatest common divisor of operand1 and operand2.

Function: int **mcmp** *(MINT *operand1, MINT *operand2)*

Compare operand1 and operand2. Return a positive value if operand1 > operand2, zero if operand1 = operand2, and a negative value if operand1 < operand2.

Function: void **min** *(MINT *dest)*

Input a decimal string from stdin, and put the read integer in dest. SPC and TAB are allowed in the number string, and are ignored.

Function: void **mout** *(MINT *src)*

Output src to stdout, as a decimal string. Also output a newline.

Function: char * **mtox** *(MINT *operand)*

Convert operand to a hexadecimal string, and return a pointer to the string. The returned string is allocated using the default memory allocation function, `malloc` by default. (See section Initialization, for an explanation of the memory allocation in MP).

Function: void **mfree** *(MINT *operand)*

De-allocate, the space used by operand. **This function should only be passed a value returned by `itom` or `xtom`.**

Miscellaneous Functions

Function: void **mpz_random** *(MP_INT *random_integer, mp_size max_size)*

Generate a random integer of at most max_size limbs. The generated random number doesn't satisfy any particular requirements of randomness.

Function: void **mpz_random2** *(MP_INT *random_integer, mp_size max_size)*

Generate a random integer of at most max_size limbs, with long strings of zeros and ones in the binary representation. Useful for testing functions and algorithms, since this kind of random numbers have proven to be more likely to trigger bugs.

Function: size_t **mpz_size** *(MP_INT *integer)*

Return the size of integer measured in number of limbs. If integer is zero, the returned value will be zero, if integer has one limb, the returned value will be one, etc. (See section Nomenclature and Data Types, for an explanation of the concept limb.)

Function: size_t **mpz_sizeinbase** *(MP_INT *integer, int base)*

Return the size of integer measured in number of digits in base base. The base may vary from 2 to 36. The returned value will be exact or 1 too big. If base is a power of 2, the returned value will always be exact.

This function is useful in order to allocate the right amount of space before converting integer to a string. The right amount of allocation is normally two more than the value returned by `mpz_sizeinbase` (one extra for a minus sign and one for the terminating '\0').

Custom Allocation

By default, the initialization functions use `malloc`, `realloc`, and `free` to do their work. If `malloc` or `realloc` fails, the MP package terminates execution after a printing fatal error message on standard error.

In some applications, you may wish to allocate memory in other ways, or you may not want to have a fatal error when there is no more memory available. To accomplish this, you can specify alternative functions for allocating and de-allocating memory. Use `mp_set_memory_functions` to do this.

`mp_set_memory_functions` has three arguments, allocate_function, reallocate_function, and deallocate_function, in that order. If an argument is NULL, the corresponding default function is retained.

The functions you supply should fit the following declarations:

`void * allocate_function (size_t alloc_size)`

This function should return a pointer to newly allocated space with at least alloc_size storage units.

`void * reallocate_function (void *ptr, size_t old_size, size_t new_size)`

This function should return a pointer to newly allocated space of at least new_size storage units, after copying the first old_size storage units from ptr. It should also de-allocate the space at ptr.

You can assume that the space at ptr was formely returned from allocate_function or reallocate_function, for a request for old_size storage units.

`void deallocate_function (void *ptr, size_t size)`

De-allocate the space pointed to by ptr.

You can assume that the space at ptr was formely returned from allocate_function or reallocate_function, for a request for size storage units.

(A storage unit is the unit in which the `sizeof` operator returns the size of an object, normally an 8 bit byte.)

NOTE: call `mp_set_memory_functions` only before calling any other MP functions. Otherwise, the user-defined allocation functions may be asked to re-allocate or de-allocate something previously allocated by the default allocation functions.

Reporting Bugs

If you think you have found a bug in the GNU MP library, please investigate it and report it. We have made this library available to you, and it is not to ask too much from you, to ask you to report the bugs that you find.

Please make sure that the bug is really in the GNU MP library.

You have to send us a test case that makes it possible for us to reproduce the bug.

You also have to explain what is wrong; if you get a crash, or if the results printed are not good and in that case, in what way.

Make sure that the bug report includes all information you would need to fix this kind of bug for someone else. Think twice.

If your bug report is good, we will do our best to help you to get a corrected version of the library; if the bug report is poor, we won't do anything about it (aside of chiding you to send better bug reports).

Send your bug report to: tege@gnu.ai.mit.edu.

If you think something in this manual is unclear, or downright incorrect, or if the language needs to be improved, please send a note to the same address.

References

- Donald E. Knuth, "The Art of Computer Programming", vol 2, "Seminumerical Algorithms", 2nd edition, Addison-Wesley, 1981.
- John D. Lipson, "Elements of Algebra and Algebraic Computing", The Benjamin Cummins Publishing Company Inc, 1981.
- Richard M. Stallman, "Using and Porting GCC", Free Software Foundation, 1993.
- Peter L. Montgomery, "Modular Multiplication Without Trial Division", Mathematics of Computation, volume 44, number 170, April 1985.

Concept Index

a

b

c

i

Function and Type Index

i

m

GNU MP

TOC

(1)

The limit of the precision is set by the available memory in your computer.

GNU Make

A Program for Directing Recompilation

Edition 0.48, for `make` Version 3.73 Beta.

April 1995

Richard M. Stallman and Roland McGrath

Go to the next section.

Published by the Free Software Foundation
675 Massachusetts Avenue,
Cambridge, MA 02139 USA
Printed copies are available for $20 each.
ISBN 1-882114-50-7

Cover art by Etienne Suvasa.

Overview of `make`

The `make` utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them. This manual describes GNU `make`, which was implemented by Richard Stallman and Roland McGrath. GNU `make` conforms to section 6.2 of IEEE Standard 1003.2-1992 (POSIX.2).

Our examples show C programs, since they are most common, but you can use `make` with any programming language whose compiler can be run with a shell command. Indeed, `make` is not limited to programs. You can use it to describe any task where some files must be updated automatically from others whenever the others change.

To prepare to use `make`, you must write a file called the makefile that describes the relationships among files in your program and provides commands for updating each file. In a program, typically, the executable file is updated from object files, which are in turn made by compiling source files.

Once a suitable makefile exists, each time you change some source files, this simple shell command:

```
make
```

suffices to perform all necessary recompilations. The `make` program uses the makefile data base and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the data base.

You can provide command line arguments to `make` to control which files should be recompiled, or h
See section How to Run `make`}.

How to Read This Manual

If you are new to `make`, or are looking for a general introduction, read the first few sections of each chapter, skipping the later sections. In each chapter, the first few sections contain introductory or general information and the later sections contain specialized or technical information. The exception is section An Introduction to Makefiles, all of which is introductory.

If you are familiar with other `make` programs, see section Features of GNU `make`}, which lists the enhancements GNU `make` has, and section Incompatibilities and Missing Features, which explains the few things GNU `make` lacks that others have.

For a quick summary, see section Summary of Options, section Quick Reference, and section Special Built-in Target Names.

Problems and Bugs

If you have problems with GNU `make` or think you've found a bug, please report it to the developers; we cannot promise to do anything but we might well want to fix it.

Before reporting a bug, make sure you've actually found a real bug. Carefully reread the documentation and see if it really says you can do what you're trying to do. If it's not clear whether you should be able to do something or not, report that too; it's a bug in the documentation!

Before reporting a bug or trying to fix it yourself, try to isolate it to the smallest possible makefile that reproduces the problem. Then send us the makefile and the exact results `make` gave you. Also say what you expected to occur; this will help us decide whether the problem was really in the documentation.

Once you've got a precise problem, please send electronic mail either through the Internet or via UUCP:

```
Internet address:
    bug-gnu-utils@prep.ai.mit.edu

UUCP path:
    mit-eddie!prep.ai.mit.edu!bug-gnu-utils
```

Please include the version number of `make` you are using. You can get this information with the command `make --version'. Be sure also to include the type of machine and operating system you are using. If possible, include the contents of the file `` `config.h' `` that is generated by the configuration process.

Non-bug suggestions are always welcome as well. If you have questions about things that are unclear in the documentation or are just obscure features, send a message to the bug reporting address. We cannot guarantee you'll get help with your problem, but many seasoned `make` users read the mailing list and

TOC

they will probably try to help you out. The maintainers sometimes answer such questions as well, wh
time permits.

Go to the next section.

Go to the previous, next section.

An Introduction to Makefiles

You need a file called a makefile to tell `make` what to do. Most often, the makefile tells `make` how to compile and link a program.

In this chapter, we will discuss a simple makefile that describes how to compile and link a text editor which consists of eight C source files and three header files. The makefile can also tell `make` how to run miscellaneous commands when explicitly asked (for example, to remove certain files as a clean-up operation). To see a more complex example of a makefile, see section Complex Makefile Example.

When `make` recompiles the editor, each changed C source file must be recompiled. If a header file has changed, each C source file that includes the header file must be recompiled to be safe. Each compilation produces an object file corresponding to the source file. Finally, if any source file has been recompiled, all the object files, whether newly made or saved from previous compilations, must be linked together to produce the new executable editor.

What a Rule Looks Like

A simple makefile consists of "rules" with the following shape:

```
target ... : dependencies ...
        command
        ...
        ...
```

A target is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as `clean' (see section Phony Targets).

A dependency is a file that is used as input to create the target. A target often depends on several files.

A command is an action that `make` carries out. A rule may have more than one command, each on its own line. **Please note:** you need to put a tab character at the beginning of every command line! This is an obscurity that catches the unwary.

Usually a command is in a rule with dependencies and serves to create a target file if any of the dependencies change. However, the rule that specifies commands for the target need not have dependencies. For example, the rule containing the delete command associated with the target `clean' does not have dependencies.

A rule, then, explains how and when to remake certain files which are the targets of the particular rule. `make` carries out the commands on the dependencies to create or update the target. A rule can also explain how and when to carry out an action. See section Writing Rules.

A makefile may contain other text besides rules, but a simple makefile need only contain rules. Rules may look somewhat more complicated than shown in this template, but all fit the pattern more or less.

A Simple Makefile

Here is a straightforward makefile that describes the way an executable file called `edit` depends on eight object files which, in turn, depend on eight C source and three header files.

In this example, all the C files include `` `defs.h' ``, but only those defining editing commands include `` `command.h' ``, and only low level files that change the editor buffer include `` `buffer.h' ``.

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

We split each long line into two lines using backslash-newline; this is like using one long line, but is easier to read.

To use this makefile to create the executable file called `` `edit' ``, type:

```
make
```

To use this makefile to delete the executable file and all the object files from the directory, type:


```
make clean
```

In the example makefile, the targets include the executable file `edit', and the object files `main.o' and `kbd.o'. The dependencies are files such as `main.c' and `defs.h'. In fact, each `.o' file is both a target and a dependency. Commands include `cc -c main.c' and `cc -c kbd.c'.

When a target is a file, it needs to be recompiled or relinked if any of its dependencies change. In addition, any dependencies that are themselves automatically generated should be updated first. In this example, `edit' depends on each of the eight object files; the object file `main.o' depends on the source file `main.c' and on the header file `defs.h'.

A shell command follows each line that contains a target and dependencies. These shell commands say how to update the target file. A tab character must come at the beginning of every command line to distinguish commands lines from other lines in the makefile. (Bear in mind that `make` does not know anything about how the commands work. It is up to you to supply commands that will update the target file properly. All `make` does is execute the commands in the rule you have specified when the target file needs to be updated.)

The target `clean' is not a file, but merely the name of an action. Since you normally do not want to carry out the actions in this rule, `clean' is not a dependency of any other rule. Consequently, `make` never does anything with it unless you tell it specifically. Note that this rule not only is not a dependency, it also does not have any dependencies, so the only purpose of the rule is to run the specified commands. Targets that do not refer to files but are just actions are called phony targets. See section Phony Targets, for information about this kind of target. See section Errors in Commands, to see how to cause `make` to ignore errors from `rm` or any other command.

How `make` Processes a Makefile

By default, `make` starts with the first rule (not counting rules whose target names start with `.'). This is called the default goal. (Goals are the targets that `make` strives ultimately to update. See section Arguments to Specify the Goals.)

In the simple example of the previous section, the default goal is to update the executable program `edit'; therefore, we put that rule first.

Thus, when you give the command:

```
make
```

`make` reads the makefile in the current directory and begins by processing the first rule. In the example, this rule is for relinking `edit'; but before `make` can fully process this rule, it must process the rules for the files that `edit' depends on, which in this case are the object files. Each of these files is processed according to its own rule. These rules say to update each `.o' file by compiling its source file. The recompilation must be done if the source file, or any of the header files named as dependencies, is more recent than the object file, or if the object file does not exist.

The other rules are processed because their targets appear as dependencies of the goal. If some other rule

is not depended on by the goal (or anything it depends on, etc.), that rule is not processed, unless you tell `make` to do so (with a command such as `make clean`).

Before recompiling an object file, `make` considers updating its dependencies, the source file and header files. This makefile does not specify anything to be done for them--the `.c' and `.h' files are not the targets of any rules--so `make` does nothing for these files. But `make` would update automatically generated C programs, such as those made by Bison or Yacc, by their own rules at this time.

After recompiling whichever object files need it, `make` decides whether to relink `edit'. This must be done if the file `edit' does not exist, or if any of the object files are newer than it. If an object file was just recompiled, it is now newer than `edit', so `edit' is relinked.

Thus, if we change the file `insert.c' and run `make`, `make` will compile that file to update `insert.o', and then link `edit'. If we change the file `command.h' and run `make`, `make` will recompile the object files `kbd.o', `command.o' and `files.o' and then link the file `edit'.

Variables Make Makefiles Simpler

In our example, we had to list all the object files twice in the rule for `edit' (repeated here):

```
edit : main.o kbd.o command.o display.o \
               insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                    insert.o search.o files.o utils.o
```

Such duplication is error-prone; if a new object file is added to the system, we might add it to one list and forget the other. We can eliminate the risk and simplify the makefile by using a variable. Variables allow a text string to be defined once and substituted in multiple places later (see section How to Use Variables).

It is standard practice for every makefile to have a variable named `objects`, `OBJECTS`, `objs`, `OBJS`, `obj`, or `OBJ` which is a list of all object file names. We would define such a variable `objects` with a line like this in the makefile:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

Then, each place we want to put a list of the object file names, we can substitute the variable's value by writing `$(objects)' (see section How to Use Variables).

Here is how the complete simple makefile looks when you use a variable for the object files:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
```


```
        cc -o edit $(objects)
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit $(objects)
```

Letting `make` Deduce the Commands

It is not necessary to spell out the commands for compiling the individual C source files, because `make` can figure them out: it has an implicit rule for updating a `.o' file from a correspondingly named `.c' file using a `cc -c' command. For example, it will use the command `cc -c main.c -o main.o' to compile `main.c' into `main.o'. We can therefore omit the commands from the rules for the object files. See section Using Implicit Rules.

When a `.c' file is used automatically in this way, it is also automatically added to the list of dependencies. We can therefore omit the `.c' files from the dependencies, provided we omit the commands.

Here is the entire example, with both of these changes, and a variable `objects` as suggested above:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
```


```
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        -rm edit $(objects)
```

This is how we would write the makefile in actual practice. (The complications associated with `clean' are described elsewhere. See section Phony Targets, and section Errors in Commands.)

Because implicit rules are so convenient, they are important. You will see them used frequently.

Another Style of Makefile

When the objects of a makefile are created only by implicit rules, an alternative style of makefile is possible. In this style of makefile, you group entries by their dependencies instead of by their targets. Here is what one looks like:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

Here `defs.h' is given as a dependency of all the object files; `command.h' and `buffer.h' are dependencies of the specific object files listed for them.

Whether this is better is a matter of taste: it is more compact, but some people dislike it because they find it clearer to put all the information about each target in one place.

Rules for Cleaning the Directory

Compiling a program is not the only thing you might want to write rules for. Makefiles commonly tell how to do a few other things besides compiling a program: for example, how to delete all the object files and executables so that the directory is `clean'.

Here is how we could write a `make` rule for cleaning our example editor:

```
clean:
        rm edit $(objects)
```

In practice, we might want to write the rule in a somewhat more complicated manner to handle unanticipated situations. We would do this:

```
.PHONY : clean
clean :
        -rm edit $(objects)
```

This prevents `make` from getting confused by an actual file called `` `clean' `` and causes it to continue in spite of errors from `rm`. (See section Phony Targets, and section Errors in Commands.)

A rule such as this should not be placed at the beginning of the makefile, because we do not want it to run by default! Thus, in the example makefile, we want the rule for `edit`, which recompiles the editor, to remain the default goal.

Since `clean` is not a dependency of `edit`, this rule will not run at all if we give the command `make' with no arguments. In order to make the rule run, we have to type `make clean'. See section How to Run `make`.

Go to the previous, next section.

Go to the previous, next section.

Writing Makefiles

The information that tells `make` how to recompile a system comes from reading a data base called the makefile.

What Makefiles Contain

Makefiles contain five kinds of things: explicit rules, implicit rules, variable definitions, directives, and comments. Rules, variables, and directives are described at length in later chapters.

- An explicit rule says when and how to remake one or more files, called the rule's targets. It lists the other files that the targets depend on, and may also give commands to use to create or update the targets. See section Writing Rules.
- An implicit rule says when and how to remake a class of files based on their names. It describes how a target may depend on a file with a name similar to the target and gives commands to create or update such a target. See section Using Implicit Rules.
- A variable definition is a line that specifies a text string value for a variable that can be substituted into the text later. The simple makefile example shows a variable definition for `objects` as a list of all object files (see section Variables Make Makefiles Simpler).
- A directive is a command for `make` to do something special while reading the makefile. These include:
  - Reading another makefile (see section Including Other Makefiles).
  - Deciding (based on the values of variables) whether to use or ignore a part of the makefile (see section Conditional Parts of Makefiles).
  - Defining a variable from a verbatim string containing multiple lines (see section Defining Variables Verbatim).
- `#' in a line of a makefile starts a comment. It and the rest of the line are ignored, except that a trailing backslash not escaped by another backslash will continue the comment across multiple lines. Comments may appear on any of the lines in the makefile, except within a `define` directive, and perhaps within commands (where the shell decides what is a comment). A line containing just a comment (with perhaps spaces before it) is effectively blank, and is ignored.

What Name to Give Your Makefile

By default, when `make` looks for the makefile, it tries the following names, in order: `GNUmakefile', `makefile' and `Makefile'.

Normally you should call your makefile either `makefile' or `Makefile'. (We recommend `Makefile' because it appears prominently near the beginning of a directory listing, right near other

important files such as `README'.) The first name checked, `GNUmakefile', is not recommend for most makefiles. You should use this name if you have a makefile that is specific to GNU make, a will not be understood by other versions of make. Other make programs look for `makefile' and `Makefile', but not `GNUmakefile'.

If make finds none of these names, it does not use any makefile. Then you must specify a goal with a command argument, and make will attempt to figure out how to remake it using only its built-in implicit rules. See section Using Implicit Rules.

If you want to use a nonstandard name for your makefile, you can specify the makefile name with the `-f' or `--file' option. The arguments `-f name' or `--file=name' tell make to read the file name as the makefile. If you use more than one `-f' or `--file' option, you can specify several makefiles. All the makefiles are effectively concatenated in the order specified. The default makefile names `GNUmakefile', `makefile' and `Makefile' are not checked automatically if you specify `-f' or `--file'.

Including Other Makefiles

The include directive tells make to suspend reading the current makefile and read one or more other makefiles before continuing. The directive is a line in the makefile that looks like this:

```
include filenames...
```

filenames can contain shell file name patterns.

Extra spaces are allowed and ignored at the beginning of the line, but a tab is not allowed. (If the line begins with a tab, it will be considered a command line.) Whitespace is required between include and the file names, and between file names; extra whitespace is ignored there and at the end of the directive. A comment starting with `#' is allowed at the end of the line. If the file names contain any variable or function references, they are expanded. See section How to Use Variables.

For example, if you have three `.mk' files, `a.mk', `b.mk', and `c.mk', and $(bar) expands to bish bash, then the following expression

```
include foo *.mk $(bar)
```

is equivalent to

```
include foo a.mk b.mk c.mk bish bash
```

When make processes an include directive, it suspends reading of the containing makefile and reads from each listed file in turn. When that is finished, make resumes reading the makefile in which the directive appears.

One occasion for using include directives is when several programs, handled by individual makefiles in various directories, need to use a common set of variable definitions (see section Setting Variables) or

pattern rules (see section Defining and Redefining Pattern Rules).

Another such occasion is when you want to generate dependencies from source files automatically; the dependencies can be put in a file that is included by the main makefile. This practice is generally cleaner than that of somehow appending the dependencies to the end of the main makefile as has been traditionally done with other versions of `make`. See section Generating Dependencies Automatically.

If the specified name does not start with a slash, and the file is not found in the current directory, several other directories are searched. First, any directories you have specified with the `-I' or `--include-dir' option are searched (see section Summary of Options). Then the following directories (if they exist) are searched, in this order: `prefix/include' (normally `/usr/local/include') `/usr/gnu/include', `/usr/local/include', `/usr/include'.

If an included makefile cannot be found in any of these directories, a warning message is generated, but it is not an immediately fatal error; processing of the makefile containing the `include` continues. Once it has finished reading makefiles, `make` will try to remake any that are out of date or don't exist. See section How Makefiles Are Remade. Only after it has tried to find a way to remake a makefile and failed, will `make` diagnose the missing makefile as a fatal error.

If you want `make` to simply ignore a makefile which does not exist and cannot be remade, with no error message, use the `-include` directive instead of `include`, like this:

```
-include filenames...
```

This is acts like `include` in every way except that there is no error (not even a warning) if any of the filenames do not exist.

The Variable `MAKEFILES`

If the environment variable `MAKEFILES` is defined, `make` considers its value as a list of names (separated by whitespace) of additional makefiles to be read before the others. This works much like the `include` directive: various directories are searched for those files (see section Including Other Makefiles). In addition, the default goal is never taken from one of these makefiles and it is not an error if the files listed in `MAKEFILES` are not found.

The main use of `MAKEFILES` is in communication between recursive invocations of `make` (see section Recursive Use of `make`}). It usually is not desirable to set the environment variable before a top-level invocation of `make`, because it is usually better not to mess with a makefile from outside. However, if you are running `make` without a specific makefile, a makefile in `MAKEFILES` can do useful things to help the built-in implicit rules work better, such as defining search paths (see section Searching Directories for Dependencies).

Some users are tempted to set `MAKEFILES` in the environment automatically on login, and program makefiles to expect this to be done. This is a very bad idea, because such makefiles will fail to work if run by anyone else. It is much better to write explicit `include` directives in the makefiles. See section Including Other Makefiles.

How Makefiles Are Remade

Sometimes makefiles can be remade from other files, such as RCS or SCCS files. If a makefile can be remade from other files, you probably want `make` to get an up-to-date version of the makefile to read in.

To this end, after reading in all makefiles, `make` will consider each as a goal target and attempt to update it. If a makefile has a rule which says how to update it (found either in that very makefile or in another one) or if an implicit rule applies to it (see section Using Implicit Rules), it will be updated if necessary. After all makefiles have been checked, if any have actually been changed, `make` starts with a clean slate and reads all the makefiles over again. (It will also attempt to update each of them over again, but normally this will not change them again, since they are already up to date.)

If the makefiles specify a double-colon rule to remake a file with commands but no dependencies, that file will always be remade (see section Double-Colon Rules). In the case of makefiles, a makefile that has a double-colon rule with commands but no dependencies will be remade every time `make` is run, and then again after `make` starts over and reads the makefiles in again. This would cause an infinite loop: `make` would constantly remake the makefile, and never do anything else. So, to avoid this, `make` will **not** attempt to remake makefiles which are specified as double-colon targets but have no dependencies.

If you do not specify any makefiles to be read with `-f' or `--file' options, `make` will try the default makefile names; see section What Name to Give Your Makefile. Unlike makefiles explicitly requested with `-f' or `--file' options, `make` is not certain that these makefiles should exist. However, if a default makefile does not exist but can be created by running `make` rules, you probably want the rules to be run so that the makefile can be used.

Therefore, if none of the default makefiles exists, `make` will try to make each of them in the same order in which they are searched for (see section What Name to Give Your Makefile) until it succeeds in making one, or it runs out of names to try. Note that it is not an error if `make` cannot find or make any makefile; a makefile is not always necessary.

When you use the `-t' or `--touch' option (see section Instead of Executing the Commands), you would not want to use an out-of-date makefile to decide which targets to touch. So the `-t' option has no effect on updating makefiles; they are really updated even if `-t' is specified. Likewise, `-q' (or `--question') and `-n' (or `--just-print') do not prevent updating of makefiles, because an out-of-date makefile would result in the wrong output for other targets. Thus, `make -f mfile -n foo' will update `mfile', read it in, and then print the commands to update `foo' and its dependencies without running them. The commands printed for `foo' will be those specified in the updated contents of `mfile'.

However, on occasion you might actually wish to prevent updating of even the makefiles. You can do this by specifying the makefiles as goals in the command line as well as specifying them as makefiles. When the makefile name is specified explicitly as a goal, the options `-t' and so on do apply to them.

Thus, `make -f mfile -n mfile foo' would read the makefile `mfile', print the commands needed to update it without actually running them, and then print the commands needed to update `foo' without running them. The commands for `foo' will be those specified by the existing contents of `mfile'.

Overriding Part of Another Makefile

Sometimes it is useful to have a makefile that is mostly just like another makefile. You can often use the `include' directive to include one in the other, and add more targets or variable definitions. However, if the two makefiles give different commands for the same target, `make` will not let you just do this. But there is another way.

In the containing makefile (the one that wants to include the other), you can use a match-anything pattern rule to say that to remake any target that cannot be made from the information in the containing makefile, `make` should look in another makefile. See section Defining and Redefining Pattern Rules, for more information on pattern rules.

For example, if you have a makefile called `Makefile' that says how to make the target `foo' (and other targets), you can write a makefile called `GNUmakefile' that contains:

```
foo:
        frobnicate > foo

%: force
        @$(MAKE) -f Makefile $@
force: ;
```

If you say `make foo', `make` will find `GNUmakefile', read it, and see that to make `foo', it needs to run the command `frobnicate > foo'. If you say `make bar', `make` will find no way to make `bar' in `GNUmakefile', so it will use the commands from the pattern rule: `make -f Makefile bar'. If `Makefile' provides a rule for updating `bar', `make` will apply the rule. And likewise for any other target that `GNUmakefile' does not say how to make.

The way this works is that the pattern rule has a pattern of just `%', so it matches any target whatever. The rule specifies a dependency `force', to guarantee that the commands will be run even if the target file already exists. We give `force' target empty commands to prevent `make` from searching for an implicit rule to build it--otherwise it would apply the same match-anything rule to `force' itself and create a dependency loop!

Go to the previous, next section.

Go to the previous, next section.

Writing Rules

A rule appears in the makefile and says when and how to remake certain files, called the rule's targets (most often only one per rule). It lists the other files that are the dependencies of the target, and commands to use to create or update the target.

The order of rules is not significant, except for determining the default goal: the target for `make` to consider, if you do not otherwise specify one. The default goal is the target of the first rule in the first makefile. If the first rule has multiple targets, only the first target is taken as the default. There are two exceptions: a target starting with a period is not a default unless it contains one or more slashes, `/', as well; and, a target that defines a pattern rule has no effect on the default goal. (See section Defining and Redefining Pattern Rules.)

Therefore, we usually write the makefile so that the first rule is the one for compiling the entire program or all the programs described by the makefile (often with a target called `all'). See section Arguments to Specify the Goals.

Rule Syntax

In general, a rule looks like this:

```
targets : dependencies
        command
        ...
```

or like this:

```
targets : dependencies ; command
        command
        ...
```

The targets are file names, separated by spaces. Wildcard characters may be used (see section Using Wildcard Characters in File Names) and a name of the form `a'(m) represents member m in archive file a (see section Archive Members as Targets). Usually there is only one target per rule, but occasionally there is a reason to have more (see section Multiple Targets in a Rule).

The command lines start with a tab character. The first command may appear on the line after the dependencies, with a tab character, or may appear on the same line, with a semicolon. Either way, the effect is the same. See section Writing the Commands in Rules.

Because dollar signs are used to start variable references, if you really want a dollar sign in a rule you must write two of them, `$$' (see section How to Use Variables). You may split a long line by inserting a

backslash followed by a newline, but this is not required, as `make` places no limit on the length of a line in a makefile.

A rule tells `make` two things: when the targets are out of date, and how to update them when necessary.

The criterion for being out of date is specified in terms of the dependencies, which consist of file names separated by spaces. (Wildcards and archive members (see section Using `make` to Update Archive Files) are allowed here too.) A target is out of date if it does not exist or if it is older than any of the dependencies (by comparison of last-modification times). The idea is that the contents of the target file are computed based on information in the dependencies, so if any of the dependencies changes, the contents of the existing target file are no longer necessarily valid.

How to update is specified by commands. These are lines to be executed by the shell (normally `sh'), but with some extra features (see section Writing the Commands in Rules).

Using Wildcard Characters in File Names

A single file name can specify many files using wildcard characters. The wildcard characters in `make` are `*', `?' and `[...]', the same as in the Bourne shell. For example, `` `*.c' `` specifies a list of all the files (in the working directory) whose names end in `.c'.

The character `~' at the beginning of a file name also has special significance. If alone, or followed by a slash, it represents your home directory. For example `` `~/bin' `` expands to `` `/home/you/bin' ``. If the `~' is followed by a word, the string represents the home directory of the user named by that word. For example `` `~john/bin' `` expands to `` `/home/john/bin' ``.

Wildcard expansion happens automatically in targets, in dependencies, and in commands (where the shell does the expansion). In other contexts, wildcard expansion happens only if you request it explicitly with the `wildcard` function.

The special significance of a wildcard character can be turned off by preceding it with a backslash. Thus, `` `foo\*bar' `` would refer to a specific file whose name consists of `foo', an asterisk, and `bar'.

Wildcard Examples

Wildcards can be used in the commands of a rule, where they are expanded by the shell. For example, here is a rule to delete all the object files:

```
clean:
        rm -f *.o
```

Wildcards are also useful in the dependencies of a rule. With the following rule in the makefile, `make print' will print all the `.c' files that have changed since the last time you printed them:

```
print: *.c
```


```
        lpr -p $?
        touch print
```

This rule uses `print' as an empty target file; see section Empty Target Files to Record Events. (The automatic variable `$?' is used to print only those files that have changed; see section Automatic Variables.)

Wildcard expansion does not happen when you define a variable. Thus, if you write this:

```
objects = *.o
```

then the value of the variable `objects` is the actual string `*.o'. However, if you use the value of `objects` in a target, dependency or command, wildcard expansion will take place at that time. To set `objects` to the expansion, instead use:

```
objects := $(wildcard *.o)
```

See section The Function `wildcard`.

Pitfalls of Using Wildcards

Now here is an example of a naive way of using wildcard expansion, that does not do what you would intend. Suppose you would like to say that the executable file `foo' is made from all the object files in the directory, and you write this:

```
objects = *.o

foo : $(objects)
        cc -o foo $(CFLAGS) $(objects)
```

The value of `objects` is the actual string `*.o'. Wildcard expansion happens in the rule for `foo', so that each *existing* `.o' file becomes a dependency of `foo' and will be recompiled if necessary.

But what if you delete all the `.o' files? When a wildcard matches no files, it is left as it is, so then `foo' will depend on the oddly-named file `*.o'. Since no such file is likely to exist, `make` will give you an error saying it cannot figure out how to make `*.o'. This is not what you want!

Actually it is possible to obtain the desired result with wildcard expansion, but you need more sophisticated techniques, including the `wildcard` function and string substitution. These are described in the following section.

The Function `wildcard`

Wildcard expansion happens automatically in rules. But wildcard expansion does not normally take place when a variable is set, or inside the arguments of a function. If you want to do wildcard expansion in such places, you need to use the `wildcard` function, like this:


```
$(wildcard pattern...)
```

This string, used anywhere in a makefile, is replaced by a space-separated list of names of existing files that match one of the given file name patterns. If no existing file name matches a pattern, then that pattern is omitted from the output of the `wildcard` function. Note that this is different from how unmatched wildcards behave in rules, where they are used verbatim rather than ignored (see section Pitfalls of Using Wildcards).

One use of the `wildcard` function is to get a list of all the C source files in a directory, like this:

```
$(wildcard *.c)
```

We can change the list of C source files into a list of object files by replacing the `.o' suffix with `.c' in the result, like this:

```
$(patsubst %.c,%.o,$(wildcard *.c))
```

(Here we have used another function, `patsubst`. See section Functions for String Substitution and Analysis.)

Thus, a makefile to compile all C source files in the directory and then link them together could be written as follows:

```
objects := $(patsubst %.c,%.o,$(wildcard *.c))

foo : $(objects)
        cc -o foo $(objects)
```

(This takes advantage of the implicit rule for compiling C programs, so there is no need to write explicit rules for compiling the files. See section The Two Flavors of Variables, for an explanation of `:=', which is a variant of `='.)

Searching Directories for Dependencies

For large systems, it is often desirable to put sources in a separate directory from the binaries. The directory search features of `make` facilitate this by searching several directories automatically to find a dependency. When you redistribute the files among directories, you do not need to change the individual rules, just the search paths.

VPATH: Search Path for All Dependencies

The value of the `make` variable `VPATH` specifies a list of directories that `make` should search. Most often, the directories are expected to contain dependency files that are not in the current directory; however, `VPATH` specifies a search list that `make` applies for all files, including files which are targets of rules.

Thus, if a file that is listed as a target or dependency does not exist in the current directory, `make` searches the directories listed in `VPATH` for a file with that name. If a file is found in one of them, that file becomes the dependency. Rules may then specify the names of source files in the dependencies as if they all existed in the current directory. See section Writing Shell Commands with Directory Search.

In the `VPATH` variable, directory names are separated by colons or blanks. The order in which directories are listed is the order followed by `make` in its search.

For example,

```
VPATH = src:../headers
```

specifies a path containing two directories, `` `src' `` and `` `../headers' ``, which `make` searches in that order.

With this value of `VPATH`, the following rule,

```
foo.o : foo.c
```

is interpreted as if it were written like this:

```
foo.o : src/foo.c
```

assuming the file `` `foo.c' `` does not exist in the current directory but is found in the directory `` `src' ``.

The `vpath` Directive

Similar to the `VPATH` variable but more selective is the `vpath` directive (note lower case), which allows you to specify a search path for a particular class of file names, those that match a particular pattern. Thus you can supply certain search directories for one class of file names and other directories (or none) for other file names.

There are three forms of the `vpath` directive:

`vpath pattern directories`

Specify the search path directories for file names that match pattern.

The search path, directories, is a list of directories to be searched, separated by colons or blanks, just like the search path used in the `VPATH` variable.

`vpath pattern`

Clear out the search path associated with pattern.

`vpath`

Clear all search paths previously specified with `vpath` directives.

A `vpath` pattern is a string containing a `%' character. The string must match the file name of a dependency that is being searched for, the `%' character matching any sequence of zero or more

characters (as in pattern rules; see section Defining and Redefining Pattern Rules). For example, `%.h` matches files that end in `.h`. (If there is no `%', the pattern must match the dependency exactly, which is not useful very often.)

`%' characters in a `vpath` directive's pattern can be quoted with preceding backslashes (`\'). Backslashes that would otherwise quote `%' characters can be quoted with more backslashes. Backslashes that quote `%' characters or other backslashes are removed from the pattern before it is compared to file names. Backslashes that are not in danger of quoting `%' characters go unmolested.

When a dependency fails to exist in the current directory, if the pattern in a `vpath` directive matches the name of the dependency file, then the directories in that directive are searched just like (and before) the directories in the `VPATH` variable.

For example,

```
vpath %.h ../headers
```

tells `make` to look for any dependency whose name ends in `.h' in the directory `../headers' if the file is not found in the current directory.

If several `vpath` patterns match the dependency file's name, then `make` processes each matching `vpath` directive one by one, searching all the directories mentioned in each directive. `make` handles multiple `vpath` directives in the order in which they appear in the makefile; multiple directives with the same pattern are independent of each other.

Thus,

```
vpath %.c foo
vpath %   blish
vpath %.c bar
```

will look for a file ending in `.c' in `foo', then `blish', then `bar', while

```
vpath %.c foo:bar
vpath %   blish
```

will look for a file ending in `.c' in `foo', then `bar', then `blish'.

Writing Shell Commands with Directory Search

When a dependency is found in another directory through directory search, this cannot change the commands of the rule; they will execute as written. Therefore, you must write the commands with care so that they will look for the dependency in the directory where `make` finds it.

This is done with the automatic variables such as `$^' (see section Automatic Variables). For instance, the value of `$^' is a list of all the dependencies of the rule, including the names of the directories in which they were found, and the value of `$@' is the target. Thus:


```
foo.o : foo.c
        cc -c $(CFLAGS) $^ -o $@
```

(The variable CFLAGS exists so you can specify flags for C compilation by implicit rules; we use it here for consistency so it will affect all C compilations uniformly; see section Variables Used by Implicit Rules.)

Often the dependencies include header files as well, which you do not want to mention in the commands. The automatic variable `$<' is just the first dependency:

```
VPATH = src:../headers
foo.o : foo.c defs.h hack.h
        cc -c $(CFLAGS) $< -o $@
```

Directory Search and Implicit Rules

The search through the directories specified in VPATH or with vpath also happens during consideration of implicit rules (see section Using Implicit Rules).

For example, when a file `foo.o' has no explicit rule, make considers implicit rules, such as the built-in rule to compile `foo.c' if that file exists. If such a file is lacking in the current directory, the appropriate directories are searched for it. If `foo.c' exists (or is mentioned in the makefile) in any of the directories, the implicit rule for C compilation is applied.

The commands of implicit rules normally use automatic variables as a matter of necessity; consequently they will use the file names found by directory search with no extra effort.

Directory Search for Link Libraries

Directory search applies in a special way to libraries used with the linker. This special feature comes into play when you write a dependency whose name is of the form `-lname'. (You can tell something strange is going on here because the dependency is normally the name of a file, and the *file name* of the library looks like `libname.a', not like `-lname'.)

When a dependency's name has the form `-lname', make handles it specially by searching for the file `libname.a' in the current directory, in directories specified by matching vpath search paths and the VPATH search path, and then in the directories `/lib', `/usr/lib', and `prefix/lib' (normally `/usr/local/lib').

For example,

```
foo : foo.c -lcurses
        cc $^ -o $@
```

would cause the command `cc foo.c /usr/lib/libcurses.a -o foo' to be executed when `foo' is older than `foo.c' or than `/usr/lib/libcurses.a'.

Phony Targets

A phony target is one that is not really the name of a file. It is just a name for some commands to be executed when you make an explicit request. There are two reasons to use a phony target: to avoid a conflict with a file of the same name, and to improve performance.

If you write a rule whose commands will not create the target file, the commands will be executed every time the target comes up for remaking. Here is an example:

```
clean:
        rm *.o temp
```

Because the `rm` command does not create a file named `` `clean' ``, probably no such file will ever exist. Therefore, the `rm` command will be executed every time you say `make clean'.

The phony target will cease to work if anything ever does create a file named `` `clean' `` in this directory. Since it has no dependencies, the file `` `clean' `` would inevitably be considered up to date, and its commands would not be executed. To avoid this problem, you can explicitly declare the target to be phony, using the special target `.PHONY` (see section Special Built-in Target Names) as follows:

```
.PHONY : clean
```

Once this is done, `make clean' will run the commands regardless of whether there is a file named `` `clean' ``.

Since it knows that phony targets do not name actual files that could be remade from other files, `make` skips the implicit rule search for phony targets (see section Using Implicit Rules). This is why declaring a target phony is good for performance, even if you are not worried about the actual file existing.

Thus, you first write the line that states that `clean` is a phony target, then you write the rule, like this:

```
.PHONY: clean
clean:
        rm *.o temp
```

A phony target should not be a dependency of a real target file; if it is, its commands are run every time `make` goes to update that file. As long as a phony target is never a dependency of a real target, the phony target commands will be executed only when the phony target is a specified goal (see section Arguments to Specify the Goals).

Phony targets can have dependencies. When one directory contains multiple programs, it is most convenient to describe all of the programs in one makefile `` `./Makefile' ``. Since the target remade by default will be the first one in the makefile, it is common to make this a phony target named `all' and give it, as dependencies, all the individual programs. For example:

```
all : prog1 prog2 prog3
```


```
.PHONY : all

prog1 : prog1.o utils.o
        cc -o prog1 prog1.o utils.o

prog2 : prog2.o
        cc -o prog2 prog2.o

prog3 : prog3.o sort.o utils.o
        cc -o prog3 prog3.o sort.o utils.o
```

Now you can say just `make' to remake all three programs, or specify as arguments the ones to remake (as in `make prog1 prog3').

When one phony target is a dependency of another, it serves as a subroutine of the other. For example, here `make cleanall' will delete the object files, the difference files, and the file `program':

```
.PHONY: cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff
        rm program

cleanobj :
        rm *.o

cleandiff :
        rm *.diff
```

Rules without Commands or Dependencies

If a rule has no dependencies or commands, and the target of the rule is a nonexistent file, then `make` imagines this target to have been updated whenever its rule is run. This implies that all targets depending on this one will always have their commands run.

An example will illustrate this:

```
clean: FORCE
        rm $(objects)
FORCE:
```

Here the target `FORCE' satisfies the special conditions, so the target `clean' that depends on it is forced to run its commands. There is nothing special about the name `FORCE', but that is one name commonly used this way.

As you can see, using `FORCE' this way has the same results as using `.PHONY: clean'.

Using `.PHONY' is more explicit and more efficient. However, other versions of `make' do not suppo `.PHONY'; thus `FORCE' appears in many makefiles. See section Phony Targets.

Empty Target Files to Record Events

The empty target is a variant of the phony target; it is used to hold commands for an action that you request explicitly from time to time. Unlike a phony target, this target file can really exist; but the file's contents do not matter, and usually are empty.

The purpose of the empty target file is to record, with its last-modification time, when the rule's commands were last executed. It does so because one of the commands is a `touch' command to update the target file.

The empty target file must have some dependencies. When you ask to remake the empty target, the commands are executed if any dependency is more recent than the target; in other words, if a dependency has changed since the last time you remade the target. Here is an example:

```
print: foo.c bar.c
        lpr -p $?
        touch print
```

With this rule, `make print' will execute the `lpr' command if either source file has changed since the last `make print'. The automatic variable `$?' is used to print only those files that have changed (see section Automatic Variables).

Special Built-in Target Names

Certain names have special meanings if they appear as targets.

`.PHONY`

The dependencies of the special target `.PHONY` are considered to be phony targets. When it is time to consider such a target, `make` will run its commands unconditionally, regardless of whether a file with that name exists or what its last-modification time is. See section Phony Targets.

`.SUFFIXES`

The dependencies of the special target `.SUFFIXES` are the list of suffixes to be used in checking for suffix rules. See section Old-Fashioned Suffix Rules.

`.DEFAULT`

The commands specified for `.DEFAULT` are used for any target for which no rules are found (either explicit rules or implicit rules). See section Defining Last-Resort Default Rules. If `.DEFAULT` commands are specified, every file mentioned as a dependency, but not as a target in a

rule, will have these commands executed on its behalf. See section Implicit Rule Search Algorithm.

`.PRECIOUS`

The targets which `.PRECIOUS` depends on are given the following special treatment: if `make` is killed or interrupted during the execution of their commands, the target is not deleted. See section Interrupting or Killing `make`. Also, if the target is an intermediate file, it will not be deleted after it is no longer needed, as is normally done. See section Chains of Implicit Rules.

You can also list the target pattern of an implicit rule (such as `%.o') as a dependency file of the special target `.PRECIOUS` to preserve intermediate files created by rules whose target patterns match that file's name.

`.IGNORE`

If you specify dependencies for `.IGNORE`, then `make` will ignore errors in execution of the commands run for those particular files. The commands for `.IGNORE` are not meaningful.

If mentioned as a target with no dependencies, `.IGNORE` says to ignore errors in execution of commands for all files. This usage of `.IGNORE' is supported only for historical compatibility. Since this affects every command in the makefile, it is not very useful; we recommend you use the more selective ways to ignore errors in specific commands. See section Errors in Commands.

`.SILENT`

If you specify dependencies for `.SILENT`, then `make` will not the print commands to remake those particular files before executing them. The commands for `.SILENT` are not meaningful.

If mentioned as a target with no dependencies, `.SILENT` says not to print any commands before executing them. This usage of `.SILENT' is supported only for historical compatibility. We recommend you use the more selective ways to silence specific commands. See section Command Echoing. If you want to silence all commands for a particular run of `make`, use the `-s' or `--silent' option (see section Summary of Options).

`.EXPORT_ALL_VARIABLES`

Simply by being mentioned as a target, this tells `make` to export all variables to child processes by default. See section Communicating Variables to a Sub-`make`}.

Any defined implicit rule suffix also counts as a special target if it appears as a target, and so does the concatenation of two suffixes, such as `.c.o'. These targets are suffix rules, an obsolete way of defining implicit rules (but a way still widely used). In principle, any target name could be special in this way if you break it in two and add both pieces to the suffix list. In practice, suffixes normally begin with `.', so these special target names also begin with `.'. See section Old-Fashioned Suffix Rules.

Multiple Targets in a Rule

A rule with multiple targets is equivalent to writing many rules, each with one target, and all identical aside from that. The same commands apply to all the targets, but their effects may vary because you can substitute the actual target name into the command using `$@'. The rule contributes the same dependencies to all the targets also.

This is useful in two cases.

- You want just dependencies, no commands. For example:

  ```
  kbd.o command.o files.o: command.h
  ```

  gives an additional dependency to each of the three object files mentioned.
- Similar commands work for all the targets. The commands do not need to be absolutely identical, since the automatic variable `$@' can be used to substitute the particular target to be remade into the commands (see section Automatic Variables). For example:

  ```
  bigoutput littleoutput : text.g
          generate text.g -$(subst output,,$@) > $@
  ```

  is equivalent to

  ```
  bigoutput : text.g
          generate text.g -big > bigoutput
  littleoutput : text.g
          generate text.g -little > littleoutput
  ```

  Here we assume the hypothetical program `generate` makes two types of output, one if given `-big' and one if given `-little'. See section Functions for String Substitution and Analysis, for an explanation of the `subst` function.

Suppose you would like to vary the dependencies according to the target, much as the variable `$@' allows you to vary the commands. You cannot do this with multiple targets in an ordinary rule, but you can do it with a static pattern rule. See section Static Pattern Rules.

Multiple Rules for One Target

One file can be the target of several rules. All the dependencies mentioned in all the rules are merged into one list of dependencies for the target. If the target is older than any dependency from any rule, the commands are executed.

There can only be one set of commands to be executed for a file. If more than one rule gives commands for the same file, `make` uses the last set given and prints an error message. (As a special case, if the file's name begins with a dot, no error message is printed. This odd behavior is only for compatibility with

other implementations of `make`.) There is no reason to write your makefiles this way; that is why `ma` gives you an error message.

An extra rule with just dependencies can be used to give a few extra dependencies to many files at once. For example, one usually has a variable named `objects` containing a list of all the compiler output files in the system being made. An easy way to say that all of them must be recompiled if `` `config.h' `` changes is to write the following:

```
objects = foo.o bar.o
foo.o : defs.h
bar.o : defs.h test.h
$(objects) : config.h
```

This could be inserted or taken out without changing the rules that really specify how to make the object files, making it a convenient form to use if you wish to add the additional dependency intermittently.

Another wrinkle is that the additional dependencies could be specified with a variable that you set with a command argument to `make` (see section Overriding Variables). For example,

```
extradeps=
$(objects) : $(extradeps)
```

means that the command `make extradeps=foo.h' will consider `` `foo.h' `` as a dependency of each object file, but plain `make' will not.

If none of the explicit rules for a target has commands, then `make` searches for an applicable implicit rule to find some commands see section Using Implicit Rules).

Static Pattern Rules

Static pattern rules are rules which specify multiple targets and construct the dependency names for each target based on the target name. They are more general than ordinary rules with multiple targets because the targets do not have to have identical dependencies. Their dependencies must be *analogous*, but not necessarily *identical*.

Syntax of Static Pattern Rules

Here is the syntax of a static pattern rule:

```
targets ...: target-pattern: dep-patterns ...
        commands
        ...
```

The targets list specifies the targets that the rule applies to. The targets can contain wildcard characters, just like the targets of ordinary rules (see section Using Wildcard Characters in File Names).

The target-pattern and dep-patterns say how to compute the dependencies of each target. Each target is matched against the target-pattern to extract a part of the target name, called the stem. This stem is substituted into each of the dep-patterns to make the dependency names (one from each dep-pattern).

Each pattern normally contains the character `%' just once. When the target-pattern matches a target, the `%' can match any part of the target name; this part is called the stem. The rest of the pattern must match exactly. For example, the target `foo.o' matches the pattern `%.o', with `foo' as the stem. The targets `foo.c' and `foo.out' do not match that pattern.

The dependency names for each target are made by substituting the stem for the `%' in each dependency pattern. For example, if one dependency pattern is `%.c', then substitution of the stem `foo' gives the dependency name `foo.c'. It is legitimate to write a dependency pattern that does not contain `%'; then this dependency is the same for all targets.

`%' characters in pattern rules can be quoted with preceding backslashes (`\'). Backslashes that would otherwise quote `%' characters can be quoted with more backslashes. Backslashes that quote `%' characters or other backslashes are removed from the pattern before it is compared to file names or has a stem substituted into it. Backslashes that are not in danger of quoting `%' characters go unmolested. For example, the pattern `the\%weird\\%pattern\\' has `the%weird\' preceding the operative `%' character, and `pattern\\' following it. The final two backslashes are left alone because they cannot affect any `%' character.

Here is an example, which compiles each of `foo.o' and `bar.o' from the corresponding `.c' file:

```
objects = foo.o bar.o

$(objects): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
```

Here `$<' is the automatic variable that holds the name of the dependency and `$@' is the automatic variable that holds the name of the target; see section Automatic Variables.

Each target specified must match the target pattern; a warning is issued for each target that does not. If you have a list of files, only some of which will match the pattern, you can use the `filter` function to remove nonmatching file names (see section Functions for String Substitution and Analysis):

```
files = foo.elc bar.o lose.o

$(filter %.o,$(files)): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
$(filter %.elc,$(files)): %.elc: %.el
        emacs -f batch-byte-compile $<
```

In this example the result of `$(filter %.o,$(files))' is `bar.o lose.o', and the first static pattern rule causes each of these object files to be updated by compiling the corresponding C source file. The result of `$(filter %.elc,$(files))' is `foo.elc', so that file is made from `foo.el'.

Another example shows how to use `$*` in static pattern rules:

```
bigoutput littleoutput : %output : text.g
        generate text.g -$* > $@
```

When the `generate` command is run, `$*` will expand to the stem, either `big' or `little'.

Static Pattern Rules versus Implicit Rules

A static pattern rule has much in common with an implicit rule defined as a pattern rule (see section Defining and Redefining Pattern Rules). Both have a pattern for the target and patterns for constructing the names of dependencies. The difference is in how `make` decides *when* the rule applies.

An implicit rule *can* apply to any target that matches its pattern, but it *does* apply only when the target has no commands otherwise specified, and only when the dependencies can be found. If more than one implicit rule appears applicable, only one applies; the choice depends on the order of rules.

By contrast, a static pattern rule applies to the precise list of targets that you specify in the rule. It cannot apply to any other target and it invariably does apply to each of the targets specified. If two conflicting rules apply, and both have commands, that's an error.

The static pattern rule can be better than an implicit rule for these reasons:

- You may wish to override the usual implicit rule for a few files whose names cannot be categorized syntactically but can be given in an explicit list.
- If you cannot be sure of the precise contents of the directories you are using, you may not be sure which other irrelevant files might lead `make` to use the wrong implicit rule. The choice might depend on the order in which the implicit rule search is done. With static pattern rules, there is no uncertainty: each rule applies to precisely the targets specified.

Double-Colon Rules

Double-colon rules are rules written with `::' instead of `:' after the target names. They are handled differently from ordinary rules when the same target appears in more than one rule.

When a target appears in multiple rules, all the rules must be the same type: all ordinary, or all double-colon. If they are double-colon, each of them is independent of the others. Each double-colon rule's commands are executed if the target is older than any dependencies of that rule. This can result in executing none, any, or all of the double-colon rules.

Double-colon rules with the same target are in fact completely separate from one another. Each double-colon rule is processed individually, just as rules with different targets are processed.

The double-colon rules for a target are executed in the order they appear in the makefile. However, the cases where double-colon rules really make sense are those where the order of executing the commands would not matter.

Double-colon rules are somewhat obscure and not often very useful; they provide a mechanism for cases

in which the method used to update a target differs depending on which dependency files caused the update, and such cases are rare.

Each double-colon rule should specify commands; if it does not, an implicit rule will be used if one applies. See section Using Implicit Rules.

Generating Dependencies Automatically

In the makefile for a program, many of the rules you need to write often say only that some object file depends on some header file. For example, if `main.c' uses `defs.h' via an `#include`, you would write:

```
main.o: defs.h
```

You need this rule so that `make` knows that it must remake `main.o' whenever `defs.h' changes. You can see that for a large program you would have to write dozens of such rules in your makefile. And, you must always be very careful to update the makefile every time you add or remove an `#include`.

To avoid this hassle, most modern C compilers can write these rules for you, by looking at the `#include` lines in the source files. Usually this is done with the `-M' option to the compiler. For example, the command:

```
cc -M main.c
```

generates the output:

```
main.o : main.c defs.h
```

Thus you no longer have to write all those rules yourself. The compiler will do it for you.

Note that such a dependency constitutes mentioning `main.o' in a makefile, so it can never be considered an intermediate file by implicit rule search. This means that `make` won't ever remove the file after using it; see section Chains of Implicit Rules.

With old `make` programs, it was traditional practice to use this compiler feature to generate dependencies on demand with a command like `make depend'. That command would create a file `depend' containing all the automatically-generated dependencies; then the makefile could use `include` to read them in (see section Including Other Makefiles).

In GNU `make`, the feature of remaking makefiles makes this practice obsolete--you need never tell `make` explicitly to regenerate the dependencies, because it always regenerates any makefile that is out of date. See section How Makefiles Are Remade.

The practice we recommend for automatic dependency generation is to have one makefile corresponding to each source file. For each source file `name.c' there is a makefile `name.d' which lists what files the object file `name.o' depends on. That way only the source files that have changed need to be

rescanned to produce the new dependencies.

Here is the pattern rule to generate a file of dependencies (i.e., a makefile) called `name.d' from a C source file called `name.c':

```
%.d: %.c
        $(SHELL) -ec '$(CC) -M $(CPPFLAGS) $< \
                      | sed '\"s/$*\\.o[ :]*/& $@/g'\" > $@'
```

See section Defining and Redefining Pattern Rules, for information on defining pattern rules. The `-e' flag to the shell makes it exit immediately if the `$(CC)` command fails (exits with a nonzero status). Normally the shell exits with the status of the last command in the pipeline (`sed` in this case), so `make` would not notice a nonzero status from the compiler.

With the GNU C compiler, you may wish to use the `-MM' flag instead of `-M'. This omits dependencies on system header files. See section 'Options Controlling the Preprocessor' in Using GNU CC, for details.

The purpose of the `sed` command is to translate (for example):

```
main.o : main.c defs.h
```

into:

```
main.o main.d : main.c defs.h
```

This makes each `.d' file depend on all the source and header files that the corresponding `.o' file depends on. `make` then knows it must regenerate the dependencies whenever any of the source or header files changes.

Once you've defined the rule to remake the `.d' files, you then use the `include` directive to read them all in. See section Including Other Makefiles. For example:

```
sources = foo.c bar.c

include $(sources:.c=.d)
```

(This example uses a substitution variable reference to translate the list of source files `foo.c bar.c' into a list of dependency makefiles, `foo.d bar.d'. See section Substitution References, for full information on substitution references.) Since the `.d' files are makefiles like any others, `make` will remake them as necessary with no further work from you. See section How Makefiles Are Remade.

Go to the previous, next section.

Go to the previous, next section.

Writing the Commands in Rules

The commands of a rule consist of shell command lines to be executed one by one. Each command line must start with a tab, except that the first command line may be attached to the target-and-dependencies line with a semicolon in between. Blank lines and lines of just comments may appear among the command lines; they are ignored. (But beware, an apparently "blank" line that begins with a tab is *not* blank! It is an empty command; see section Using Empty Commands.)

Users use many different shell programs, but commands in makefiles are always interpreted by `/bin/sh' unless the makefile specifies otherwise. See section Command Execution.

The shell that is in use determines whether comments can be written on command lines, and what syntax they use. When the shell is `/bin/sh', a `#' starts a comment that extends to the end of the line. The `#' does not have to be at the beginning of a line. Text on a line before a `#' is not part of the comment.

Command Echoing

Normally `make` prints each command line before it is executed. We call this echoing because it gives the appearance that you are typing the commands yourself.

When a line starts with `@', the echoing of that line is suppressed. The `@' is discarded before the command is passed to the shell. Typically you would use this for a command whose only effect is to print something, such as an `echo` command to indicate progress through the makefile:

```
@echo About to make distribution files
```

When `make` is given the flag `-n' or `--just-print', echoing is all that happens, no execution. See section Summary of Options. In this case and only this case, even the commands starting with `@' are printed. This flag is useful for finding out which commands `make` thinks are necessary without actually doing them.

The `-s' or `--silent' flag to `make` prevents all echoing, as if all commands started with `@'. A rule in the makefile for the special target `.SILENT` without dependencies has the same effect (see section Special Built-in Target Names). `.SILENT` is essentially obsolete since `@' is more flexible.

Command Execution

When it is time to execute commands to update a target, they are executed by making a new subshell for each line. (In practice, `make` may take shortcuts that do not affect the results.)

Please note: this implies that shell commands such as `cd` that set variables local to each process will not affect the following command lines. If you want to use `cd` to affect the next command, put the two on a

single line with a semicolon between them. Then `make` will consider them a single command and pa them, together, to a shell which will execute them in sequence. For example:

```
foo : bar/lose
        cd bar; gobble lose > ../foo
```

If you would like to split a single shell command into multiple lines of text, you must use a backslash at the end of all but the last subline. Such a sequence of lines is combined into a single line, by deleting the backslash-newline sequences, before passing it to the shell. Thus, the following is equivalent to the preceding example:

```
foo : bar/lose
        cd bar;  \
        gobble lose > ../foo
```

The program used as the shell is taken from the variable `SHELL`. By default, the program `` `/bin/sh' `` is used.

Unlike most variables, the variable `SHELL` is never set from the environment. This is because the `SHELL` environment variable is used to specify your personal choice of shell program for interactive use. It would be very bad for personal choices like this to affect the functioning of makefiles. See section Variables from the Environment.

Parallel Execution

GNU `make` knows how to execute several commands at once. Normally, `make` will execute only one command at a time, waiting for it to finish before executing the next. However, the `-j' or `--jobs' option tells `make` to execute many commands simultaneously.

If the `-j' option is followed by an integer, this is the number of commands to execute at once; this is called the number of job slots. If there is nothing looking like an integer after the `-j' option, there is no limit on the number of job slots. The default number of job slots is one, which means serial execution (one thing at a time).

One unpleasant consequence of running several commands simultaneously is that output from all of the commands comes when the commands send it, so messages from different commands may be interspersed.

Another problem is that two processes cannot both take input from the same device; so to make sure that only one command tries to take input from the terminal at once, `make` will invalidate the standard input streams of all but one running command. This means that attempting to read from standard input will usually be a fatal error (a `Broken pipe' signal) for most child processes if there are several.

It is unpredictable which command will have a valid standard input stream (which will come from the terminal, or wherever you redirect the standard input of `make`). The first command run will always get it first, and the first command started after that one finishes will get it next, and so on.

We will change how this aspect of `make` works if we find a better alternative. In the mean time, you should not rely on any command using standard input at all if you are using the parallel execution feature; but if you are not using this feature, then standard input works normally in all commands.

If a command fails (is killed by a signal or exits with a nonzero status), and errors are not ignored for that command (see section Errors in Commands), the remaining command lines to remake the same target will not be run. If a command fails and the `-k' or `--keep-going' option was not given (see section Summary of Options), `make` aborts execution. If make terminates for any reason (including a signal) with child processes running, it waits for them to finish before actually exiting.

When the system is heavily loaded, you will probably want to run fewer jobs than when it is lightly loaded. You can use the `-l' option to tell `make` to limit the number of jobs to run at once, based on the load average. The `-l' or `--max-load' option is followed by a floating-point number. For example,

```
-l 2.5
```

will not let `make` start more than one job if the load average is above 2.5. The `-l' option with no following number removes the load limit, if one was given with a previous `-l' option.

More precisely, when `make` goes to start up a job, and it already has at least one job running, it checks the current load average; if it is not lower than the limit given with `-l', `make` waits until the load average goes below that limit, or until all the other jobs finish.

By default, there is no load limit.

Errors in Commands

After each shell command returns, `make` looks at its exit status. If the command completed successfully, the next command line is executed in a new shell; after the last command line is finished, the rule is finished.

If there is an error (the exit status is nonzero), `make` gives up on the current rule, and perhaps on all rules.

Sometimes the failure of a certain command does not indicate a problem. For example, you may use the `mkdir` command to ensure that a directory exists. If the directory already exists, `mkdir` will report an error, but you probably want `make` to continue regardless.

To ignore errors in a command line, write a `-' at the beginning of the line's text (after the initial tab). The `-' is discarded before the command is passed to the shell for execution.

For example,

```
clean:
        -rm -f *.o
```

This causes `rm` to continue even if it is unable to remove a file.

When you run `make` with the `-i' or `--ignore-errors' flag, errors are ignored in all commands of all rules. A rule in the makefile for the special target `.IGNORE` has the same effect, if there are no dependencies. These ways of ignoring errors are obsolete because `-' is more flexible.

When errors are to be ignored, because of either a `-' or the `-i' flag, `make` treats an error return just like success, except that it prints out a message that tells you the status code the command exited with, and says that the error has been ignored.

When an error happens that `make` has not been told to ignore, it implies that the current target cannot be correctly remade, and neither can any other that depends on it either directly or indirectly. No further commands will be executed for these targets, since their preconditions have not been achieved.

Normally `make` gives up immediately in this circumstance, returning a nonzero status. However, if the `-k' or `--keep-going' flag is specified, `make` continues to consider the other dependencies of the pending targets, remaking them if necessary, before it gives up and returns nonzero status. For example, after an error in compiling one object file, `make -k' will continue compiling other object files even though it already knows that linking them will be impossible. See section Summary of Options.

The usual behavior assumes that your purpose is to get the specified targets up to date; once `make` learns that this is impossible, it might as well report the failure immediately. The `-k' option says that the real purpose is to test as many of the changes made in the program as possible, perhaps to find several independent problems so that you can correct them all before the next attempt to compile. This is why Emacs' `compile` command passes the `-k' flag by default.

Usually when a command fails, if it has changed the target file at all, the file is corrupted and cannot be used--or at least it is not completely updated. Yet the file's timestamp says that it is now up to date, so the next time `make` runs, it will not try to update that file. The situation is just the same as when the command is killed by a signal; see section Interrupting or Killing `make`. So generally the right thing to do is to delete the target file if the command fails after beginning to change the file. `make` will do this if `.DELETE_ON_ERROR` appears as a target. This is almost always what you want `make` to do, but it is not historical practice; so for compatibility, you must explicitly request it.

Interrupting or Killing `make`

If `make` gets a fatal signal while a command is executing, it may delete the target file that the command was supposed to update. This is done if the target file's last-modification time has changed since `make` first checked it.

The purpose of deleting the target is to make sure that it is remade from scratch when `make` is next run. Why is this? Suppose you type Ctrl-c while a compiler is running, and it has begun to write an object file `foo.o'. The Ctrl-c kills the compiler, resulting in an incomplete file whose last-modification time is newer than the source file `foo.c'. But `make` also receives the Ctrl-c signal and deletes this incomplete file. If `make` did not do this, the next invocation of `make` would think that `foo.o' did not require updating--resulting in a strange error message from the linker when it tries to link an object file half of which is missing.

You can prevent the deletion of a target file in this way by making the special target `.PRECIOUS` depend on it. Before remaking a target, `make` checks to see whether it appears on the dependencies of `.PRECIOUS`, and thereby decides whether the target should be deleted if a signal happens. Some reasons why you might do this are that the target is updated in some atomic fashion, or exists only to record a modification-time (its contents do not matter), or must exist at all times to prevent other sorts of trouble.

Recursive Use of `make`

Recursive use of `make` means using `make` as a command in a makefile. This technique is useful when you want separate makefiles for various subsystems that compose a larger system. For example, suppose you have a subdirectory `` `subdir' `` which has its own makefile, and you would like the containing directory's makefile to run `make` on the subdirectory. You can do it by writing this:

```
subsystem:
        cd subdir; $(MAKE)
```

or, equivalently, this (see section Summary of Options):

```
subsystem:
        $(MAKE) -C subdir
```

You can write recursive `make` commands just by copying this example, but there are many things to know about how they work and why, and about how the sub-`make` relates to the top-level `make`.

How the `MAKE` Variable Works

Recursive `make` commands should always use the variable `MAKE`, not the explicit command name `make', as shown here:

```
subsystem:
        cd subdir; $(MAKE)
```

The value of this variable is the file name with which `make` was invoked. If this file name was `` `/bin/make' ``, then the command executed is `cd subdir; /bin/make'. If you use a special version of `make` to run the top-level makefile, the same special version will be executed for recursive invocations.

As a special feature, using the variable `MAKE` in the commands of a rule alters the effects of the `-t' (`--touch'), `-n' (`--just-print'), or `-q' (`--question') option. Using the `MAKE` variable has the same effect as using a `+' character at the beginning of the command line. See section Instead of Executing the Commands.

Consider the command `make -t' in the above example. (The `-t' option marks targets as up to date

without actually running any commands; see section Instead of Executing the Commands.) Following the usual definition of `-t', a `make -t' command in the example would create a file named `subsystem' and do nothing else. What you really want it to do is run `cd subdir; make -t'; but that would require executing the command, and `-t' says not to execute commands.

The special feature makes this do what you want: whenever a command line of a rule contains the variable MAKE, the flags `-t', `-n' and `-q' do not apply to that line. Command lines containing MAKE are executed normally despite the presence of a flag that causes most commands not to be run. The usual MAKEFLAGS mechanism passes the flags to the sub-make (see section Communicating Options to a Sub-make}), so your request to touch the files, or print the commands, is propagated to the subsystem.

Communicating Variables to a Sub-make

Variable values of the top-level make can be passed to the sub-make through the environment by explicit request. These variables are defined in the sub-make as defaults, but do not override what is specified in the makefile used by the sub-make makefile unless you use the `-e' switch (see section Summary of Options).

To pass down, or export, a variable, make adds the variable and its value to the environment for running each command. The sub-make, in turn, uses the environment to initialize its table of variable values. See section Variables from the Environment.

Except by explicit request, make exports a variable only if it is either defined in the environment initially or set on the command line, and if its name consists only of letters, numbers, and underscores. Some shells cannot cope with environment variable names consisting of characters other than letters, numbers, and underscores.

The special variables SHELL and MAKEFLAGS are always exported (unless you unexport them). MAKEFILES is exported if you set it to anything.

make automatically passes down variable values that were defined on the command line, by putting them in the MAKEFLAGS variable. See the next section.

Variables are *not* normally passed down if they were created by default by make (see section Variables Used by Implicit Rules). The sub-make will define these for itself.

If you want to export specific variables to a sub-make, use the export directive, like this:

```
export variable ...
```

If you want to *prevent* a variable from being exported, use the unexport directive, like this:

```
unexport variable ...
```

As a convenience, you can define a variable and export it at the same time by doing:

```
export variable = value
```

has the same result as:

```
variable = value
export variable
```

and

```
export variable := value
```

has the same result as:

```
variable := value
export variable
```

Likewise,

```
export variable += value
```

is just like:

```
variable += value
export variable
```

See section Appending More Text to Variables.

You may notice that the `export` and `unexport` directives work in `make` in the same way they work in the shell, `sh`.

If you want all variables to be exported by default, you can use `export` by itself:

```
export
```

This tells `make` that variables which are not explicitly mentioned in an `export` or `unexport` directive should be exported. Any variable given in an `unexport` directive will still *not* be exported. If you use `export` by itself to export variables by default, variables whose names contain characters other than alphanumerics and underscores will not be exported unless specifically mentioned in an `export` directive.

The behavior elicited by an `export` directive by itself was the default in older versions of GNU `make`. If your makefiles depend on this behavior and you want to be compatible with old versions of `make`, you can write a rule for the special target `.EXPORT_ALL_VARIABLES` instead of using the `export` directive. This will be ignored by old `make`s, while the `export` directive will cause a syntax error.

Likewise, you can use `unexport` by itself to tell `make` *not* to export variables by default. Since this is the default behavior, you would only need to do this if `export` had been used by itself earlier (in an included makefile, perhaps). You **cannot** use `export` and `unexport` by themselves to have variables exported for some commands and not for others. The last `export` or `unexport` directive that appears

by itself determines the behavior for the entire run of `make`.

As a special feature, the variable `MAKELEVEL` is changed when it is passed down from level to level. This variable's value is a string which is the depth of the level as a decimal number. The value is `0' for the top-level `make`; `1' for a sub-`make`, `2' for a sub-sub-`make`, and so on. The incrementation happens when `make` sets up the environment for a command.

The main use of `MAKELEVEL` is to test it in a conditional directive (see section Conditional Parts of Makefiles); this way you can write a makefile that behaves one way if run recursively and another way if run directly by you.

You can use the variable `MAKEFILES` to cause all sub-`make` commands to use additional makefiles. The value of `MAKEFILES` is a whitespace-separated list of file names. This variable, if defined in the outer-level makefile, is passed down through the environment; then it serves as a list of extra makefiles for the sub-`make` to read before the usual or specified ones. See section The Variable `MAKEFILES`}.

Communicating Options to a Sub-`make`

Flags such as `-s' and `-k' are passed automatically to the sub-`make` through the variable `MAKEFLAGS`. This variable is set up automatically by `make` to contain the flag letters that `make` received. Thus, if you do `make -ks' then `MAKEFLAGS` gets the value `ks'.

As a consequence, every sub-`make` gets a value for `MAKEFLAGS` in its environment. In response, it takes the flags from that value and processes them as if they had been given as arguments. See section Summary of Options.

Likewise variables defined on the command line are passed to the sub-`make` through `MAKEFLAGS`. Words in the value of `MAKEFLAGS` that contain `=', `make` treats as variable definitions just as if they appeared on the command line. See section Overriding Variables.

The options `-C', `-f', `-o', and `-W' are not put into `MAKEFLAGS`; these options are not passed down.

The `-j' option is a special case (see section Parallel Execution). If you set it to some numeric value, `-j 1' is always put into `MAKEFLAGS` instead of the value you specified. This is because if the `-j' option were passed down to sub-`make`s, you would get many more jobs running in parallel than you asked for. If you give `-j' with no numeric argument, meaning to run as many jobs as possible in parallel, this is passed down, since multiple infinities are no more than one.

If you do not want to pass the other flags down, you must change the value of `MAKEFLAGS`, like this:

```
MAKEFLAGS=
subsystem:
        cd subdir; $(MAKE)
```

or like this:

```
subsystem:
```


```
        cd subdir; $(MAKE) MAKEFLAGS=
```

The command line variable definitions really appear in the variable `MAKEOVERRIDES`, and `MAKEFLAGS` contains a reference to this variable. If you do want to pass flags down normally, but don't want to pass down the command line variable definitions, you can reset `MAKEOVERRIDES` to empty, like this:

```
MAKEOVERRIDES =
```

This is not usually useful to do. However, some systems have a small fixed limit on the size of the environment, and putting so much information in into the value of `MAKEFLAGS` can exceed it. If you see the error message `Arg list too long', this may be the problem. (For strict compliance with POSIX.2, changing `MAKEOVERRIDES` does not affect `MAKEFLAGS` if the special target `.POSIX' appears in the makefile. You probably do not care about this.)

A similar variable `MFLAGS` exists also, for historical compatibility. It has the same value as `MAKEFLAGS` except that it does not contain the command line variable definitions, and it always begins with a hyphen unless it is empty (`MAKEFLAGS` begins with a hyphen only when it begins with an option that has no single-letter version, such as `--warn-undefined-variables'). `MFLAGS` was traditionally used explicitly in the recursive `make` command, like this:

```
subsystem:
        cd subdir; $(MAKE) $(MFLAGS)
```

but now `MAKEFLAGS` makes this usage redundant. If you want your makefiles to be compatible with old `make` programs, use this technique; it will work fine with more modern `make` versions too.

The `MAKEFLAGS` variable can also be useful if you want to have certain options, such as `-k' (see section Summary of Options), set each time you run `make`. You simply put a value for `MAKEFLAGS` in your environment. You can also set `MAKEFLAGS` in a makefile, to specify additional flags that should also be in effect for that makefile. (Note that you cannot use `MFLAGS` this way. That variable is set only for compatibility; `make` does not interpret a value you set for it in any way.)

When `make` interprets the value of `MAKEFLAGS` (either from the environment or from a makefile), it first prepends a hyphen if the value does not already begin with one. Then it chops the value into words separated by blanks, and parses these words as if they were options given on the command line (except that `-C', `-f', `-h', `-o', `-W', and their long-named versions are ignored; and there is no error for an invalid option).

If you do put `MAKEFLAGS` in your environment, you should be sure not to include any options that will drastically affect the actions of `make` and undermine the purpose of makefiles and of `make` itself. For instance, the `-t', `-n', and `-q' options, if put in one of these variables, could have disastrous consequences and would certainly have at least surprising and probably annoying effects.

The `--print-directory' Option

If you use several levels of recursive `make` invocations, the `-w' or `--print-directory' option can make the output a lot easier to understand by showing each directory as `make` starts processing it and as `make` finishes processing it. For example, if `make -w' is run in the directory `/u/gnu/make', `make` will print a line of the form:

```
make: Entering directory `/u/gnu/make'.
```

before doing anything else, and a line of the form:

```
make: Leaving directory `/u/gnu/make'.
```

when processing is completed.

Normally, you do not need to specify this option because `make' does it for you: `-w' is turned on automatically when you use the `-C' option, and in sub-`make`s. `make` will not automatically turn on `-w' if you also use `-s', which says to be silent, or if you use `--no-print-directory' to explicitly disable it.

Defining Canned Command Sequences

When the same sequence of commands is useful in making various targets, you can define it as a canned sequence with the `define` directive, and refer to the canned sequence from the rules for those targets. The canned sequence is actually a variable, so the name must not conflict with other variable names.

Here is an example of defining a canned sequence of commands:

```
define run-yacc
yacc $(firstword $^)
mv y.tab.c $@
endef
```

Here `run-yacc` is the name of the variable being defined; `endef` marks the end of the definition; the lines in between are the commands. The `define` directive does not expand variable references and function calls in the canned sequence; the `$' characters, parentheses, variable names, and so on, all become part of the value of the variable you are defining. See section Defining Variables Verbatim, for a complete explanation of `define`.

The first command in this example runs Yacc on the first dependency of whichever rule uses the canned sequence. The output file from Yacc is always named `y.tab.c'. The second command moves the output to the rule's target file name.

To use the canned sequence, substitute the variable into the commands of a rule. You can substitute it like any other variable (see section Basics of Variable References). Because variables defined by `define` are recursively expanded variables, all the variable references you wrote inside the `define` are

expanded now. For example:

```
foo.c : foo.y
        $(run-yacc)
```

`foo.y' will be substituted for the variable `$^' when it occurs in `run-yacc`'s value, and `foo.c' for `$@'.

This is a realistic example, but this particular one is not needed in practice because `make` has an implicit rule to figure out these commands based on the file names involved (see section Using Implicit Rules).

In command execution, each line of a canned sequence is treated just as if the line appeared on its own in the rule, preceded by a tab. In particular, `make` invokes a separate subshell for each line. You can use the special prefix characters that affect command lines (`@', `-', and `+') on each line of a canned sequence. See section Writing the Commands in Rules. For example, using this canned sequence:

```
define frobnicate
@echo "frobnicating target $@"
frob-step-1 $< -o $@-step-1
frob-step-2 $@-step-1 -o $@
endef
```

`make` will not echo the first line, the `echo` command. But it *will* echo the following two command lines.

On the other hand, prefix characters on the command line that refers to a canned sequence apply to every line in the sequence. So the rule:

```
frob.out: frob.in
        @$(frobnicate)
```

does not echo *any* commands. (See section Command Echoing, for a full explanation of `@'.)

Using Empty Commands

It is sometimes useful to define commands which do nothing. This is done simply by giving a command that consists of nothing but whitespace. For example:

```
target: ;
```

defines an empty command string for `target'. You could also use a line beginning with a tab character to define an empty command string, but this would be confusing because such a line looks empty.

You may be wondering why you would want to define a command string that does nothing. The only reason this is useful is to prevent a target from getting implicit commands (from implicit rules or the `.DEFAULT` special target; see section Using Implicit Rules and see section Defining Last-Resort Default Rules).

You may be inclined to define empty command strings for targets that are not actual files, but only e so that their dependencies can be remade. However, this is not the best way to do that, because the dependencies may not be remade properly if the target file actually does exist. See section Phony Targets, for a better way to do this.

Go to the previous, next section.

Go to the previous, next section.

TOC

How to Use Variables

A variable is a name defined in a makefile to represent a string of text, called the variable's value. These values are substituted by explicit request into targets, dependencies, commands, and other parts of the makefile. (In some other versions of `make`, variables are called macros.)

Variables and functions in all parts of a makefile are expanded when read, except for the shell commands in rules, the right-hand sides of variable definitions using `=', and the bodies of variable definitions using the `define` directive.

Variables can represent lists of file names, options to pass to compilers, programs to run, directories to look in for source files, directories to write output in, or anything else you can imagine.

A variable name may be any sequence of characters not containing `:', `#', `=', or leading or trailing whitespace. However, variable names containing characters other than letters, numbers, and underscores should be avoided, as they may be given special meanings in the future, and with some shells they cannot be passed through the environment to a sub-`make` (see section Communicating Variables to a Sub-`make`).

Variable names are case-sensitive. The names `foo', `FOO', and `Foo' all refer to different variables.

It is traditional to use upper case letters in variable names, but we recommend using lower case letters for variable names that serve internal purposes in the makefile, and reserving upper case for parameters that control implicit rules or for parameters that the user should override with command options (see section Overriding Variables).

A few variables have names that are a single punctuation character or just a few characters. These are the automatic variables, and they have particular specialized uses. See section Automatic Variables.

Basics of Variable References

To substitute a variable's value, write a dollar sign followed by the name of the variable in parentheses or braces: either `$(foo)' or `${foo}' is a valid reference to the variable `foo`. This special significance of `$' is why you must write `$$' to have the effect of a single dollar sign in a file name or command.

Variable references can be used in any context: targets, dependencies, commands, most directives, and new variable values. Here is an example of a common case, where a variable holds the names of all the object files in a program:

```
objects = program.o foo.o utils.o
program : $(objects)
        cc -o program $(objects)
```


```
$(objects) : defs.h
```

Variable references work by strict textual substitution. Thus, the rule

```
foo = c
prog.o : prog.$(foo)
        $(foo)$(foo) -$(foo) prog.$(foo)
```

could be used to compile a C program `prog.c'. Since spaces before the variable value are ignored in variable assignments, the value of `foo` is precisely `c'. (Don't actually write your makefiles this way!)

A dollar sign followed by a character other than a dollar sign, open-parenthesis or open-brace treats that single character as the variable name. Thus, you could reference the variable `x` with `$x'. However, this practice is strongly discouraged, except in the case of the automatic variables (see section Automatic Variables).

The Two Flavors of Variables

There are two ways that a variable in GNU `make` can have a value; we call them the two flavors of variables. The two flavors are distinguished in how they are defined and in what they do when expanded.

The first flavor of variable is a recursively expanded variable. Variables of this sort are defined by lines using `=' (see section Setting Variables) or by the `define` directive (see section Defining Variables Verbatim). The value you specify is installed verbatim; if it contains references to other variables, these references are expanded whenever this variable is substituted (in the course of expanding some other string). When this happens, it is called recursive expansion.

For example,

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:;echo $(foo)
```

will echo `Huh?': `$(foo)' expands to `$(bar)' which expands to `$(ugh)' which finally expands to `Huh?'.

This flavor of variable is the only sort supported by other versions of `make`. It has its advantages and its disadvantages. An advantage (most would say) is that:

```
CFLAGS = $(include_dirs) -O
include_dirs = -Ifoo -Ibar
```

will do what was intended: when `CFLAGS' is expanded in a command, it will expand to `-Ifoo -Ibar -O'. A major disadvantage is that you cannot append something on the end of a variable, as in


```
CFLAGS = $(CFLAGS) -O
```

because it will cause an infinite loop in the variable expansion. (Actually `make` detects the infinite loop and reports an error.)

Another disadvantage is that any functions (see section Functions for Transforming Text) referenced in the definition will be executed every time the variable is expanded. This makes `make` run slower; worse, it causes the `wildcard` and `shell` functions to give unpredictable results because you cannot easily control when they are called, or even how many times.

To avoid all the problems and inconveniences of recursively expanded variables, there is another flavor: simply expanded variables.

Simply expanded variables are defined by lines using `:=' (see section Setting Variables). The value of a simply expanded variable is scanned once and for all, expanding any references to other variables and functions, when the variable is defined. The actual value of the simply expanded variable is the result of expanding the text that you write. It does not contain any references to other variables; it contains their values *as of the time this variable was defined*. Therefore,

```
x := foo
y := $(x) bar
x := later
```

is equivalent to

```
y := foo bar
x := later
```

When a simply expanded variable is referenced, its value is substituted verbatim.

Here is a somewhat more complicated example, illustrating the use of `:=' in conjunction with the `shell` function. (See section The `shell` Function.) This example also shows use of the variable `MAKELEVEL`, which is changed when it is passed down from level to level. (See section Communicating Variables to a Sub-`make`}, for information about `MAKELEVEL`.)

```
ifeq (0,${MAKELEVEL})
cur-dir   := $(shell pwd)
whoami    := $(shell whoami)
host-type := $(shell arch)
MAKE := ${MAKE} host-type=${host-type} whoami=${whoami}
endif
```

An advantage of this use of `:=' is that a typical `descend into a directory' command then looks like this:

```
${subdirs}:
      ${MAKE} cur-dir=${cur-dir}/$@ -C $@ all
```

Simply expanded variables generally make complicated makefile programming more predictable bec they work like variables in most programming languages. They allow you to redefine a variable usin own value (or its value processed in some way by one of the expansion functions) and to use the expansion functions much more efficiently (see section Functions for Transforming Text).

You can also use them to introduce controlled leading whitespace into variable values. Leading whitespace characters are discarded from your input before substitution of variable references and function calls; this means you can include leading spaces in a variable value by protecting them with variable references, like this:

```
nullstring :=
space := $(nullstring) # end of the line
```

Here the value of the variable `space` is precisely one space. The comment `# end of the line' is included here just for clarity. Since trailing space characters are *not* stripped from variable values, just a space at the end of the line would have the same effect (but be rather hard to read). If you put whitespace at the end of a variable value, it is a good idea to put a comment like that at the end of the line to make your intent clear. Conversely, if you do *not* want any whitespace characters at the end of your variable value, you must remember not to put a random comment on the end of the line after some whitespace, such as this:

```
dir := /foo/bar    # directory to put the frobs in
```

Here the value of the variable `dir` is `/foo/bar ' (with four trailing spaces), which was probably not the intention. (Imagine something like `$(dir)/file' with this definition!)

Advanced Features for Reference to Variables

This section describes some advanced features you can use to reference variables in more flexible ways.

Substitution References

A substitution reference substitutes the value of a variable with alterations that you specify. It has the form `$(var:a=b)' (or `${var:a=b}') and its meaning is to take the value of the variable var, replace every a at the end of a word with b in that value, and substitute the resulting string.

When we say "at the end of a word", we mean that a must appear either followed by whitespace or at the end of the value in order to be replaced; other occurrences of a in the value are unaltered. For example:

```
foo := a.o b.o c.o
bar := $(foo:.o=.c)
```

sets `bar' to `a.c b.c c.c'. See section Setting Variables.

A substitution reference is actually an abbreviation for use of the `patsubst` expansion function (see section Functions for String Substitution and Analysis). We provide substitution references as well as

`patsubst` for compatibility with other implementations of `make`.

Another type of substitution reference lets you use the full power of the `patsubst` function. It has the same form `$(var:a=b)' described above, except that now a must contain a single `%' character. This case is equivalent to `$(patsubst a,b,$(var))'. See section Functions for String Substitution and Analysis, for a description of the `patsubst` function.

For example:

```
foo := a.o b.o c.o
bar := $(foo:%.o=%.c)
```

sets `bar' to `a.c b.c c.c'.

Computed Variable Names

Computed variable names are a complicated concept needed only for sophisticated makefile programming. For most purposes you need not consider them, except to know that making a variable with a dollar sign in its name might have strange results. However, if you are the type that wants to understand everything, or you are actually interested in what they do, read on.

Variables may be referenced inside the name of a variable. This is called a computed variable name or a nested variable reference. For example,

```
x = y
y = z
a := $($(x))
```

defines `a` as `z': the `$(x)' inside `$($(x))' expands to `y', so `$($(x))' expands to `$(y)' which in turn expands to `z'. Here the name of the variable to reference is not stated explicitly; it is computed by expansion of `$(x)'. The reference `$(x)' here is nested within the outer variable reference.

The previous example shows two levels of nesting, but any number of levels is possible. For example, here are three levels:

```
x = y
y = z
z = u
a := $($($(x)))
```

Here the innermost `$(x)' expands to `y', so `$($(x))' expands to `$(y)' which in turn expands to `z'; now we have `$(z)', which becomes `u'.

References to recursively-expanded variables within a variable name are reexpanded in the usual fashion. For example:

```
x = $(y)
```


```
y = z
z = Hello
a := $($(x))
```

defines `a` as `Hello': `$($(x))' becomes `$($(y))' which becomes `$(z)' which becomes `Hello'.

Nested variable references can also contain modified references and function invocations (see section Functions for Transforming Text), just like any other reference. For example, using the `subst` function (see section Functions for String Substitution and Analysis):

```
x = variable1
variable2 := Hello
y = $(subst 1,2,$(x))
z = y
a := $($($(z)))
```

eventually defines `a` as `Hello'. It is doubtful that anyone would ever want to write a nested reference as convoluted as this one, but it works: `$($($(z)))' expands to `$($(y))' which becomes `$($(subst 1,2,$(x)))'. This gets the value `variable1' from `x` and changes it by substitution to `variable2', so that the entire string becomes `$(variable2)', a simple variable reference whose value is `Hello'.

A computed variable name need not consist entirely of a single variable reference. It can contain several variable references, as well as some invariant text. For example,

```
a_dirs := dira dirb
1_dirs := dir1 dir2

a_files := filea fileb
1_files := file1 file2

ifeq "$(use_a)" "yes"
a1 := a
else
a1 := 1
endif

ifeq "$(use_dirs)" "yes"
df := dirs
else
df := files
endif

dirs := $($(a1)_$(df))
```

will give `dirs` the same value as `a_dirs`, `1_dirs`, `a_files` or `1_files` depending on the settings of `use_a` and `use_dirs`.

Computed variable names can also be used in substitution references:

```
a_objects := a.o b.o c.o
1_objects := 1.o 2.o 3.o

sources := $($(a1)_objects:.o=.c)
```

defines `sources` as either `a.c b.c c.c' or `1.c 2.c 3.c', depending on the value of `a1`.

The only restriction on this sort of use of nested variable references is that they cannot specify part of the name of a function to be called. This is because the test for a recognized function name is done before the expansion of nested references. For example,

```
ifdef do_sort
func := sort
else
func := strip
endif

bar := a d b g q c

foo := $($(func) $(bar))
```

attempts to give `foo' the value of the variable `sort a d b g q c' or `strip a d b g q c', rather than giving `a d b g q c' as the argument to either the `sort` or the `strip` function. This restriction could be removed in the future if that change is shown to be a good idea.

You can also use computed variable names in the left-hand side of a variable assignment, or in a `define` directive, as in:

```
dir = foo
$(dir)_sources := $(wildcard $(dir)/*.c)
define $(dir)_print
lpr $($(dir)_sources)
endef
```

This example defines the variables `dir', `foo_sources', and `foo_print'.

Note that nested variable references are quite different from recursively expanded variables (see section The Two Flavors of Variables), though both are used together in complex ways when doing makefile programming.

How Variables Get Their Values

Variables can get values in several different ways:

- You can specify an overriding value when you run `make`. See section Overriding Variables.
- You can specify a value in the makefile, either with an assignment (see section Setting Variables) or with a verbatim definition (see section Defining Variables Verbatim).
- Variables in the environment become `make` variables. See section Variables from the Environment.
- Several automatic variables are given new values for each rule. Each of these has a single conventional use. See section Automatic Variables.
- Several variables have constant initial values. See section Variables Used by Implicit Rules.

Setting Variables

To set a variable from the makefile, write a line starting with the variable name followed by `=' or `:='. Whatever follows the `=' or `:=' on the line becomes the value. For example,

```
objects = main.o foo.o bar.o utils.o
```

defines a variable named `objects`. Whitespace around the variable name and immediately after the `=' is ignored.

Variables defined with `=' are recursively expanded variables. Variables defined with `:=' are simply expanded variables; these definitions can contain variable references which will be expanded before the definition is made. See section The Two Flavors of Variables.

The variable name may contain function and variable references, which are expanded when the line is read to find the actual variable name to use.

There is no limit on the length of the value of a variable except the amount of swapping space on the computer. When a variable definition is long, it is a good idea to break it into several lines by inserting backslash-newline at convenient places in the definition. This will not affect the functioning of `make`, but it will make the makefile easier to read.

Most variable names are considered to have the empty string as a value if you have never set them. Several variables have built-in initial values that are not empty, but you can set them in the usual ways (see section Variables Used by Implicit Rules). Several special variables are set automatically to a new value for each rule; these are called the automatic variables (see section Automatic Variables).

Appending More Text to Variables

Often it is useful to add more text to the value of a variable already defined. You do this with a line containing `+=', like this:

```
objects += another.o
```

This takes the value of the variable `objects`, and adds the text `another.o' to it (preceded by a single space). Thus:

```
objects = main.o foo.o bar.o utils.o
objects += another.o
```

sets `objects` to `main.o foo.o bar.o utils.o another.o'.

Using `+=' is similar to:

```
objects = main.o foo.o bar.o utils.o
objects := $(objects) another.o
```

but differs in ways that become important when you use more complex values.

When the variable in question has not been defined before, `+=' acts just like normal `=': it defines a recursively-expanded variable. However, when there *is* a previous definition, exactly what `+=' does depends on what flavor of variable you defined originally. See section The Two Flavors of Variables, for an explanation of the two flavors of variables.

When you add to a variable's value with `+=', `make` acts essentially as if you had included the extra text in the initial definition of the variable. If you defined it first with `:=', making it a simply-expanded variable, `+=' adds to that simply-expanded definition, and expands the new text before appending it to the old value just as `:=' does (see section Setting Variables, for a full explanation of `:='). In fact,

```
variable := value
variable += more
```

is exactly equivalent to:

```
variable := value
variable := $(variable) more
```

On the other hand, when you use `+=' with a variable that you defined first to be recursively-expanded using plain `=', `make` does something a bit different. Recall that when you define a recursively-expanded variable, `make` does not expand the value you set for variable and function references immediately. Instead it stores the text verbatim, and saves these variable and function references to be expanded later, when you refer to the new variable (see section The Two Flavors of Variables). When you use `+=' on a recursively-expanded variable, it is this unexpanded text to which `make` appends the new text you

specify.

```
variable = value
variable += more
```

is roughly equivalent to:

```
temp = value
variable = $(temp) more
```

except that of course it never defines a variable called `temp`. The importance of this comes when the variable's old value contains variable references. Take this common example:

```
CFLAGS = $(includes) -O
...
CFLAGS += -pg # enable profiling
```

The first line defines the `CFLAGS` variable with a reference to another variable, `includes`. (`CFLAGS` is used by the rules for C compilation; see section Catalogue of Implicit Rules.) Using `=' for the definition makes `CFLAGS` a recursively-expanded variable, meaning `$(includes) -O' is *not* expanded when `make` processes the definition of `CFLAGS`. Thus, `includes` need not be defined yet for its value to take effect. It only has to be defined before any reference to `CFLAGS`. If we tried to append to the value of `CFLAGS` without using `+=', we might do it like this:

```
CFLAGS := $(CFLAGS) -pg # enable profiling
```

This is pretty close, but not quite what we want. Using `:=' redefines `CFLAGS` as a simply-expanded variable; this means `make` expands the text `$(CFLAGS) -pg' before setting the variable. If `includes` is not yet defined, we get ` -O -pg', and a later definition of `includes` will have no effect. Conversely, by using `+=' we set `CFLAGS` to the *unexpanded* value `$(includes) -O -pg'. Thus we preserve the reference to `includes`, so if that variable gets defined at any later point, a reference like `$(CFLAGS)' still uses its value.

The `override` Directive

If a variable has been set with a command argument (see section Overriding Variables), then ordinary assignments in the makefile are ignored. If you want to set the variable in the makefile even though it was set with a command argument, you can use an `override` directive, which is a line that looks like this:

```
override variable = value
```

or

```
override variable := value
```

To append more text to a variable defined on the command line, use:

```
override variable += more text
```

See section Appending More Text to Variables.

The `override` directive was not invented for escalation in the war between makefiles and command arguments. It was invented so you can alter and add to values that the user specifies with command arguments.

For example, suppose you always want the `-g' switch when you run the C compiler, but you would like to allow the user to specify the other switches with a command argument just as usual. You could use this `override` directive:

```
override CFLAGS += -g
```

You can also use `override` directives with `define` directives. This is done as you might expect:

```
override define foo
bar
endef
```

See the next section for information about `define`.

Defining Variables Verbatim

Another way to set the value of a variable is to use the `define` directive. This directive has an unusual syntax which allows newline characters to be included in the value, which is convenient for defining canned sequences of commands (see section Defining Canned Command Sequences).

The `define` directive is followed on the same line by the name of the variable and nothing more. The value to give the variable appears on the following lines. The end of the value is marked by a line containing just the word `endef`. Aside from this difference in syntax, `define` works just like `=': it creates a recursively-expanded variable (see section The Two Flavors of Variables). The variable name may contain function and variable references, which are expanded when the directive is read to find the actual variable name to use.

```
define two-lines
echo foo
echo $(bar)
endef
```

The value in an ordinary assignment cannot contain a newline; but the newlines that separate the lines of the value in a `define` become part of the variable's value (except for the final newline which precedes the `endef` and is not considered part of the value).

The previous example is functionally equivalent to this:

```
two-lines = echo foo; echo $(bar)
```

since two commands separated by semicolon behave much like two separate shell commands. However, note that using two separate lines means `make` will invoke the shell twice, running an independent subshell for each line. See section Command Execution.

If you want variable definitions made with `define` to take precedence over command-line variable definitions, you can use the `override` directive together with `define`:

```
override define two-lines
foo
$(bar)
endef
```

See section The `override` Directive.

Variables from the Environment

Variables in `make` can come from the environment in which `make` is run. Every environment variable that `make` sees when it starts up is transformed into a `make` variable with the same name and value. But an explicit assignment in the makefile, or with a command argument, overrides the environment. (If the `-e' flag is specified, then values from the environment override assignments in the makefile. See section Summary of Options. But this is not recommended practice.)

Thus, by setting the variable `CFLAGS` in your environment, you can cause all C compilations in most makefiles to use the compiler switches you prefer. This is safe for variables with standard or conventional meanings because you know that no makefile will use them for other things. (But this is not totally reliable; some makefiles set `CFLAGS` explicitly and therefore are not affected by the value in the environment.)

When `make` is invoked recursively, variables defined in the outer invocation can be passed to inner invocations through the environment (see section Recursive Use of `make`). By default, only variables that came from the environment or the command line are passed to recursive invocations. You can use the `export` directive to pass other variables. See section Communicating Variables to a Sub-`make`}, for full details.

Other use of variables from the environment is not recommended. It is not wise for makefiles to depend for their functioning on environment variables set up outside their control, since this would cause different users to get different results from the same makefile. This is against the whole purpose of most makefiles.

Such problems would be especially likely with the variable `SHELL`, which is normally present in the environment to specify the user's choice of interactive shell. It would be very undesirable for this choice to affect `make`. So `make` ignores the environment value of `SHELL`.

Go to the previous, next section.

TOC

Go to the previous, next section.

Conditional Parts of Makefiles

A conditional causes part of a makefile to be obeyed or ignored depending on the values of variables. Conditionals can compare the value of one variable to another, or the value of a variable to a constant string. Conditionals control what `make` actually "sees" in the makefile, so they *cannot* be used to control shell commands at the time of execution.

Example of a Conditional

The following example of a conditional tells `make` to use one set of libraries if the `CC` variable is `gcc', and a different set of libraries otherwise. It works by controlling which of two command lines will be used as the command for a rule. The result is that `CC=gcc' as an argument to `make` changes not only which compiler is used but also which libraries are linked.

```
libs_for_gcc = -lgnu
normal_libs =

foo: $(objects)
ifeq ($(CC),gcc)
        $(CC) -o foo $(objects) $(libs_for_gcc)
else
        $(CC) -o foo $(objects) $(normal_libs)
endif
```

This conditional uses three directives: one `ifeq`, one `else` and one `endif`.

The `ifeq` directive begins the conditional, and specifies the condition. It contains two arguments, separated by a comma and surrounded by parentheses. Variable substitution is performed on both arguments and then they are compared. The lines of the makefile following the `ifeq` are obeyed if the two arguments match; otherwise they are ignored.

The `else` directive causes the following lines to be obeyed if the previous conditional failed. In the example above, this means that the second alternative linking command is used whenever the first alternative is not used. It is optional to have an `else` in a conditional.

The `endif` directive ends the conditional. Every conditional must end with an `endif`. Unconditional makefile text follows.

As this example illustrates, conditionals work at the textual level: the lines of the conditional are treated as part of the makefile, or ignored, according to the condition. This is why the larger syntactic units of the makefile, such as rules, may cross the beginning or the end of the conditional.

When the variable `CC` has the value `gcc', the above example has this effect:


```
foo: $(objects)
        $(CC) -o foo $(objects) $(libs_for_gcc)
```

When the variable `CC` has any other value, the effect is this:

```
foo: $(objects)
        $(CC) -o foo $(objects) $(normal_libs)
```

Equivalent results can be obtained in another way by conditionalizing a variable assignment and then using the variable unconditionally:

```
libs_for_gcc = -lgnu
normal_libs =

ifeq ($(CC),gcc)
  libs=$(libs_for_gcc)
else
  libs=$(normal_libs)
endif

foo: $(objects)
        $(CC) -o foo $(objects) $(libs)
```

Syntax of Conditionals

The syntax of a simple conditional with no `else` is as follows:

```
conditional-directive
text-if-true
endif
```

The text-if-true may be any lines of text, to be considered as part of the makefile if the condition is true. If the condition is false, no text is used instead.

The syntax of a complex conditional is as follows:

```
conditional-directive
text-if-true
else
text-if-false
endif
```

If the condition is true, text-if-true is used; otherwise, text-if-false is used instead. The text-if-false can be any number of lines of text.

The syntax of the conditional-directive is the same whether the conditional is simple or complex. Th
are four different directives that test different conditions. Here is a table of them:

`ifeq (arg1, arg2)`
`ifeq 'arg1' 'arg2'`
`ifeq "arg1" "arg2"`
`ifeq "arg1" 'arg2'`
`ifeq 'arg1' "arg2"`

Expand all variable references in arg1 and arg2 and compare them. If they are identical, the text-if-true is effective; otherwise, the text-if-false, if any, is effective.

Often you want to test if a variable has a non-empty value. When the value results from complex expansions of variables and functions, expansions you would consider empty may actually contain whitespace characters and thus are not seen as empty. However, you can use the `strip` function (see section Functions for String Substitution and Analysis) to avoid interpreting whitespace as a non-empty value. For example:

```
ifeq ($(strip $(foo)),)
text-if-empty
endif
```

will evaluate text-if-empty even if the expansion of `$(foo)` contains whitespace characters.

`ifneq (arg1, arg2)`
`ifneq 'arg1' 'arg2'`
`ifneq "arg1" "arg2"`
`ifneq "arg1" 'arg2'`
`ifneq 'arg1' "arg2"`

Expand all variable references in arg1 and arg2 and compare them. If they are different, the text-if-true is effective; otherwise, the text-if-false, if any, is effective.

`ifdef variable-name`

If the variable variable-name has a non-empty value, the text-if-true is effective; otherwise, the text-if-false, if any, is effective. Variables that have never been defined have an empty value.

Note that `ifdef` only tests whether a variable has a value. It does not expand the variable to see if that value is nonempty. Consequently, tests using `ifdef` return true for all definitions except those like `foo =`. To test for an empty value, use `ifeq ($(foo),)`. For example,

```
bar =
foo = $(bar)
ifdef foo
frobozz = yes
else
frobozz = no
```


```
endif
```

sets `frobozz' to `yes', while:

```
foo =
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

sets `frobozz' to `no'.

`ifndef variable-name`

If the variable variable-name has an empty value, the text-if-true is effective; otherwise, the text-if-false, if any, is effective.

Extra spaces are allowed and ignored at the beginning of the conditional directive line, but a tab is not allowed. (If the line begins with a tab, it will be considered a command for a rule.) Aside from this, extra spaces or tabs may be inserted with no effect anywhere except within the directive name or within an argument. A comment starting with `#' may appear at the end of the line.

The other two directives that play a part in a conditional are `else` and `endif`. Each of these directives is written as one word, with no arguments. Extra spaces are allowed and ignored at the beginning of the line, and spaces or tabs at the end. A comment starting with `#' may appear at the end of the line.

Conditionals affect which lines of the makefile `make` uses. If the condition is true, `make` reads the lines of the text-if-true as part of the makefile; if the condition is false, `make` ignores those lines completely. It follows that syntactic units of the makefile, such as rules, may safely be split across the beginning or the end of the conditional.

`make` evaluates conditionals when it reads a makefile. Consequently, you cannot use automatic variables in the tests of conditionals because they are not defined until commands are run (see section Automatic Variables).

To prevent intolerable confusion, it is not permitted to start a conditional in one makefile and end it in another. However, you may write an `include` directive within a conditional, provided you do not attempt to terminate the conditional inside the included file.

Conditionals that Test Flags

You can write a conditional that tests `make` command flags such as `-t' by using the variable `MAKEFLAGS` together with the `findstring` function (see section Functions for String Substitution and Analysis). This is useful when `touch` is not enough to make a file appear up to date.

The `findstring` function determines whether one string appears as a substring of another. If you want to test for the `-t' flag, use `t' as the first string and the value of `MAKEFLAGS` as the other.

For example, here is how to arrange to use `ranlib -t' to finish marking an archive file up to date:

```
archive.a: ...
ifneq (,$(findstring t,$(MAKEFLAGS)))
        +touch archive.a
        +ranlib -t archive.a
else
        ranlib archive.a
endif
```

The `+' prefix marks those command lines as "recursive" so that they will be executed despite use of the `-t' flag. See section Recursive Use of `make`.

Go to the previous, next section.

Go to the previous, next section.

Functions for Transforming Text

Functions allow you to do text processing in the makefile to compute the files to operate on or the commands to use. You use a function in a function call, where you give the name of the function and some text (the arguments) for the function to operate on. The result of the function's processing is substituted into the makefile at the point of the call, just as a variable might be substituted.

Function Call Syntax

A function call resembles a variable reference. It looks like this:

```
$(function arguments)
```

or like this:

```
${function arguments}
```

Here function is a function name; one of a short list of names that are part of `make`. There is no provision for defining new functions.

The arguments are the arguments of the function. They are separated from the function name by one or more spaces or tabs, and if there is more than one argument, then they are separated by commas. Such whitespace and commas are not part of an argument's value. The delimiters which you use to surround the function call, whether parentheses or braces, can appear in an argument only in matching pairs; the other kind of delimiters may appear singly. If the arguments themselves contain other function calls or variable references, it is wisest to use the same kind of delimiters for all the references; write `$(subst a,b,$(x))', not `$(subst a,b,${x})'. This is because it is clearer, and because only one type of delimiter is matched to find the end of the reference.

The text written for each argument is processed by substitution of variables and function calls to produce the argument value, which is the text on which the function acts. The substitution is done in the order in which the arguments appear.

Commas and unmatched parentheses or braces cannot appear in the text of an argument as written; leading spaces cannot appear in the text of the first argument as written. These characters can be put into the argument value by variable substitution. First define variables `comma` and `space` whose values are isolated comma and space characters, then substitute these variables where such characters are wanted, like this:

```
comma:= ,
empty:=
space:= $(empty) $(empty)
```


```
foo:= a b c
bar:= $(subst $(space),$(comma),$(foo))
# bar is now `a,b,c'.
```

Here the `subst` function replaces each space with a comma, through the value of `foo`, and substitutes the result.

Functions for String Substitution and Analysis

Here are some functions that operate on strings:

`$(subst from,to,text)`

Performs a textual replacement on the text text: each occurrence of from is replaced by to. The result is substituted for the function call. For example,

```
$(subst ee,EE,feet on the street)
```

substitutes the string `fEEt on the strEEt'.

`$(patsubst pattern,replacement,text)`

Finds whitespace-separated words in text that match pattern and replaces them with replacement. Here pattern may contain a `%' which acts as a wildcard, matching any number of any characters within a word. If replacement also contains a `%', the `%' is replaced by the text that matched the `%' in pattern.

`%' characters in `patsubst` function invocations can be quoted with preceding backslashes (`\'). Backslashes that would otherwise quote `%' characters can be quoted with more backslashes. Backslashes that quote `%' characters or other backslashes are removed from the pattern before it is compared file names or has a stem substituted into it. Backslashes that are not in danger of quoting `%' characters go unmolested. For example, the pattern `` `the\%weird\\%pattern\\' `` has `the%weird\' preceding the operative `%' character, and `pattern\\' following it. The final two backslashes are left alone because they cannot affect any `%' character.

Whitespace between words is folded into single space characters; leading and trailing whitespace is discarded.

For example,

```
$(patsubst %.c,%.o,x.c.c bar.c)
```

produces the value `x.c.o bar.o'.

Substitution references (see section Substitution References) are a simpler way to get the effect of the `patsubst` function:

```
$(var:pattern=replacement)
```

is equivalent to

```
$(patsubst pattern,replacement,$(var))
```

The second shorthand simplifies one of the most common uses of `patsubst`: replacing the suffix at the end of file names.

```
$(var:suffix=replacement)
```

is equivalent to

```
$(patsubst %suffix,%replacement,$(var))
```

For example, you might have a list of object files:

```
objects = foo.o bar.o baz.o
```

To get the list of corresponding source files, you could simply write:

```
$(objects:.o=.c)
```

instead of using the general form:

```
$(patsubst %.o,%.c,$(objects))
```

`$(strip string)`

Removes leading and trailing whitespace from string and replaces each internal sequence of one or more whitespace characters with a single space. Thus, `$(strip a b c )' results in `a b c'.

The function `strip` can be very useful when used in conjunction with conditionals. When comparing something with the empty string `' using `ifeq` or `ifneq`, you usually want a string of just whitespace to match the empty string (see section Conditional Parts of Makefiles).

Thus, the following may fail to have the desired results:

```
.PHONY: all
ifneq   "$(needs_made)" ""
all: $(needs_made)
else
all:;@echo 'Nothing to make!'
endif
```

Replacing the variable reference `$(needs_made)' with the function call `$(strip $(needs_made))' in the `ifneq` directive would make it more robust.

`$(findstring find,in)`

Searches in for an occurrence of find. If it occurs, the value is find; otherwise, the value is empty. You can use this function in a conditional to test for the presence of a specific substring in a given

string. Thus, the two examples,

```
$(findstring a,a b c)
$(findstring a,b c)
```

produce the values `a' and `' (the empty string), respectively. See section Conditionals that Test Flags, for a practical application of `findstring`.

`$(filter pattern...,text)`

Removes all whitespace-separated words in text that do *not* match any of the pattern words, returning only matching words. The patterns are written using `%', just like the patterns used in the `patsubst` function above.

The `filter` function can be used to separate out different types of strings (such as file names) in a variable. For example:

```
sources := foo.c bar.c baz.s ugh.h
foo: $(sources)
        cc $(filter %.c %.s,$(sources)) -o foo
```

says that `foo' depends of `foo.c', `bar.c', `baz.s' and `ugh.h' but only `foo.c', `bar.c' and `baz.s' should be specified in the command to the compiler.

`$(filter-out pattern...,text)`

Removes all whitespace-separated words in text that *do* match the pattern words, returning only the words that *do not* match. This is the exact opposite of the `filter` function.

For example, given:

```
objects=main1.o foo.o main2.o bar.o
mains=main1.o main2.o
```

the following generates a list which contains all the object files not in `mains':

```
$(filter-out $(mains),$(objects))
```

`$(sort list)`

Sorts the words of list in lexical order, removing duplicate words. The output is a list of words separated by single spaces. Thus,

```
$(sort foo bar lose)
```

returns the value `bar foo lose'.

Incidentally, since `sort` removes duplicate words, you can use it for this purpose even if you don't care about the sort order.

Here is a realistic example of the use of `subst` and `patsubst`. Suppose that a makefile uses the

VPATH variable to specify a list of directories that make should search for dependency files (see sec VPATH: Search Path for All Dependencies). This example shows how to tell the C compiler to searc header files in the same list of directories.

The value of VPATH is a list of directories separated by colons, such as `src:../headers'. First, the subst function is used to change the colons to spaces:

```
$(subst :, ,$(VPATH))
```

This produces `src ../headers'. Then patsubst is used to turn each directory name into a `-I' flag. These can be added to the value of the variable CFLAGS, which is passed automatically to the C compiler, like this:

```
override CFLAGS += $(patsubst %,-I%,$(subst :, ,$(VPATH)))
```

The effect is to append the text `-Isrc -I../headers' to the previously given value of CFLAGS. The override directive is used so that the new value is assigned even if the previous value of CFLAGS was specified with a command argument (see section The override Directive Directive}).

Functions for File Names

Several of the built-in expansion functions relate specifically to taking apart file names or lists of file names.

Each of the following functions performs a specific transformation on a file name. The argument of the function is regarded as a series of file names, separated by whitespace. (Leading and trailing whitespace is ignored.) Each file name in the series is transformed in the same way and the results are concatenated with single spaces between them.

`$(dir names...)`

Extracts the directory-part of each file name in names. The directory-part of the file name is everything up through (and including) the last slash in it. If the file name contains no slash, the directory part is the string `./'. For example,

```
$(dir src/foo.c hacks)
```

produces the result `src/ ./'.

`$(notdir names...)`

Extracts all but the directory-part of each file name in names. If the file name contains no slash, it is left unchanged. Otherwise, everything through the last slash is removed from it.

A file name that ends with a slash becomes an empty string. This is unfortunate, because it means that the result does not always have the same number of whitespace-separated file names as the argument had; but we do not see any other valid alternative.

For example,


```
$(notdir src/foo.c hacks)
```

produces the result `foo.c hacks'.

`$(suffix names...)`

Extracts the suffix of each file name in names. If the file name contains a period, the suffix is everything starting with the last period. Otherwise, the suffix is the empty string. This frequently means that the result will be empty when names is not, and if names contains multiple file names, the result may contain fewer file names.

For example,

```
$(suffix src/foo.c hacks)
```

produces the result `.c'.

`$(basename names...)`

Extracts all but the suffix of each file name in names. If the file name contains a period, the basename is everything starting up to (and not including) the last period. Otherwise, the basename is the entire file name. For example,

```
$(basename src/foo.c hacks)
```

produces the result `src/foo hacks'.

`$(addsuffix suffix,names...)`

The argument names is regarded as a series of names, separated by whitespace; suffix is used as a unit. The value of suffix is appended to the end of each individual name and the resulting larger names are concatenated with single spaces between them. For example,

```
$(addsuffix .c,foo bar)
```

produces the result `foo.c bar.c'.

`$(addprefix prefix,names...)`

The argument names is regarded as a series of names, separated by whitespace; prefix is used as a unit. The value of prefix is prepended to the front of each individual name and the resulting larger names are concatenated with single spaces between them. For example,

```
$(addprefix src/,foo bar)
```

produces the result `src/foo src/bar'.

`$(join list1,list2)`

Concatenates the two arguments word by word: the two first words (one from each argument) concatenated form the first word of the result, the two second words form the second word of the result, and so on. So the nth word of the result comes from the nth word of each argument. If one argument has more words that the other, the extra words are copied unchanged into the result.

For example, `$(join a b,.c .o)' produces `a.c b.o'.

Whitespace between the words in the lists is not preserved; it is replaced with a single space.

This function can merge the results of the `dir` and `notdir` functions, to produce the original list of files which was given to those two functions.

`$(word n,text)`

Returns the nth word of text. The legitimate values of n start from 1. If n is bigger than the number of words in text, the value is empty. For example,

```
$(word 2, foo bar baz)
```

returns `bar'.

`$(words text)`

Returns the number of words in text. Thus, the last word of text is `$(word $(words text),text)`.

`$(firstword names...)`

The argument names is regarded as a series of names, separated by whitespace. The value is the first name in the series. The rest of the names are ignored.

For example,

```
$(firstword foo bar)
```

produces the result `foo'. Although `$(firstword text)` is the same as `$(word 1,text)`, the `firstword` function is retained for its simplicity.

`$(wildcard pattern)`

The argument pattern is a file name pattern, typically containing wildcard characters (as in shell file name patterns). The result of `wildcard` is a space-separated list of the names of existing files that match the pattern. See section Using Wildcard Characters in File Names.

The `foreach` Function

The `foreach` function is very different from other functions. It causes one piece of text to be used repeatedly, each time with a different substitution performed on it. It resembles the `for` command in the shell `sh` and the `foreach` command in the C-shell `csh`.

The syntax of the `foreach` function is:

```
$(foreach var,list,text)
```

The first two arguments, var and list, are expanded before anything else is done; note that the last argument, text, is **not** expanded at the same time. Then for each word of the expanded value of list, the variable named by the expanded value of var is set to that word, and text is expanded. Presumably text

contains references to that variable, so its expansion will be different each time.

The result is that text is expanded as many times as there are whitespace-separated words in list. The multiple expansions of text are concatenated, with spaces between them, to make the result of `foreach`.

This simple example sets the variable `files' to the list of all files in the directories in the list `dirs':

```
dirs := a b c d
files := $(foreach dir,$(dirs),$(wildcard $(dir)/*))
```

Here text is `$(wildcard $(dir)/*)'. The first repetition finds the value `a' for `dir`, so it produces the same result as `$(wildcard a/*)'; the second repetition produces the result of `$(wildcard b/*)'; and the third, that of `$(wildcard c/*)'.

This example has the same result (except for setting `dirs') as the following example:

```
files := $(wildcard a/* b/* c/* d/*)
```

When text is complicated, you can improve readability by giving it a name, with an additional variable:

```
find_files = $(wildcard $(dir)/*)
dirs := a b c d
files := $(foreach dir,$(dirs),$(find_files))
```

Here we use the variable `find_files` this way. We use plain `=' to define a recursively-expanding variable, so that its value contains an actual function call to be reexpanded under the control of `foreach`; a simply-expanded variable would not do, since `wildcard` would be called only once at the time of defining `find_files`.

The `foreach` function has no permanent effect on the variable var; its value and flavor after the `foreach` function call are the same as they were beforehand. The other values which are taken from list are in effect only temporarily, during the execution of `foreach`. The variable var is a simply-expanded variable during the execution of `foreach`. If var was undefined before the `foreach` function call, it is undefined after the call. See section The Two Flavors of Variables.

You must take care when using complex variable expressions that result in variable names because many strange things are valid variable names, but are probably not what you intended. For example,

```
files := $(foreach Esta escrito en espanol!,b c ch,$(find_files))
```

might be useful if the value of `find_files` references the variable whose name is `Esta escrito en espanol!' (es un nombre bastante largo, no?), but it is more likely to be a mistake.

The `origin` Function

The `origin` function is unlike most other functions in that it does not operate on the values of variables; it tells you something *about* a variable. Specifically, it tells you where it came from.

The syntax of the `origin` function is:

```
$(origin variable)
```

Note that variable is the *name* of a variable to inquire about; not a *reference* to that variable. Therefore you would not normally use a `$' or parentheses when writing it. (You can, however, use a variable reference in the name if you want the name not to be a constant.)

The result of this function is a string telling you how the variable variable was defined:

`undefined'

if variable was never defined.

`default'

if variable has a default definition, as is usual with `CC` and so on. See section Variables Used by Implicit Rules. Note that if you have redefined a default variable, the `origin` function will return the origin of the later definition.

`environment'

if variable was defined as an environment variable and the `-e' option is *not* turned on (see section Summary of Options).

`environment override'

if variable was defined as an environment variable and the `-e' option *is* turned on (see section Summary of Options).

`file'

if variable was defined in a makefile.

`command line'

if variable was defined on the command line.

`override'

if variable was defined with an `override` directive in a makefile (see section The `override` Directive).

`automatic'

if variable is an automatic variable defined for the execution of the commands for each rule (see section Automatic Variables).

This information is primarily useful (other than for your curiosity) to determine if you want to believe the

value of a variable. For example, suppose you have a makefile `foo' that includes another makefile `bar'. You want a variable `bletch' to be defined in `bar' if you run the command `make -f bar', even if the environment contains a definition of `bletch'. However, if `foo' defined `bletch' before including `bar', you do not want to override that definition. This could be done by using an `override' directive in `foo', giving that definition precedence over the later definition in `bar'; unfortunately, the `override' directive would also override any command line definitions. So, `bar' could include:

```
ifdef bletch
ifeq "$(origin bletch)" "environment"
bletch = barf, gag, etc.
endif
endif
```

If `bletch' has been defined from the environment, this will redefine it.

If you want to override a previous definition of `bletch' if it came from the environment, even under `-e', you could instead write:

```
ifneq "$(findstring environment,$(origin bletch))" ""
bletch = barf, gag, etc.
endif
```

Here the redefinition takes place if `$(origin bletch)' returns either `environment' or `environment override'. See section Functions for String Substitution and Analysis.

The `shell' Function

The `shell' function is unlike any other function except the `wildcard' function (see section The Function `wildcard') in that it communicates with the world outside of `make'.

The `shell' function performs the same function that backquotes (``') perform in most shells: it does command expansion. This means that it takes an argument that is a shell command and returns the output of the command. The only processing `make' does on the result, before substituting it into the surrounding text, is to convert newlines to spaces.

The commands run by calls to the `shell' function are run when the function calls are expanded. In most cases, this is when the makefile is read in. The exception is that function calls in the commands of the rules are expanded when the commands are run, and this applies to `shell' function calls like all others.

Here are some examples of the use of the `shell' function:

```
contents := $(shell cat foo)
```

sets `contents' to the contents of the file `foo', with a space (rather than a newline) separating each line.


```
files := $(shell echo *.c)
```

sets `files` to the expansion of `*.c'. Unless `make` is using a very strange shell, this has the same result as `$(wildcard *.c)'.

Go to the previous, next section.

Go to the previous, next section.

How to Run `make`

A makefile that says how to recompile a program can be used in more than one way. The simplest use is to recompile every file that is out of date. Usually, makefiles are written so that if you run `make` with no arguments, it does just that.

But you might want to update only some of the files; you might want to use a different compiler or different compiler options; you might want just to find out which files are out of date without changing them.

By giving arguments when you run `make`, you can do any of these things and many others.

The exit status of `make` is always one of three values:

`0`

The exit status is zero if `make` is successful.

`2`

The exit status is two if `make` encounters any errors. It will print messages describing the particular errors.

`1`

The exit status is one if you use the `-q' flag and `make` determines that some target is not already up to date. See section Instead of Executing the Commands.

Arguments to Specify the Makefile

The way to specify the name of the makefile is with the `-f' or `--file' option (`--makefile' also works). For example, `-f altmake' says to use the file `` `altmake' `` as the makefile.

If you use the `-f' flag several times and follow each `-f' with an argument, all the specified files are used jointly as makefiles.

If you do not use the `-f' or `--file' flag, the default is to try `` `GNUmakefile' ``, `` `makefile' ``, and `` `Makefile' ``, in that order, and use the first of these three which exists or can be made (see section Writing Makefiles).

Arguments to Specify the Goals

The goals are the targets that `make` should strive ultimately to update. Other targets are updated as well if they appear as dependencies of goals, or dependencies of dependencies of goals, etc.

By default, the goal is the first target in the makefile (not counting targets that start with a period). Therefore, makefiles are usually written so that the first target is for compiling the entire program or

programs they describe. If the first rule in the makefile has several targets, only the first target in the becomes the default goal, not the whole list.

You can specify a different goal or goals with arguments to `make`. Use the name of the goal as an argument. If you specify several goals, `make` processes each of them in turn, in the order you name them.

Any target in the makefile may be specified as a goal (unless it starts with `-' or contains an `=', in which case it will be parsed as a switch or variable definition, respectively). Even targets not in the makefile may be specified, if `make` can find implicit rules that say how to make them.

One use of specifying a goal is if you want to compile only a part of the program, or only one of several programs. Specify as a goal each file that you wish to remake. For example, consider a directory containing several programs, with a makefile that starts like this:

```
.PHONY: all
all: size nm ld ar as
```

If you are working on the program `size`, you might want to say `make size' so that only the files of that program are recompiled.

Another use of specifying a goal is to make files that are not normally made. For example, there may be a file of debugging output, or a version of the program that is compiled specially for testing, which has a rule in the makefile but is not a dependency of the default goal.

Another use of specifying a goal is to run the commands associated with a phony target (see section Phony Targets) or empty target (see section Empty Target Files to Record Events). Many makefiles contain a phony target named `` `clean' `` which deletes everything except source files. Naturally, this is done only if you request it explicitly with `make clean'. Following is a list of typical phony and empty target names. See section Standard Targets for Users, for a detailed list of all the standard target names which GNU software packages use.

`` `all' ``

Make all the top-level targets the makefile knows about.

`` `clean' ``

Delete all files that are normally created by running `make`.

`` `mostlyclean' ``

Like `clean', but may refrain from deleting a few files that people normally don't want to recompile. For example, the `mostlyclean' target for GCC does not delete `` `libgcc.a' ``, because recompiling it is rarely necessary and takes a lot of time.

`` `distclean' ``

`` `realclean' ``

`` `clobber' ``

Any of these targets might be defined to delete *more* files than `clean' does. For example, this would delete configuration files or links that you would normally create as preparation for compilation, even if the makefile itself cannot create these files.

`install'
: Copy the executable file into a directory that users typically search for commands; copy any auxiliary files that the executable uses into the directories where it will look for them.

`print'
: Print listings of the source files that have changed.

`tar'
: Create a tar file of the source files.

`shar'
: Create a shell archive (shar file) of the source files.

`dist'
: Create a distribution file of the source files. This might be a tar file, or a shar file, or a compressed version of one of the above, or even more than one of the above.

`TAGS'
: Update a tags table for this program.

`check'

`test'
: Perform self tests on the program this makefile builds.

Instead of Executing the Commands

The makefile tells `make` how to tell whether a target is up to date, and how to update each target. But updating the targets is not always what you want. Certain options specify other activities for `make`.

`-n'

`--just-print'

`--dry-run'

`--recon'
: "No-op". The activity is to print what commands would be used to make the targets up to date, but not actually execute them.

`-t'

`--touch'
: "Touch". The activity is to mark the targets as up to date without actually changing them. In other words, `make` pretends to compile the targets but does not really change their contents.

`-q'

`--question'
: "Question". The activity is to find out silently whether the targets are up to date already; but execute no commands in either case. In other words, neither compilation nor output will occur.

`-W file'

`--what-if=file'

`--assume-new=file'

`--new-file=file'

> "What if". Each `-W' flag is followed by a file name. The given files' modification times are recorded by `make` as being the present time, although the actual modification times remain the same. You can use the `-W' flag in conjunction with the `-n' flag to see what would happen if you were to modify specific files.

With the `-n' flag, `make` prints the commands that it would normally execute but does not execute them.

With the `-t' flag, `make` ignores the commands in the rules and uses (in effect) the command `touch` for each target that needs to be remade. The `touch` command is also printed, unless `-s' or `.SILENT` is used. For speed, `make` does not actually invoke the program `touch`. It does the work directly.

With the `-q' flag, `make` prints nothing and executes no commands, but the exit status code it returns is zero if and only if the targets to be considered are already up to date. If the exit status is one, then some updating needs to be done. If `make` encounters an error, the exit status is two, so you can distinguish an error from a target that is not up to date.

It is an error to use more than one of these three flags in the same invocation of `make`.

The `-n', `-t', and `-q' options do not affect command lines that begin with `+' characters or contain the strings `$(MAKE)' or `${MAKE}'. Note that only the line containing the `+' character or the strings `$(MAKE)' or `${MAKE}' is run regardless of these options. Other lines in the same rule are not run unless they too begin with `+' or contain `$(MAKE)' or `${MAKE}' (See section How the `MAKE` Variable Works.)

The `-W' flag provides two features:

- If you also use the `-n' or `-q' flag, you can see what `make` would do if you were to modify some files.
- Without the `-n' or `-q' flag, when `make` is actually executing commands, the `-W' flag can direct `make` to act as if some files had been modified, without actually modifying the files.

Note that the options `-p' and `-v' allow you to obtain other information about `make` or about the makefiles in use (see section Summary of Options).

Avoiding Recompilation of Some Files

Sometimes you may have changed a source file but you do not want to recompile all the files that depend on it. For example, suppose you add a macro or a declaration to a header file that many other files depend on. Being conservative, `make` assumes that any change in the header file requires recompilation of all dependent files, but you know that they do not need to be recompiled and you would rather not waste the time waiting for them to compile.

If you anticipate the problem before changing the header file, you can use the `-t' flag. This flag tells `make` not to run the commands in the rules, but rather to mark the target up to date by changing its

last-modification date. You would follow this procedure:

1. Use the command `make' to recompile the source files that really need recompilation.
2. Make the changes in the header files.
3. Use the command `make -t' to mark all the object files as up to date. The next time you run `make`, the changes in the header files will not cause any recompilation.

If you have already changed the header file at a time when some files do need recompilation, it is too late to do this. Instead, you can use the `-o file' flag, which marks a specified file as "old" (see section Summary of Options). This means that the file itself will not be remade, and nothing else will be remade on its account. Follow this procedure:

1. Recompile the source files that need compilation for reasons independent of the particular header file, with `make -o headerfile'. If several header files are involved, use a separate `-o' option for each header file.
2. Touch all the object files with `make -t'.

Overriding Variables

An argument that contains `=' specifies the value of a variable: `v=x' sets the value of the variable v to x. If you specify a value in this way, all ordinary assignments of the same variable in the makefile are ignored; we say they have been overridden by the command line argument.

The most common way to use this facility is to pass extra flags to compilers. For example, in a properly written makefile, the variable `CFLAGS` is included in each command that runs the C compiler, so a file `` `foo.c' `` would be compiled something like this:

```
cc -c $(CFLAGS) foo.c
```

Thus, whatever value you set for `CFLAGS` affects each compilation that occurs. The makefile probably specifies the usual value for `CFLAGS`, like this:

```
CFLAGS=-g
```

Each time you run `make`, you can override this value if you wish. For example, if you say `make CFLAGS='-g -O'', each C compilation will be done with `cc -c -g -O'. (This illustrates how you can use quoting in the shell to enclose spaces and other special characters in the value of a variable when you override it.)

The variable `CFLAGS` is only one of many standard variables that exist just so that you can change them this way. See section Variables Used by Implicit Rules, for a complete list.

You can also program the makefile to look at additional variables of your own, giving the user the ability to control other aspects of how the makefile works by changing the variables.

When you override a variable with a command argument, you can define either a recursively-expanded variable or a simply-expanded variable. The examples shown above make a recursively-expanded

variable; to make a simply-expanded variable, write `:=' instead of `='. But, unless you want to include a variable reference or function call in the *value* that you specify, it makes no difference which kind of variable you create.

There is one way that the makefile can change a variable that you have overridden. This is to use the `override` directive, which is a line that looks like this: `override variable = value' (see section The `override` Directive).

Testing the Compilation of a Program

Normally, when an error happens in executing a shell command, `make` gives up immediately, returning a nonzero status. No further commands are executed for any target. The error implies that the goal cannot be correctly remade, and `make` reports this as soon as it knows.

When you are compiling a program that you have just changed, this is not what you want. Instead, you would rather that `make` try compiling every file that can be tried, to show you as many compilation errors as possible.

On these occasions, you should use the `-k' or `--keep-going' flag. This tells `make` to continue to consider the other dependencies of the pending targets, remaking them if necessary, before it gives up and returns nonzero status. For example, after an error in compiling one object file, `make -k' will continue compiling other object files even though it already knows that linking them will be impossible. In addition to continuing after failed shell commands, `make -k' will continue as much as possible after discovering that it does not know how to make a target or dependency file. This will always cause an error message, but without `-k', it is a fatal error (see section Summary of Options).

The usual behavior of `make` assumes that your purpose is to get the goals up to date; once `make` learns that this is impossible, it might as well report the failure immediately. The `-k' flag says that the real purpose is to test as much as possible of the changes made in the program, perhaps to find several independent problems so that you can correct them all before the next attempt to compile. This is why Emacs' M-x compile command passes the `-k' flag by default.

Summary of Options

Here is a table of all the options `make` understands:

`-b'

`-m'

These options are ignored for compatibility with other versions of `make`.

`-C dir'

`--directory=dir'

Change to directory dir before reading the makefiles. If multiple `-C' options are specified, each is interpreted relative to the previous one: `-C / -C etc' is equivalent to `-C /etc'. This is typically used with recursive invocations of `make` (see section Recursive Use of `make`).

`-d'

`--debug'

Print debugging information in addition to normal processing. The debugging information says which files are being considered for remaking, which file-times are being compared and with what results, which files actually need to be remade, which implicit rules are considered and which are applied--everything interesting about how `make` decides what to do.

`-e'

`--environment-overrides'

Give variables taken from the environment precedence over variables from makefiles. See section Variables from the Environment.

`-f file'

`--file=file'

`--makefile=file'

Read the file named file as a makefile. See section Writing Makefiles.

`-h'

`--help'

Remind you of the options that `make` understands and then exit.

`-i'

`--ignore-errors'

Ignore all errors in commands executed to remake files. See section Errors in Commands.

`-I dir'

`--include-dir=dir'

Specifies a directory dir to search for included makefiles. See section Including Other Makefiles. If several `-I' options are used to specify several directories, the directories are searched in the order specified.

`-j [jobs]'

`--jobs=[jobs]'

Specifies the number of jobs (commands) to run simultaneously. With no argument, `make` runs as many jobs simultaneously as possible. If there is more than one `-j' option, the last one is effective. See section Parallel Execution, for more information on how commands are run.

`-k'

`--keep-going'

Continue as much as possible after an error. While the target that failed, and those that depend on it, cannot be remade, the other dependencies of these targets can be processed all the same. See section Testing the Compilation of a Program.

`-l [load]'

`--load-average[=load]'

`--max-load[=load]'

Specifies that no new jobs (commands) should be started if there are other jobs running and the load average is at least load (a floating-point number). With no argument, removes a previous load limit. See section Parallel Execution.

`-n'

`--just-print'

`--dry-run'

`--recon'

Print the commands that would be executed, but do not execute them. See section Instead of Executing the Commands.

`-o file'

`--old-file=file'

`--assume-old=file'

Do not remake the file file even if it is older than its dependencies, and do not remake anything on account of changes in file. Essentially the file is treated as very old and its rules are ignored. See section Avoiding Recompilation of Some Files.

`-p'

`--print-data-base'

Print the data base (rules and variable values) that results from reading the makefiles; then execute as usual or as otherwise specified. This also prints the version information given by the `-v' switch (see below). To print the data base without trying to remake any files, use `make -p -f /dev/null'.

`-q'

`--question'

"Question mode". Do not run any commands, or print anything; just return an exit status that is zero if the specified targets are already up to date, one if any remaking is required, or two if an error is encountered. See section Instead of Executing the Commands.

`-r'

`--no-builtin-rules'

Eliminate use of the built-in implicit rules (see section Using Implicit Rules). You can still define your own by writing pattern rules (see section Defining and Redefining Pattern Rules). The `-r' option also clears out the default list of suffixes for suffix rules (see section Old-Fashioned Suffix Rules). But you can still define your own suffixes with a rule for `.SUFFIXES`, and then define your own suffix rules.

`-s'

`--silent'

`--quiet'

Silent operation; do not print the commands as they are executed. See section Command Echoing.

`-S'

`--no-keep-going'

`--stop'

Cancel the effect of the `-k' option. This is never necessary except in a recursive `make` where `-k' might be inherited from the top-level `make` via `MAKEFLAGS` (see section Recursive Use of `make`) or if you set `-k' in `MAKEFLAGS` in your environment.

`-t'

`--touch'

Touch files (mark them up to date without really changing them) instead of running their commands. This is used to pretend that the commands were done, in order to fool future invocations of `make`. See section Instead of Executing the Commands.

`-v'

`--version'

Print the version of the `make` program plus a copyright, a list of authors, and a notice that there is no warranty; then exit.

`-w'

`--print-directory'

Print a message containing the working directory both before and after executing the makefile. This may be useful for tracking down errors from complicated nests of recursive `make` commands. See section Recursive Use of `make`. (In practice, you rarely need to specify this option since `make' does it for you; see section The `--print-directory' Option.)

`--no-print-directory'

Disable printing of the working directory under `-w`. This option is useful when `-w` is turned on automatically, but you do not want to see the extra messages. See section The `--print-directory' Option.

`-W file'

`--what-if=file'

`--new-file=file'

`--assume-new=file'

Pretend that the target file has just been modified. When used with the `-n' flag, this shows you what would happen if you were to modify that file. Without `-n', it is almost the same as running a `touch` command on the given file before running `make`, except that the modification time is changed only in the imagination of `make`. See section Instead of Executing the Commands.

`--warn-undefined-variables'

Issue a warning message whenever `make` sees a reference to an undefined variable. This can be helpful when you are trying to debug makefiles which use variables in complex ways.

Go to the previous, next section.

Go to the previous, next section.

Using Implicit Rules

Certain standard ways of remaking target files are used very often. For example, one customary way to make an object file is from a C source file using the C compiler, `cc`.

Implicit rules tell `make` how to use customary techniques so that you do not have to specify them in detail when you want to use them. For example, there is an implicit rule for C compilation. File names determine which implicit rules are run. For example, C compilation typically takes a `.c' file and makes a `.o' file. So `make` applies the implicit rule for C compilation when it sees this combination of file name endings.

A chain of implicit rules can apply in sequence; for example, `make` will remake a `.o' file from a `.y' file by way of a `.c' file. See section Chains of Implicit Rules.

The built-in implicit rules use several variables in their commands so that, by changing the values of the variables, you can change the way the implicit rule works. For example, the variable `CFLAGS` controls the flags given to the C compiler by the implicit rule for C compilation. See section Variables Used by Implicit Rules.

You can define your own implicit rules by writing pattern rules. See section Defining and Redefining Pattern Rules.

Suffix rules are a more limited way to define implicit rules. Pattern rules are more general and clearer, but suffix rules are retained for compatibility. See section Old-Fashioned Suffix Rules.

Using Implicit Rules

To allow `make` to find a customary method for updating a target file, all you have to do is refrain from specifying commands yourself. Either write a rule with no command lines, or don't write a rule at all. Then `make` will figure out which implicit rule to use based on which kind of source file exists or can be made.

For example, suppose the makefile looks like this:

```
foo : foo.o bar.o
        cc -o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

Because you mention `foo.o' but do not give a rule for it, `make` will automatically look for an implicit rule that tells how to update it. This happens whether or not the file `foo.o' currently exists.

If an implicit rule is found, it can supply both commands and one or more dependencies (the source files). You would want to write a rule for `foo.o' with no command lines if you need to specify additional dependencies, such as header files, that the implicit rule cannot supply.

Each implicit rule has a target pattern and dependency patterns. There may be many implicit rules wi the same target pattern. For example, numerous rules make `.o' files: one, from a `.c' file with the C compiler; another, from a `.p' file with the Pascal compiler; and so on. The rule that actually applies is the one whose dependencies exist or can be made. So, if you have a file `foo.c', make will run the C compiler; otherwise, if you have a file `foo.p', make will run the Pascal compiler; and so on.

Of course, when you write the makefile, you know which implicit rule you want make to use, and you know it will choose that one because you know which possible dependency files are supposed to exist. See section Catalogue of Implicit Rules, for a catalogue of all the predefined implicit rules.

Above, we said an implicit rule applies if the required dependencies "exist or can be made". A file "can be made" if it is mentioned explicitly in the makefile as a target or a dependency, or if an implicit rule can be recursively found for how to make it. When an implicit dependency is the result of another implicit rule, we say that chaining is occurring. See section Chains of Implicit Rules.

In general, make searches for an implicit rule for each target, and for each double-colon rule, that has no commands. A file that is mentioned only as a dependency is considered a target whose rule specifies nothing, so implicit rule search happens for it. See section Implicit Rule Search Algorithm, for the details of how the search is done.

Note that explicit dependencies do not influence implicit rule search. For example, consider this explicit rule:

```
foo.o: foo.p
```

The dependency on `foo.p' does not necessarily mean that make will remake `foo.o' according to the implicit rule to make an object file, a `.o' file, from a Pascal source file, a `.p' file. For example, if `foo.c' also exists, the implicit rule to make an object file from a C source file is used instead, because it appears before the Pascal rule in the list of predefined implicit rules (see section Catalogue of Implicit Rules).

If you do not want an implicit rule to be used for a target that has no commands, you can give that target empty commands by writing a semicolon (see section Using Empty Commands).

Catalogue of Implicit Rules

Here is a catalogue of predefined implicit rules which are always available unless the makefile explicitly overrides or cancels them. See section Canceling Implicit Rules, for information on canceling or overriding an implicit rule. The `-r' or `--no-builtin-rules' option cancels all predefined rules.

Not all of these rules will always be defined, even when the `-r' option is not given. Many of the predefined implicit rules are implemented in make as suffix rules, so which ones will be defined depends on the suffix list (the list of dependencies of the special target .SUFFIXES). The default suffix list is: .out, .a, .ln, .o, .c, .cc, .C, .p, .f, .F, .r, .y, .l, .s, .S, .mod, .sym, .def, .h, .info, .dvi, .tex, .texinfo, .texi, .txinfo, .w, .ch .web, .sh, .elc, .el. All of the implicit rules described below whose dependencies have one of these suffixes are actually suffix rules. If you

modify the suffix list, the only predefined suffix rules in effect will be those named by one or two of the suffixes that are on the list you specify; rules whose suffixes fail to be on the list are disabled. See section Old-Fashioned Suffix Rules, for full details on suffix rules.

Compiling C programs

`n.o' is made automatically from `n.c' with a command of the form `$(CC) -c $(CPPFLAGS) $(CFLAGS)'.

Compiling C++ programs

`n.o' is made automatically from `n.cc' or `n.C' with a command of the form `$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)'. We encourage you to use the suffix `.cc' for C++ source files instead of `.C'.

Compiling Pascal programs

`n.o' is made automatically from `n.p' with the command `$(PC) -c $(PFLAGS)'.

Compiling Fortran and Ratfor programs

`n.o' is made automatically from `n.r', `n.F' or `n.f' by running the Fortran compiler. The precise command used is as follows:

`.f'

`$(FC) -c $(FFLAGS)'.

`.F'

`$(FC) -c $(FFLAGS) $(CPPFLAGS)'.

`.r'

`$(FC) -c $(FFLAGS) $(RFLAGS)'.

- Preprocessing Fortran and Ratfor programs `n.f' is made automatically from `n.r' or `n.F'. This rule runs just the preprocessor to convert a Ratfor or preprocessable Fortran program into a strict Fortran program. The precise command used is as follows:

`.F'

`$(FC) -F $(CPPFLAGS) $(FFLAGS)'.

`.r'

`$(FC) -F $(FFLAGS) $(RFLAGS)'.

- Compiling Modula-2 programs `n.sym' is made from `n.def' with a command of the form `$(M2C) $(M2FLAGS) $(DEFFLAGS)'. `n.o' is made from `n.mod'; the form is: `$(M2C) $(M2FLAGS) $(MODFLAGS)'.

- Assembling and preprocessing assembler programs `n.o' is made automatically from `n.s' by running the assembler, as. The precise command is `$(AS) $(ASFLAGS)'.

`n.s' is made automatically from `n.S' by running the C preprocessor, cpp. The precise command is `$(CPP) $(CPPFLAGS)'.

- Linking a single object file `n' is made automatically from `n.o' by running the linker (usually called ld) via the C compiler. The precise command used is `$(CC) $(LDFLAGS) n.o $(LOADLIBES)'.

This rule does the right thing for a simple program with only one source file. It will also do the right thing if there are multiple object files (presumably coming from various other source files), one of which

has a name matching that of the executable file. Thus,

```
x: y.o z.o
```

when `x.c', `y.c' and `z.c' all exist will execute:

```
cc -c x.c -o x.o
cc -c y.c -o y.o
cc -c z.c -o z.o
cc x.o y.o z.o -o x
rm -f x.o
rm -f y.o
rm -f z.o
```

In more complicated cases, such as when there is no object file whose name derives from the executable file name, you must write an explicit command for linking.

Each kind of file automatically made into `.o' object files will be automatically linked by using the compiler (`$(CC)', `$(FC)' or `$(PC)'; the C compiler `$(CC)' is used to assemble `.s' files) without the `-c' option. This could be done by using the `.o' object files as intermediates, but it is faster to do the compiling and linking in one step, so that's how it's done.

- Yacc for C programs `n.c' is made automatically from `n.y' by running Yacc with the command `$(YACC) $(YFLAGS)'.
- Lex for C programs `n.c' is made automatically from `n.l' by by running Lex. The actual command is `$(LEX) $(LFLAGS)'.
- Lex for Ratfor programs `n.r' is made automatically from `n.l' by by running Lex. The actual command is `$(LEX) $(LFLAGS)'.

The convention of using the same suffix `.l' for all Lex files regardless of whether they produce C code or Ratfor code makes it impossible for `make` to determine automatically which of the two languages you are using in any particular case. If `make` is called upon to remake an object file from a `.l' file, it must guess which compiler to use. It will guess the C compiler, because that is more common. If you are using Ratfor, make sure `make` knows this by mentioning `n.r' in the makefile. Or, if you are using Ratfor exclusively, with no C files, remove `.c' from the list of implicit rule suffixes with:

```
.SUFFIXES:
.SUFFIXES: .o .r .f .l ...
```

- Making Lint Libraries from C, Yacc, or Lex programs `n.ln' is made from `n.c' by running `lint`. The precise command is `$(LINT) $(LINTFLAGS) $(CPPFLAGS) -i'. The same command is used on the C code produced from `n.y' or `n.l'.
- TeX and Web `n.dvi' is made from `n.tex' with the command `$(TEX)'. `n.tex' is made from `n.web' with `$(WEAVE)', or from `n.w' (and from `n.ch' if it exists or can be made) with `$(CWEAVE)'. `n.p' is made from `n.web' with `$(TANGLE)' and `n.c' is made from `n.w' (and from `n.ch' if it exists or can be made) with `$(CTANGLE)'.
- Texinfo and Info `n.dvi' is made from `n.texinfo', `n.texi', or `n.txinfo', with the

command `$(TEXI2DVI) $(TEXI2DVI_FLAGS)'. `n.info' is made from `n.texinfo', `n.texi', or `n.txinfo', with the command `$(MAKEINFO) $(MAKEINFO_FLAGS)'.

- RCS Any file `n' is extracted if necessary from an RCS file named either `n,v' or `RCS/n,v'. The precise command used is `$(CO) $(COFLAGS)'. `n' will not be extracted from RCS if it already exists, even if the RCS file is newer. The rules for RCS are terminal (see section Match-Anything Pattern Rules), so RCS files cannot be generated from another source; they must actually exist.

- SCCS Any file `n' is extracted if necessary from an SCCS file named either `s.n' or `SCCS/s.n'. The precise command used is `$(GET) $(GFLAGS)'. The rules for SCCS are terminal (see section Match-Anything Pattern Rules), so SCCS files cannot be generated from another source; they must actually exist.

For the benefit of SCCS, a file `n' is copied from `n.sh' and made executable (by everyone). This is for shell scripts that are checked into SCCS. Since RCS preserves the execution permission of a file, you do not need to use this feature with RCS.

We recommend that you avoid using of SCCS. RCS is widely held to be superior, and is also free. By choosing free software in place of comparable (or inferior) proprietary software, you support the free software movement.

Usually, you want to change only the variables listed in the table above, which are documented in the following section.

However, the commands in built-in implicit rules actually use variables such as COMPILE.c, LINK.p, and PREPROCESS.S, whose values contain the commands listed above.

make follows the convention that the rule to compile a `.x' source file uses the variable COMPILE.x. Similarly, the rule to produce an executable from a `.x' file uses LINK.x; and the rule to preprocess a `.x' file uses PREPROCESS.x.

Every rule that produces an object file uses the variable OUTPUT_OPTION. make defines this variable either to contain `-o $@', or to be empty, depending on a compile-time option. You need the `-o' option to ensure that the output goes into the right file when the source file is in a different directory, as when using VPATH (see section Searching Directories for Dependencies). However, compilers on some systems do not accept a `-o' switch for object files. If you use such a system, and use VPATH, some compilations will put their output in the wrong place. A possible workaround for this problem is to give OUTPUT_OPTION the value `; mv $*.o $@'.

Variables Used by Implicit Rules

The commands in built-in implicit rules make liberal use of certain predefined variables. You can alter these variables in the makefile, with arguments to make, or in the environment to alter how the implicit rules work without redefining the rules themselves.

For example, the command used to compile a C source file actually says `$(CC) -c $(CFLAGS) $(CPPFLAGS)'. The default values of the variables used are `cc' and nothing, resulting in the command `cc -c'. By redefining `CC' to `ncc', you could cause `ncc' to be used for all C compilations performed by

the implicit rule. By redefining `CFLAGS' to be `-g', you could pass the `-g' option to each compilati
All implicit rules that do C compilation use `$(CC)' to get the program name for the compiler and *all* include `$(CFLAGS)' among the arguments given to the compiler.

The variables used in implicit rules fall into two classes: those that are names of programs (like `CC`) and those that contain arguments for the programs (like `CFLAGS`). (The "name of a program" may also contain some command arguments, but it must start with an actual executable program name.) If a variable value contains more than one argument, separate them with spaces.

Here is a table of variables used as names of programs in built-in rules:

`AR`
: Archive-maintaining program; default `ar'.

`AS`
: Program for doing assembly; default `as'.

`CC`
: Program for compiling C programs; default `cc'.

`CXX`
: Program for compiling C++ programs; default `g++'.

`CO`
: Program for extracting a file from RCS; default `co'.

`CPP`
: Program for running the C preprocessor, with results to standard output; default `$(CC) -E'.

`FC`
: Program for compiling or preprocessing Fortran and Ratfor programs; default `f77'.

`GET`
: Program for extracting a file from SCCS; default `get'.

`LEX`
: Program to use to turn Lex grammars into C programs or Ratfor programs; default `lex'.

`PC`
: Program for compiling Pascal programs; default `pc'.

`YACC`
: Program to use to turn Yacc grammars into C programs; default `yacc'.

`YACCR`
: Program to use to turn Yacc grammars into Ratfor programs; default `yacc -r'.

`MAKEINFO`
: Program to convert a Texinfo source file into an Info file; default `makeinfo'.

`TEX`
: Program to make TeX DVI files from TeX source; default `tex'.

`TEXI2DVI`

Program to make TeX DVI files from Texinfo source; default `texi2dvi'.

`WEAVE`

Program to translate Web into TeX; default `weave'.

`CWEAVE`

Program to translate C Web into TeX; default `cweave'.

`TANGLE`

Program to translate Web into Pascal; default `tangle'.

`CTANGLE`

Program to translate C Web into C; default `ctangle'.

`RM`

Command to remove a file; default `rm -f'.

Here is a table of variables whose values are additional arguments for the programs above. The default values for all of these is the empty string, unless otherwise noted.

`ARFLAGS`

Flags to give the archive-maintaining program; default `rv'.

`ASFLAGS`

Extra flags to give to the assembler (when explicitly invoked on a `.s' or `.S' file).

`CFLAGS`

Extra flags to give to the C compiler.

`CXXFLAGS`

Extra flags to give to the C++ compiler.

`COFLAGS`

Extra flags to give to the RCS `co` program.

`CPPFLAGS`

Extra flags to give to the C preprocessor and programs that use it (the C and Fortran compilers).

`FFLAGS`

Extra flags to give to the Fortran compiler.

`GFLAGS`

Extra flags to give to the SCCS `get` program.

`LDFLAGS`

Extra flags to give to compilers when they are supposed to invoke the linker, `ld'.

`LFLAGS`

Extra flags to give to Lex.

`PFLAGS`

Extra flags to give to the Pascal compiler.

`RFLAGS`

Extra flags to give to the Fortran compiler for Ratfor programs.

YFLAGS

Extra flags to give to Yacc.

Chains of Implicit Rules

Sometimes a file can be made by a sequence of implicit rules. For example, a file `n.o' could be made from `n.y' by running first Yacc and then cc. Such a sequence is called a chain.

If the file `n.c' exists, or is mentioned in the makefile, no special searching is required: make finds that the object file can be made by C compilation from `n.c'; later on, when considering how to make `n.c', the rule for running Yacc is used. Ultimately both `n.c' and `n.o' are updated.

However, even if `n.c' does not exist and is not mentioned, make knows how to envision it as the missing link between `n.o' and `n.y'! In this case, `n.c' is called an intermediate file. Once make has decided to use the intermediate file, it is entered in the data base as if it had been mentioned in the makefile, along with the implicit rule that says how to create it.

Intermediate files are remade using their rules just like all other files. The difference is that the intermediate file is deleted when make is finished. Therefore, the intermediate file which did not exist before make also does not exist after make. The deletion is reported to you by printing a `rm -f' command that shows what make is doing. (You can list the target pattern of an implicit rule (such as `%.o') as a dependency of the special target .PRECIOUS to preserve intermediate files made by implicit rules whose target patterns match that file's name; see section Interrupting or Killing make.)

A chain can involve more than two implicit rules. For example, it is possible to make a file `foo' from `RCS/foo.y,v' by running RCS, Yacc and cc. Then both `foo.y' and `foo.c' are intermediate files that are deleted at the end.

No single implicit rule can appear more than once in a chain. This means that make will not even consider such a ridiculous thing as making `foo' from `foo.o.o' by running the linker twice. This constraint has the added benefit of preventing any infinite loop in the search for an implicit rule chain.

There are some special implicit rules to optimize certain cases that would otherwise be handled by rule chains. For example, making `foo' from `foo.c' could be handled by compiling and linking with separate chained rules, using `foo.o' as an intermediate file. But what actually happens is that a special rule for this case does the compilation and linking with a single cc command. The optimized rule is used in preference to the step-by-step chain because it comes earlier in the ordering of rules.

Defining and Redefining Pattern Rules

You define an implicit rule by writing a pattern rule. A pattern rule looks like an ordinary rule, except that its target contains the character `%' (exactly one of them). The target is considered a pattern for matching file names; the `%' can match any nonempty substring, while other characters match only themselves. The dependencies likewise use `%' to show how their names relate to the target name.

Thus, a pattern rule `%.o : %.c' says how to make any file `stem.o' from another file `stem.c'.

Note that expansion using `%' in pattern rules occurs **after** any variable or function expansions, which take place when the makefile is read. See section How to Use Variables, and section Functions for Transforming Text.

Introduction to Pattern Rules

A pattern rule contains the character `%' (exactly one of them) in the target; otherwise, it looks exactly like an ordinary rule. The target is a pattern for matching file names; the `%' matches any nonempty substring, while other characters match only themselves.

For example, `%.c' as a pattern matches any file name that ends in `.c'. `s.%.c' as a pattern matches any file name that starts with `s.', ends in `.c' and is at least five characters long. (There must be at least one character to match the `%'.) The substring that the `%' matches is called the stem.

`%' in a dependency of a pattern rule stands for the same stem that was matched by the `%' in the target. In order for the pattern rule to apply, its target pattern must match the file name under consideration, and its dependency patterns must name files that exist or can be made. These files become dependencies of the target.

Thus, a rule of the form

```
%.o : %.c ; command...
```

specifies how to make a file `n.o', with another file `n.c' as its dependency, provided that `n.c' exists or can be made.

There may also be dependencies that do not use `%'; such a dependency attaches to every file made by this pattern rule. These unvarying dependencies are useful occasionally.

A pattern rule need not have any dependencies that contain `%', or in fact any dependencies at all. Such a rule is effectively a general wildcard. It provides a way to make any file that matches the target pattern. See section Defining Last-Resort Default Rules.

Pattern rules may have more than one target. Unlike normal rules, this does not act as many different rules with the same dependencies and commands. If a pattern rule has multiple targets, `make` knows that the rule's commands are responsible for making all of the targets. The commands are executed only once to make all the targets. When searching for a pattern rule to match a target, the target patterns of a rule other than the one that matches the target in need of a rule are incidental: `make` worries only about giving commands and dependencies to the file presently in question. However, when this file's commands are run, the other targets are marked as having been updated themselves.

The order in which pattern rules appear in the makefile is important since this is the order in which they are considered. Of equally applicable rules, only the first one found is used. The rules you write take precedence over those that are built in. Note however, that a rule whose dependencies actually exist or are mentioned always takes priority over a rule with dependencies that must be made by chaining other implicit rules.

Pattern Rule Examples

Here are some examples of pattern rules actually predefined in `make`. First, the rule that compiles `.c' files into `.o' files:

```
%.o : %.c
        $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

defines a rule that can make any file `x.o' from `x.c'. The command uses the automatic variables `$@' and `$<' to substitute the names of the target file and the source file in each case where the rule applies (see section Automatic Variables).

Here is a second built-in rule:

```
% :: RCS/%,v
        $(CO) $(COFLAGS) $<
```

defines a rule that can make any file `x' whatsoever from a corresponding file `x,v' in the subdirectory `RCS'. Since the target is `%', this rule will apply to any file whatever, provided the appropriate dependency file exists. The double colon makes the rule terminal, which means that its dependency may not be an intermediate file (see section Match-Anything Pattern Rules).

This pattern rule has two targets:

```
%.tab.c %.tab.h: %.y
        bison -d $<
```

This tells `make` that the command `bison -d x.y' will make both `x.tab.c' and `x.tab.h'. If the file `foo' depends on the files `parse.tab.o' and `scan.o' and the file `scan.o' depends on the file `parse.tab.h', when `parse.y' is changed, the command `bison -d parse.y' will be executed only once, and the dependencies of both `parse.tab.o' and `scan.o' will be satisfied. (Presumably the file `parse.tab.o' will be recompiled from `parse.tab.c' and the file `scan.o' from `scan.c', while `foo' is linked from `parse.tab.o', `scan.o', and its other dependencies, and it will execute happily ever after.)

Automatic Variables

Suppose you are writing a pattern rule to compile a `.c' file into a `.o' file: how do you write the `cc' command so that it operates on the right source file name? You cannot write the name in the command, because the name is different each time the implicit rule is applied.

What you do is use a special feature of `make`, the automatic variables. These variables have values computed afresh for each rule that is executed, based on the target and dependencies of the rule. In this example, you would use `$@' for the object file name and `$<' for the source file name.

Here is a table of automatic variables:

$@

The file name of the target of the rule. If the target is an archive member, then `$@' is the name of the archive file. In a pattern rule that has multiple targets (see section Introduction to Pattern Rules), `$@' is the name of whichever target caused the rule's commands to be run.

$%

The target member name, when the target is an archive member. See section Using `make` to Update Archive Files. For example, if the target is `` `foo.a(bar.o)' `` then `$%' is `` `bar.o' `` and `$@' is `` `foo.a' ``. `$%' is empty when the target is not an archive member.

$<

The name of the first dependency. If the target got its commands from an implicit rule, this will be the first dependency added by the implicit rule (see section Using Implicit Rules).

$?

The names of all the dependencies that are newer than the target, with spaces between them. For dependencies which are archive members, only the member named is used (see section Using `make` to Update Archive Files).

$^

The names of all the dependencies, with spaces between them. For dependencies which are archive members, only the member named is used (see section Using `make` to Update Archive Files). A target has only one dependency on each other file it depends on, no matter how many times each file is listed as a dependency. So if you list a dependency more than once for a target, the value of $^ contains just one copy of the name.

$+

This is like `$^', but dependencies listed more than once are duplicated in the order they were listed in the makefile. This is primarily useful for use in linking commands where it is meaningful to repeat library file names in a particular order.

$*

The stem with which an implicit rule matches (see section How Patterns Match). If the target is `` `dir/a.foo.b' `` and the target pattern is `` `a.%.b' `` then the stem is `` `dir/foo' ``. The stem is useful for constructing names of related files.

In a static pattern rule, the stem is part of the file name that matched the `%' in the target pattern.

In an explicit rule, there is no stem; so `$*' cannot be determined in that way. Instead, if the target name ends with a recognized suffix (see section Old-Fashioned Suffix Rules), `$*' is set to the target name minus the suffix. For example, if the target name is `foo.c', then `$*' is set to `foo', since `.c' is a suffix. GNU `make` does this bizarre thing only for compatibility with other implementations of `make`. You should generally avoid using `$*' except in implicit rules or static pattern rules.

If the target name in an explicit rule does not end with a recognized suffix, `$*' is set to the empty string for that rule.

`$?' is useful even in explicit rules when you wish to operate on only the dependencies that have changed. For example, suppose that an archive named `lib' is supposed to contain copies of several object files. This rule copies just the changed object files into the archive:

```
lib: foo.o bar.o lose.o win.o
        ar r lib $?
```

Of the variables listed above, four have values that are single file names, and two have values that are lists of file names. These six have variants that get just the file's directory name or just the file name within the directory. The variant variables' names are formed by appending `D' or `F', respectively. These variants are semi-obsolete in GNU `make` since the functions `dir` and `notdir` can be used to get a similar effect (see section Functions for File Names). Note, however, that the `F' variants all omit the trailing slash which always appears in the output of the `dir` function. Here is a table of the variants:

`$(@D)'

The directory part of the file name of the target, with the trailing slash removed. If the value of `$@' is `dir/foo.o' then `$(@D)' is `dir'. This value is `.' if `$@' does not contain a slash.

`$(@F)'

The file-within-directory part of the file name of the target. If the value of `$@' is `dir/foo.o' then `$(@F)' is `foo.o'. `$(@F)' is equivalent to `$(notdir $@)'.

`$(*D)'

`$(*F)'

The directory part and the file-within-directory part of the stem; `dir' and `foo' in this example.

`$(%D)'

`$(%F)'

The directory part and the file-within-directory part of the target archive member name. This makes sense only for archive member targets of the form `archive(member)' and is useful only when member may contain a directory name. (See section Archive Members as Targets.)

`$(<D)'

`$(<F)'

The directory part and the file-within-directory part of the first dependency.

`$(^D)'

`$(^F)'

Lists of the directory parts and the file-within-directory parts of all dependencies.

`$(?D)'

`$(?F)'

Lists of the directory parts and the file-within-directory parts of all dependencies that are newer than the target.

Note that we use a special stylistic convention when we talk about these automatic variables; we write

"the value of `$<'", rather than "the variable `<'" as we would write for ordinary variables such as `objects` and `CFLAGS`. We think this convention looks more natural in this special case. Please do assume it has a deep significance; `$<' refers to the variable named `<` just as `$(CFLAGS)' refers to the variable named `CFLAGS`. You could just as well use `$(<)' in place of `$<'.

How Patterns Match

A target pattern is composed of a `%' between a prefix and a suffix, either or both of which may be empty. The pattern matches a file name only if the file name starts with the prefix and ends with the suffix, without overlap. The text between the prefix and the suffix is called the stem. Thus, when the pattern `%.o' matches the file name `test.o', the stem is `test'. The pattern rule dependencies are turned into actual file names by substituting the stem for the character `%'. Thus, if in the same example one of the dependencies is written as `%.c', it expands to `test.c'.

When the target pattern does not contain a slash (and it usually does not), directory names in the file names are removed from the file name before it is compared with the target prefix and suffix. After the comparison of the file name to the target pattern, the directory names, along with the slash that ends them, are added on to the dependency file names generated from the pattern rule's dependency patterns and the file name. The directories are ignored only for the purpose of finding an implicit rule to use, not in the application of that rule. Thus, `e%t' matches the file name `src/eat', with `src/a' as the stem. When dependencies are turned into file names, the directories from the stem are added at the front, while the rest of the stem is substituted for the `%'. The stem `src/a' with a dependency pattern `c%r' gives the file name `src/car'.

Match-Anything Pattern Rules

When a pattern rule's target is just `%', it matches any file name whatever. We call these rules match-anything rules. They are very useful, but it can take a lot of time for `make` to think about them, because it must consider every such rule for each file name listed either as a target or as a dependency.

Suppose the makefile mentions `foo.c'. For this target, `make` would have to consider making it by linking an object file `foo.c.o', or by C compilation-and-linking in one step from `foo.c.c', or by Pascal compilation-and-linking from `foo.c.p', and many other possibilities.

We know these possibilities are ridiculous since `foo.c' is a C source file, not an executable. If `make` did consider these possibilities, it would ultimately reject them, because files such as `foo.c.o' and `foo.c.p' would not exist. But these possibilities are so numerous that `make` would run very slowly if it had to consider them.

To gain speed, we have put various constraints on the way `make` considers match-anything rules. There are two different constraints that can be applied, and each time you define a match-anything rule you must choose one or the other for that rule.

One choice is to mark the match-anything rule as terminal by defining it with a double colon. When a rule is terminal, it does not apply unless its dependencies actually exist. Dependencies that could be made with other implicit rules are not good enough. In other words, no further chaining is allowed beyond a terminal rule.

For example, the built-in implicit rules for extracting sources from RCS and SCCS files are terminal; as a result, if the file `foo.c,v' does not exist, make will not even consider trying to make it as an intermediate file from `foo.c,v.o' or from `RCS/SCCS/s.foo.c,v'. RCS and SCCS files are generally ultimate source files, which should not be remade from any other files; therefore, make can save time by not looking for ways to remake them.

If you do not mark the match-anything rule as terminal, then it is nonterminal. A nonterminal match-anything rule cannot apply to a file name that indicates a specific type of data. A file name indicates a specific type of data if some non-match-anything implicit rule target matches it.

For example, the file name `foo.c' matches the target for the pattern rule `%.c : %.y' (the rule to run Yacc). Regardless of whether this rule is actually applicable (which happens only if there is a file `foo.y'), the fact that its target matches is enough to prevent consideration of any nonterminal match-anything rules for the file `foo.c'. Thus, make will not even consider trying to make `foo.c' as an executable file from `foo.c.o', `foo.c.c', `foo.c.p', etc.

The motivation for this constraint is that nonterminal match-anything rules are used for making files containing specific types of data (such as executable files) and a file name with a recognized suffix indicates some other specific type of data (such as a C source file).

Special built-in dummy pattern rules are provided solely to recognize certain file names so that nonterminal match-anything rules will not be considered. These dummy rules have no dependencies and no commands, and they are ignored for all other purposes. For example, the built-in implicit rule

```
%.p :
```

exists to make sure that Pascal source files such as `foo.p' match a specific target pattern and thereby prevent time from being wasted looking for `foo.p.o' or `foo.p.c'.

Dummy pattern rules such as the one for `%.p' are made for every suffix listed as valid for use in suffix rules (see section Old-Fashioned Suffix Rules).

Canceling Implicit Rules

You can override a built-in implicit rule (or one you have defined yourself) by defining a new pattern rule with the same target and dependencies, but different commands. When the new rule is defined, the built-in one is replaced. The new rule's position in the sequence of implicit rules is determined by where you write the new rule.

You can cancel a built-in implicit rule by defining a pattern rule with the same target and dependencies, but no commands. For example, the following would cancel the rule that runs the assembler:

```
%.o : %.s
```

Defining Last-Resort Default Rules

You can define a last-resort implicit rule by writing a terminal match-anything pattern rule with no dependencies (see section Match-Anything Pattern Rules). This is just like any other pattern rule; the only thing special about it is that it will match any target. So such a rule's commands are used for all targets and dependencies that have no commands of their own and for which no other implicit rule applies.

For example, when testing a makefile, you might not care if the source files contain real data, only that they exist. Then you might do this:

```
%::
        touch $@
```

to cause all the source files needed (as dependencies) to be created automatically.

You can instead define commands to be used for targets for which there are no rules at all, even ones which don't specify commands. You do this by writing a rule for the target `.DEFAULT`. Such a rule's commands are used for all dependencies which do not appear as targets in any explicit rule, and for which no implicit rule applies. Naturally, there is no `.DEFAULT` rule unless you write one.

If you use `.DEFAULT` with no commands or dependencies:

```
.DEFAULT:
```

the commands previously stored for `.DEFAULT` are cleared. Then `make` acts as if you had never defined `.DEFAULT` at all.

If you do not want a target to get the commands from a match-anything pattern rule or `.DEFAULT`, but you also do not want any commands to be run for the target, you can give it empty commands (see section Using Empty Commands).

You can use a last-resort rule to override part of another makefile. See section Overriding Part of Another Makefile.

Old-Fashioned Suffix Rules

Suffix rules are the old-fashioned way of defining implicit rules for `make`. Suffix rules are obsolete because pattern rules are more general and clearer. They are supported in GNU `make` for compatibility with old makefiles. They come in two kinds: double-suffix and single-suffix.

A double-suffix rule is defined by a pair of suffixes: the target suffix and the source suffix. It matches any file whose name ends with the target suffix. The corresponding implicit dependency is made by replacing the target suffix with the source suffix in the file name. A two-suffix rule whose target and source suffixes are `.o' and `.c' is equivalent to the pattern rule `%.o : %.c'.

A single-suffix rule is defined by a single suffix, which is the source suffix. It matches any file name, and the corresponding implicit dependency name is made by appending the source suffix. A single-suffix rule whose source suffix is `.c' is equivalent to the pattern rule `% : %.c'.

Suffix rule definitions are recognized by comparing each rule's target against a defined list of known suffixes. When `make` sees a rule whose target is a known suffix, this rule is considered a single-suffix rule. When `make` sees a rule whose target is two known suffixes concatenated, this rule is taken as a double-suffix rule.

For example, `.c' and `.o' are both on the default list of known suffixes. Therefore, if you define a rule whose target is `.c.o', `make` takes it to be a double-suffix rule with source suffix `.c' and target suffix `.o'. Here is the old-fashioned way to define the rule for compiling a C source file:

```
.c.o:
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

Suffix rules cannot have any dependencies of their own. If they have any, they are treated as normal files with funny names, not as suffix rules. Thus, the rule:

```
.c.o: foo.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

tells how to make the file `.c.o' from the dependency file `foo.h', and is not at all like the pattern rule:

```
%.o: %.c foo.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

which tells how to make `.o' files from `.c' files, and makes all `.o' files using this pattern rule also depend on `foo.h'.

Suffix rules with no commands are also meaningless. They do not remove previous rules as do pattern rules with no commands (see section Canceling Implicit Rules). They simply enter the suffix or pair of suffixes concatenated as a target in the data base.

The known suffixes are simply the names of the dependencies of the special target `.SUFFIXES`. You can add your own suffixes by writing a rule for `.SUFFIXES` that adds more dependencies, as in:

```
.SUFFIXES: .hack .win
```

which adds `.hack' and `.win' to the end of the list of suffixes.

If you wish to eliminate the default known suffixes instead of just adding to them, write a rule for `.SUFFIXES` with no dependencies. By special dispensation, this eliminates all existing dependencies of `.SUFFIXES`. You can then write another rule to add the suffixes you want. For example,

```
.SUFFIXES:            # Delete the default suffixes
```


```
.SUFFIXES: .c .o .h   # Define our suffix list
```

The `-r' or `--no-builtin-rules' flag causes the default list of suffixes to be empty.

The variable `SUFFIXES` is defined to the default list of suffixes before `make` reads any makefiles. You can change the list of suffixes with a rule for the special target `.SUFFIXES`, but that does not alter this variable.

Implicit Rule Search Algorithm

Here is the procedure `make` uses for searching for an implicit rule for a target t. This procedure is followed for each double-colon rule with no commands, for each target of ordinary rules none of which have commands, and for each dependency that is not the target of any rule. It is also followed recursively for dependencies that come from implicit rules, in the search for a chain of rules.

Suffix rules are not mentioned in this algorithm because suffix rules are converted to equivalent pattern rules once the makefiles have been read in.

For an archive member target of the form `archive(member)', the following algorithm is run twice, first using the entire target name t, and second using `(member)' as the target t if the first run found no rule.

1. Split t into a directory part, called d, and the rest, called n. For example, if t is `src/foo.o', then d is `src/' and n is `foo.o'.
2. Make a list of all the pattern rules one of whose targets matches t or n. If the target pattern contains a slash, it is matched against t; otherwise, against n.
3. If any rule in that list is *not* a match-anything rule, then remove all nonterminal match-anything rules from the list.
4. Remove from the list all rules with no commands.
5. For each pattern rule in the list:
    1. Find the stem s, which is the nonempty part of t or n matched by the `%' in the target pattern.
    2. Compute the dependency names by substituting s for `%'; if the target pattern does not contain a slash, append d to the front of each dependency name.
    3. Test whether all the dependencies exist or ought to exist. (If a file name is mentioned in the makefile as a target or as an explicit dependency, then we say it ought to exist.)

       If all dependencies exist or ought to exist, or there are no dependencies, then this rule applies.
6. If no pattern rule has been found so far, try harder. For each pattern rule in the list:
    1. If the rule is terminal, ignore it and go on to the next rule.
    2. Compute the dependency names as before.
    3. Test whether all the dependencies exist or ought to exist.
    4. For each dependency that does not exist, follow this algorithm recursively to see if the dependency can be made by an implicit rule.

5. If all dependencies exist, ought to exist, or can be made by implicit rules, then this rule applies.

7. If no implicit rule applies, the rule for `.DEFAULT`, if any, applies. In that case, give t the same commands that `.DEFAULT` has. Otherwise, there are no commands for t.

Once a rule that applies has been found, for each target pattern of the rule other than the one that matched t or n, the `%' in the pattern is replaced with s and the resultant file name is stored until the commands to remake the target file t are executed. After these commands are executed, each of these stored file names are entered into the data base and marked as having been updated and having the same update status as the file t.

When the commands of a pattern rule are executed for t, the automatic variables are set corresponding to the target and dependencies. See section Automatic Variables.

Go to the previous, next section.

Go to the previous, next section.

Using make to Update Archive Files

Archive files are files containing named subfiles called members; they are maintained with the program `ar` and their main use is as subroutine libraries for linking.

Archive Members as Targets

An individual member of an archive file can be used as a target or dependency in `make`. You specify the member named member in archive file archive as follows:

```
archive(member)
```

This construct is available only in targets and dependencies, not in commands! Most programs that you might use in commands do not support this syntax and cannot act directly on archive members. Only `ar` and other programs specifically designed to operate on archives can do so. Therefore, valid commands to update an archive member target probably must use `ar`. For example, this rule says to create a member `` `hack.o' `` in archive `` `foolib' `` by copying the file `` `hack.o' ``:

```
foolib(hack.o) : hack.o
        ar cr foolib hack.o
```

In fact, nearly all archive member targets are updated in just this way and there is an implicit rule to do it for you. **Note:** The `c' flag to `ar` is required if the archive file does not already exist.

To specify several members in the same archive, you can write all the member names together between the parentheses. For example:

```
foolib(hack.o kludge.o)
```

is equivalent to:

```
foolib(hack.o) foolib(kludge.o)
```

You can also use shell-style wildcards in an archive member reference. See section Using Wildcard Characters in File Names. For example, `foolib(*.o)' expands to all existing members of the `` `foolib' `` archive whose names end in `.o'; perhaps `foolib(hack.o) foolib(kludge.o)'.

Implicit Rule for Archive Member Targets

Recall that a target that looks like `a(m)' stands for the member named m in the archive file a.

When make looks for an implicit rule for such a target, as a special feature it considers implicit rules that match `(m)', as well as those that match the actual target `a(m)'.

This causes one special rule whose target is `(%)' to match. This rule updates the target `a(m)' by copying the file m into the archive. For example, it will update the archive member target `foo.a(bar.o)' by copying the *file* `bar.o' into the archive `foo.a' as a *member* named `bar.o'.

When this rule is chained with others, the result is very powerful. Thus, `make "foo.a(bar.o)"' (the quotes are needed to protect the `(' and `)' from being interpreted specially by the shell) in the presence of a file `bar.c' is enough to cause the following commands to be run, even without a makefile:

```
cc -c bar.c -o bar.o
ar r foo.a bar.o
rm -f bar.o
```

Here make has envisioned the file `bar.o' as an intermediate file. See section Chains of Implicit Rules.

Implicit rules such as this one are written using the automatic variable `$%'. See section Automatic Variables.

An archive member name in an archive cannot contain a directory name, but it may be useful in a makefile to pretend that it does. If you write an archive member target `foo.a(dir/file.o)', make will perform automatic updating with this command:

```
ar r foo.a dir/file.o
```

which has the effect of copying the file `dir/file.o' into a member named `file.o'. In connection with such usage, the automatic variables %D and %F may be useful.

Updating Archive Symbol Directories

An archive file that is used as a library usually contains a special member named `__.SYMDEF' that contains a directory of the external symbol names defined by all the other members. After you update any other members, you need to update `__.SYMDEF' so that it will summarize the other members properly. This is done by running the ranlib program:

```
ranlib archivefile
```

Normally you would put this command in the rule for the archive file, and make all the members of the archive file dependencies of that rule. For example,


```
libfoo.a: libfoo.a(x.o) libfoo.a(y.o) ...
        ranlib libfoo.a
```

The effect of this is to update archive members `x.o', `y.o', etc., and then update the symbol directory member `__.SYMDEF' by running `ranlib`. The rules for updating the members are not shown here; most likely you can omit them and use the implicit rule which copies files into the archive, as described in the preceding section.

This is not necessary when using the GNU `ar` program, which updates the `__.SYMDEF' member automatically.

Dangers When Using Archives

It is important to be careful when using parallel execution (the `-j` switch; see section Parallel Execution) and archives. If multiple `ar` commands run at the same time on the same archive file, they will not know about each other and can corrupt the file.

Possibly a future version of `make` will provide a mechanism to circumvent this problem by serializing all commands that operate on the same archive file. But for the time being, you must either write your makefiles to avoid this problem in some other way, or not use `-j`.

Suffix Rules for Archive Files

You can write a special kind of suffix rule for dealing with archive files. See section Old-Fashioned Suffix Rules, for a full explanation of suffix rules. Archive suffix rules are obsolete in GNU `make`, because pattern rules for archives are a more general mechanism (see section Implicit Rule for Archive Member Targets). But they are retained for compatibility with other `make`s.

To write a suffix rule for archives, you simply write a suffix rule using the target suffix `.a' (the usual suffix for archive files). For example, here is the old-fashioned suffix rule to update a library archive from C source files:

```
.c.a:
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o
        $(AR) r $@ $*.o
        $(RM) $*.o
```

This works just as if you had written the pattern rule:

```
(%.o): %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o
        $(AR) r $@ $*.o
        $(RM) $*.o
```

In fact, this is just what `make` does when it sees a suffix rule with `.a' as the target suffix. Any double-suffix rule `.x.a' is converted to a pattern rule with the target pattern `(%.o)' and a dependency pattern of `%.x'.

Since you might want to use `.a' as the suffix for some other kind of file, `make` also converts archive suffix rules to pattern rules in the normal way (see section Old-Fashioned Suffix Rules). Thus a double-suffix rule `.x.a' produces two pattern rules: `(%.o): %.x' and `%.a: %.x'.

Go to the previous, next section.

Go to the previous, next section.

Features of GNU `make`

Here is a summary of the features of GNU `make`, for comparison with and credit to other versions of `make`. We consider the features of `make` in 4.2 BSD systems as a baseline. If you are concerned with writing portable makefiles, you should use only the features of `make` *not* listed here or in section Incompatibilities and Missing Features.

Many features come from the version of `make` in System V.

- The `VPATH` variable and its special meaning. See section Searching Directories for Dependencies. This feature exists in System V `make`, but is undocumented. It is documented in 4.3 BSD `make` (which says it mimics System V's `VPATH` feature).
- Included makefiles. See section Including Other Makefiles. Allowing multiple files to be included with a single directive is a GNU extension.
- Variables are read from and communicated via the environment. See section Variables from the Environment.
- Options passed through the variable `MAKEFLAGS` to recursive invocations of `make`. See section Communicating Options to a Sub-`make`.
- The automatic variable `$%` is set to the member name in an archive reference. See section Automatic Variables.
- The automatic variables `$@`, `$*`, `$<`, `$%`, and `$?` have corresponding forms like `$(@F)` and `$(@D)`. We have generalized this to `$^` as an obvious extension. See section Automatic Variables.
- Substitution variable references. See section Basics of Variable References.
- The command-line options `-b' and `-m', accepted and ignored. In System V `make`, these options actually do something.
- Execution of recursive commands to run `make` via the variable `MAKE` even if `-n', `-q' or `-t' is specified. See section Recursive Use of `make`.
- Support for suffix `.a' in suffix rules. See section Suffix Rules for Archive Files. This feature is obsolete in GNU `make`, because the general feature of rule chaining (see section Chains of Implicit Rules) allows one pattern rule for installing members in an archive (see section Implicit Rule for Archive Member Targets) to be sufficient.
- The arrangement of lines and backslash-newline combinations in commands is retained when the commands are printed, so they appear as they do in the makefile, except for the stripping of initial whitespace.

The following features were inspired by various other versions of `make`. In some cases it is unclear exactly which versions inspired which others.

- Pattern rules using `%'. This has been implemented in several versions of `make`. We're not sure who invented it first, but it's been spread around a bit. See section Defining and Redefining Pattern

Rules.

- Rule chaining and implicit intermediate files. This was implemented by Stu Feldman in his version of `make` for AT&T Eighth Edition Research Unix, and later by Andrew Hume of AT&T Bell Labs in his `mk` program (where he terms it "transitive closure"). We do not really know if we got this from either of them or thought it up ourselves at the same time. See section Chains of Implicit Rules.
- The automatic variable `$^` containing a list of all dependencies of the current target. We did not invent this, but we have no idea who did. See section Automatic Variables. The automatic variable `$+` is a simple extension of `$^`.
- The "what if" flag (`-W' in GNU `make`) was (as far as we know) invented by Andrew Hume in `mk`. See section Instead of Executing the Commands.
- The concept of doing several things at once (parallelism) exists in many incarnations of `make` and similar programs, though not in the System V or BSD implementations. See section Command Execution.
- Modified variable references using pattern substitution come from SunOS 4. See section Basics of Variable References. This functionality was provided in GNU `make` by the `patsubst` function before the alternate syntax was implemented for compatibility with SunOS 4. It is not altogether clear who inspired whom, since GNU `make` had `patsubst` before SunOS 4 was released.
- The special significance of `+' characters preceding command lines (see section Instead of Executing the Commands) is mandated by IEEE Standard 1003.2-1992 (POSIX.2).
- The `+=' syntax to append to the value of a variable comes from SunOS 4 `make`. See section Appending More Text to Variables.
- The syntax `archive(mem1 mem2...)' to list multiple members in a single archive file comes from SunOS 4 `make`. See section Archive Members as Targets.
- The `-include` directive to include makefiles with no error for a nonexistent file comes from SunOS 4 `make`. (But note that SunOS 4 `make` does not allow multiple makefiles to be specified in one `-include` directive.)

The remaining features are inventions new in GNU `make`:

- Use the `-v' or `--version' option to print version and copyright information.
- Use the `-h' or `--help' option to summarize the options to `make`.
- Simply-expanded variables. See section The Two Flavors of Variables.
- Pass command-line variable assignments automatically through the variable `MAKE` to recursive `make` invocations. See section Recursive Use of `make`.
- Use the `-C' or `--directory' command option to change directory. See section Summary of Options.
- Make verbatim variable definitions with `define`. See section Defining Variables Verbatim.
- Declare phony targets with the special target `.PHONY`.

  Andrew Hume of AT&T Bell Labs implemented a similar feature with a different syntax in his `mk`

program. This seems to be a case of parallel discovery. See section Phony Targets.

- Manipulate text by calling functions. See section Functions for Transforming Text.
- Use the `-o' or `--old-file' option to pretend a file's modification-time is old. See section Avoiding Recompilation of Some Files.
- Conditional execution.

  This feature has been implemented numerous times in various versions of `make`; it seems a natural extension derived from the features of the C preprocessor and similar macro languages and is not a revolutionary concept. See section Conditional Parts of Makefiles.
- Specify a search path for included makefiles. See section Including Other Makefiles.
- Specify extra makefiles to read with an environment variable. See section The Variable `MAKEFILES`.
- Strip leading sequences of `./' from file names, so that `` `./file' `` and `` `file' `` are considered to be the same file.
- Use a special search method for library dependencies written in the form `-lname'. See section Directory Search for Link Libraries.
- Allow suffixes for suffix rules (see section Old-Fashioned Suffix Rules) to contain any characters. In other versions of `make`, they must begin with `.' and not contain any `/' characters.
- Keep track of the current level of `make` recursion using the variable `MAKELEVEL`. See section Recursive Use of `make`.
- Specify static pattern rules. See section Static Pattern Rules.
- Provide selective `vpath` search. See section Searching Directories for Dependencies.
- Provide computed variable references. See section Basics of Variable References.
- Update makefiles. See section How Makefiles Are Remade. System V `make` has a very, very limited form of this functionality in that it will check out SCCS files for makefiles.
- Various new built-in implicit rules. See section Catalogue of Implicit Rules.
- The built-in variable `MAKE_VERSION' gives the version number of `make`.

Go to the previous, next section.

Go to the previous, next section.

Incompatibilities and Missing Features

The `make` programs in various other systems support a few features that are not implemented in GNU `make`. The POSIX.2 standard (IEEE Standard 1003.2-1992) which specifies `make` does not require any of these features.

- A target of the form `file((entry))' stands for a member of archive file file. The member is chosen, not by name, but by being an object file which defines the linker symbol entry.

  This feature was not put into GNU `make` because of the nonmodularity of putting knowledge into `make` of the internal format of archive file symbol tables. See section Updating Archive Symbol Directories.

- Suffixes (used in suffix rules) that end with the character `~' have a special meaning to System V `make`; they refer to the SCCS file that corresponds to the file one would get without the `~'. For example, the suffix rule `.c~.o' would make the file `n.o' from the SCCS file `s.n.c'. For complete coverage, a whole series of such suffix rules is required. See section Old-Fashioned Suffix Rules.

  In GNU `make`, this entire series of cases is handled by two pattern rules for extraction from SCCS, in combination with the general feature of rule chaining. See section Chains of Implicit Rules.

- In System V `make`, the string `$$@' has the strange meaning that, in the dependencies of a rule with multiple targets, it stands for the particular target that is being processed.

  This is not defined in GNU `make` because `$$' should always stand for an ordinary `$'.

  It is possible to get this functionality through the use of static pattern rules (see section Static Pattern Rules). The System V `make` rule:

  ```
  $(targets): $$@.o lib.a
  ```

  can be replaced with the GNU `make` static pattern rule:

  ```
  $(targets): %: %.o lib.a
  ```

- In System V and 4.3 BSD `make`, files found by `VPATH` search (see section Searching Directories for Dependencies) have their names changed inside command strings. We feel it is much cleaner to always use automatic variables and thus make this feature obsolete.

- In some Unix `make`s, the automatic variable `$*` appearing in the dependencies of a rule has the amazingly strange "feature" of expanding to the full name of the *target of that rule*. We cannot imagine what went on in the minds of Unix `make` developers to do this; it is utterly inconsistent with the normal definition of `$*`.

- In some Unix `make`s, implicit rule search (see section Using Implicit Rules) is apparently done for *all* targets, not just those without commands. This means you can do:


```
foo.o:
        cc -c foo.c
```

and Unix `make` will intuit that `` `foo.o' `` depends on `` `foo.c' ``.

We feel that such usage is broken. The dependency properties of `make` are well-defined (for GNU `make`, at least), and doing such a thing simply does not fit the model.

- GNU `make` does not include any built-in implicit rules for compiling or preprocessing EFL programs. If we hear of anyone who is using EFL, we will gladly add them.
- It appears that in SVR4 `make`, a suffix rule can be specified with no commands, and it is treated as if it had empty commands (see section Using Empty Commands). For example:

```
.c.a:
```

will override the built-in `` `.c.a' `` suffix rule.

We feel that it is cleaner for a rule without commands to always simply add to the dependency list for the target. The above example can be easily rewritten to get the desired behavior in GNU `make`:

```
.c.a: ;
```

- Some versions of `make` invoke the shell with the `-e' flag, except under `-k' (see section Testing the Compilation of a Program). The `-e' flag tells the shell to exit as soon as any program it runs returns a nonzero status. We feel it is cleaner to write each shell command line to stand on its own and not require this special treatment.

Go to the previous, next section.

Go to the previous, next section.

Makefile Conventions

This chapter describes conventions for writing the Makefiles for GNU programs.

General Conventions for Makefiles

Every Makefile should contain this line:

```
SHELL = /bin/sh
```

to avoid trouble on systems where the `SHELL` variable might be inherited from the environment. (This is never a problem with GNU `make`.)

Different `make` programs have incompatible suffix lists and implicit rules, and this sometimes creates confusion or misbehavior. So it is a good idea to set the suffix list explicitly using only the suffixes you need in the particular Makefile, like this:

```
.SUFFIXES:
.SUFFIXES: .c .o
```

The first line clears out the suffix list, the second introduces all suffixes which may be subject to implicit rules in this Makefile.

Don't assume that `.' is in the path for command execution. When you need to run programs that are a part of your package during the make, please make sure that it uses `./' if the program is built as part of the make or `$(srcdir)/' if the file is an unchanging part of the source code. Without one of these prefixes, the current search path is used.

The distinction between `./' and `$(srcdir)/' is important when using the `--srcdir' option to `configure'. A rule of the form:

```
foo.1 : foo.man sedscript
        sed -e sedscript foo.man > foo.1
```

will fail when the current directory is not the source directory, because `foo.man' and `sedscript' are not in the current directory.

When using GNU `make`, relying on `VPATH' to find the source file will work in the case where there is a single dependency file, since the `make' automatic variable `$<' will represent the source file wherever it is. (Many versions of `make` set `$<' only in implicit rules.) A makefile target like

```
foo.o : bar.c
        $(CC) -I. -I$(srcdir) $(CFLAGS) -c bar.c -o foo.o
```

should instead be written as

```
foo.o : bar.c
        $(CC) -I. -I$(srcdir) $(CFLAGS) -c $< -o $@
```

in order to allow `VPATH' to work correctly. When the target has multiple dependencies, using an explicit `$(srcdir)' is the easiest way to make the rule work well. For example, the target above for `foo.1' is best written as:

```
foo.1 : foo.man sedscript
        sed -e $(srcdir)/sedscript $(srcdir)/foo.man > $@
```

Utilities in Makefiles

Write the Makefile commands (and any shell scripts, such as `configure`) to run in `sh`, not in `csh`. Don't use any special features of `ksh` or `bash`.

The `configure` script and the Makefile rules for building and installation should not use any utilities directly except these:

```
cat cmp cp echo egrep expr grep
ln mkdir mv pwd rm rmdir sed test touch
```

Stick to the generally supported options for these programs. For example, don't use `mkdir -p', convenient as it may be, because most systems don't support it.

The Makefile rules for building and installation can also use compilers and related programs, but should do so via `make` variables so that the user can substitute alternatives. Here are some of the programs we mean:

```
ar bison cc flex install ld lex
make makeinfo ranlib texi2dvi yacc
```

Use the following `make` variables:

```
$(AR) $(BISON) $(CC) $(FLEX) $(INSTALL) $(LD) $(LEX)
$(MAKE) $(MAKEINFO) $(RANLIB) $(TEXI2DVI) $(YACC)
```

When you use `ranlib`, you should make sure nothing bad happens if the system does not have `ranlib`. Arrange to ignore an error from that command, and print a message before the command to tell the user that failure of the `ranlib` command does not mean a problem.

If you use symbolic links, you should implement a fallback for systems that don't have symbolic links.

It is ok to use other utilities in Makefile portions (or scripts) intended only for particular systems where you know those utilities to exist.

Standard Targets for Users

All GNU programs should have the following targets in their Makefiles:

`all'

Compile the entire program. This should be the default target. This target need not rebuild any documentation files; Info files should normally be included in the distribution, and DVI files should be made only when explicitly asked for.

`install'

Compile the program and copy the executables, libraries, and so on to the file names where they should reside for actual use. If there is a simple test to verify that a program is properly installed, this target should run that test.

If possible, write the `install` target rule so that it does not modify anything in the directory where the program was built, provided `make all' has just been done. This is convenient for building the program under one user name and installing it under another.

The commands should create all the directories in which files are to be installed, if they don't already exist. This includes the directories specified as the values of the variables `prefix` and `exec_prefix`, as well as all subdirectories that are needed. One way to do this is by means of an `installdirs` target as described below.

Use `-' before any command for installing a man page, so that `make` will ignore any errors. This is in case there are systems that don't have the Unix man page documentation system installed.

The way to install Info files is to copy them into `$(infodir)' with `$(INSTALL_DATA)` (see section Variables for Specifying Commands), and then run the `install-info` program if it is present. `install-info` is a script that edits the Info `dir' file to add or update the menu entry for the given Info file; it will be part of the Texinfo package. Here is a sample rule to install an Info file:

```
$(infodir)/foo.info: foo.info
# There may be a newer info file in . than in srcdir.
        -if test -f foo.info; then d=.; \
         else d=$(srcdir); fi; \
        $(INSTALL_DATA) $$d/foo.info $@; \
# Run install-info only if it exists.
# Use `if' instead of just prepending `-' to the
# line so we notice real errors from install-info.
# We use `$(SHELL) -c' because some shells do not
# fail gracefully when there is an unknown command.
        if $(SHELL) -c 'install-info --version' \
           >/dev/null 2>&1; then \
          install-info --infodir=$(infodir) $$d/foo.info; \
        else true; fi
```

`uninstall'

Delete all the installed files that the `install' target would create (but not the noninstalled files su as `make all' would create).

This rule should not modify the directories where compilation is done, only the directories where files are installed.

`clean'

Delete all files from the current directory that are normally created by building the program. Don't delete the files that record the configuration. Also preserve files that could be made by building, but normally aren't because the distribution comes with them.

Delete `.dvi' files here if they are not part of the distribution.

`distclean'

Delete all files from the current directory that are created by configuring or building the program. If you have unpacked the source and built the program without creating any other files, `make distclean' should leave only the files that were in the distribution.

`mostlyclean'

Like `clean', but may refrain from deleting a few files that people normally don't want to recompile. For example, the `mostlyclean' target for GCC does not delete `libgcc.a', because recompiling it is rarely necessary and takes a lot of time.

`maintainer-clean'

Delete almost everything from the current directory that can be reconstructed with this Makefile. This typically includes everything deleted by `distclean`, plus more: C source files produced by Bison, tags tables, Info files, and so on.

The reason we say "almost everything" is that `make maintainer-clean' should not delete `configure' even if `configure' can be remade using a rule in the Makefile. More generally, `make maintainer-clean' should not delete anything that needs to exist in order to run `configure' and then begin to build the program. This is the only exception; `maintainer-clean` should delete everything else that can be rebuilt.

The `maintainer-clean' is intended to be used by a maintainer of the package, not by ordinary users. You may need special tools to reconstruct some of the files that `make maintainer-clean' deletes. Since these files are normally included in the distribution, we don't take care to make them easy to reconstruct. If you find you need to unpack the full distribution again, don't blame us.

To help make users aware of this, the commands for `maintainer-clean` should start with these two:

```
@echo "This command is intended for maintainers to use;"
@echo "it deletes files that may require special tools to rebuild."
```

`TAGS'

Update a tags table for this program.

`info'

Generate any Info files needed. The best way to write the rules is as follows:


```
info: foo.info

foo.info: foo.texi chap1.texi chap2.texi
        $(MAKEINFO) $(srcdir)/foo.texi
```

You must define the variable `MAKEINFO` in the Makefile. It should run the `makeinfo` program, which is part of the Texinfo distribution.

`dvi'

Generate DVI files for all TeXinfo documentation. For example:

```
dvi: foo.dvi

foo.dvi: foo.texi chap1.texi chap2.texi
        $(TEXI2DVI) $(srcdir)/foo.texi
```

You must define the variable `TEXI2DVI` in the Makefile. It should run the program `texi2dvi`, which is part of the Texinfo distribution. Alternatively, write just the dependencies, and allow GNU Make to provide the command.

`dist'

Create a distribution tar file for this program. The tar file should be set up so that the file names in the tar file start with a subdirectory name which is the name of the package it is a distribution for. This name can include the version number.

For example, the distribution tar file of GCC version 1.40 unpacks into a subdirectory named `` `gcc-1.40' ``.

The easiest way to do this is to create a subdirectory appropriately named, use `ln` or `cp` to install the proper files in it, and then `tar` that subdirectory.

The `dist` target should explicitly depend on all non-source files that are in the distribution, to make sure they are up to date in the distribution. See section 'Making Releases' in GNU Coding Standards.

`check'

Perform self-tests (if any). The user must build the program before running the tests, but need not install the program; you should write the self-tests so that they work when the program is built but not installed.

The following targets are suggested as conventional names, for programs in which they are useful.

`installcheck`

Perform installation tests (if any). The user must build and install the program before running the tests. You should not assume that `` `$(bindir)' `` is in the search path.

`installdirs`

It's useful to add a target named `installdirs' to create the directories where files are installed, and their parent directories. There is a script called `` `mkinstalldirs' `` which is convenient for this;

find it in the Texinfo package. You can use a rule like this:

```
# Make sure all installation directories (e.g. $(bindir))
# actually exist by making them if necessary.
installdirs: mkinstalldirs
        $(srcdir)/mkinstalldirs $(bindir) $(datadir) \
                                $(libdir) $(infodir) \
                                $(mandir)
```

This rule should not modify the directories where compilation is done. It should do nothing but create installation directories.

Variables for Specifying Commands

Makefiles should provide variables for overriding certain commands, options, and so on.

In particular, you should run most utility programs via variables. Thus, if you use Bison, have a variable named `BISON` whose default value is set with `BISON = bison', and refer to it with `$(BISON)` whenever you need to use Bison.

File management utilities such as `ln`, `rm`, `mv`, and so on, need not be referred to through variables in this way, since users don't need to replace them with other programs.

Each program-name variable should come with an options variable that is used to supply options to the program. Append `FLAGS' to the program-name variable name to get the options variable name--for example, `BISONFLAGS`. (The name `CFLAGS` is an exception to this rule, but we keep it because it is standard.) Use `CPPFLAGS` in any compilation command that runs the preprocessor, and use `LDFLAGS` in any compilation command that does linking as well as in any direct use of `ld`.

If there are C compiler options that *must* be used for proper compilation of certain files, do not include them in `CFLAGS`. Users expect to be able to specify `CFLAGS` freely themselves. Instead, arrange to pass the necessary options to the C compiler independently of `CFLAGS`, by writing them explicitly in the compilation commands or by defining an implicit rule, like this:

```
CFLAGS = -g
ALL_CFLAGS = -I. $(CFLAGS)
.c.o:
        $(CC) -c $(CPPFLAGS) $(ALL_CFLAGS) $<
```

Do include the `-g' option in `CFLAGS`, because that is not *required* for proper compilation. You can consider it a default that is only recommended. If the package is set up so that it is compiled with GCC by default, then you might as well include `-O' in the default value of `CFLAGS` as well.

Put `CFLAGS` last in the compilation command, after other variables containing compiler options, so the user can use `CFLAGS` to override the others.

Every Makefile should define the variable `INSTALL`, which is the basic command for installing a file into the system.

Every Makefile should also define the variables `INSTALL_PROGRAM` and `INSTALL_DATA`. (The default for each of these should be `$(INSTALL)`.) Then it should use those variables as the commands for actual installation, for executables and nonexecutables respectively. Use these variables as follows:

```
$(INSTALL_PROGRAM) foo $(bindir)/foo
$(INSTALL_DATA) libfoo.a $(libdir)/libfoo.a
```

Always use a file name, not a directory name, as the second argument of the installation commands. Use a separate command for each file to be installed.

Variables for Installation Directories

Installation directories should always be named by variables, so it is easy to install in a nonstandard place. The standard names for these variables are described below. They are based on a standard filesystem layout; variants of it are used in SVR4, 4.4BSD, Linux, Ultrix v4, and other modern operating systems.

These two variables set the root for the installation. All the other installation directories should be subdirectories of one of these two, and nothing should be directly installed into these two directories.

`prefix'

A prefix used in constructing the default values of the variables listed below. The default value of `prefix` should be `` `/usr/local' `` When building the complete GNU system, the prefix will be empty and `` `/usr' `` will be a symbolic link to `` `/' ``.

`exec_prefix'

A prefix used in constructing the default values of some of the variables listed below. The default value of `exec_prefix` should be `$(prefix)`.

Generally, `$(exec_prefix)` is used for directories that contain machine-specific files (such as executables and subroutine libraries), while `$(prefix)` is used directly for other directories.

Executable programs are installed in one of the following directories.

`bindir'

The directory for installing executable programs that users can run. This should normally be `` `/usr/local/bin' ``, but write it as `` `$(exec_prefix)/bin' ``.

`sbindir'

The directory for installing executable programs that can be run from the shell, but are only generally useful to system administrators. This should normally be `` `/usr/local/sbin' ``, but write it as `` `$(exec_prefix)/sbin' ``.

`libexecdir'

The directory for installing executable programs to be run by other programs rather than by users. This directory should normally be `` `/usr/local/libexec' ``, but write it as `` `$(exec_prefix)/libexec' ``.

Data files used by the program during its execution are divided into categories in two ways.

- Some files are normally modified by programs; others are never normally modified (though users

may edit some of these).

- Some files are architecture-independent and can be shared by all machines at a site; some are architecture-dependent and can be shared only by machines of the same kind and operating system; others may never be shared between two machines.

This makes for six different possibilities. However, we want to discourage the use of architecture-dependent files, aside from of object files and libraries. It is much cleaner to make other data files architecture-independent, and it is generally not hard.

Therefore, here are the variables makefiles should use to specify directories:

`datadir'

The directory for installing read-only architecture independent data files. This should normally be `/usr/local/share', but write it as `$(prefix)/share'. As a special exception, see `$(infodir)' and `$(includedir)' below.

`sysconfdir'

The directory for installing read-only data files that pertain to a single machine--that is to say, files for configuring a host. Mailer and network configuration files, `/etc/passwd', and so forth belong here. All the files in this directory should be ordinary ASCII text files. This directory should normally be `/usr/local/etc', but write it as `$(prefix)/etc'.

Do not install executables in this directory (they probably belong in `$(libexecdir)' or `$(sbindir))'. Also do not install files that are modified in the normal course of their use (programs whose purpose is to change the configuration of the system excluded). Those probably belong in `$(localstatedir)'.

`sharedstatedir'

The directory for installing architecture-independent data files which the programs modify while they run. This should normally be `/usr/local/com', but write it as `$(prefix)/com'.

`localstatedir'

The directory for installing data files which the programs modify while they run, and that pertain to one specific machine. Users should never need to modify files in this directory to configure the package's operation; put such configuration information in separate files that go in `datadir' or `$(sysconfdir)'. `$(localstatedir)' should normally be `/usr/local/var', but write it as `$(prefix)/var'.

`libdir'

The directory for object files and libraries of object code. Do not install executables here, they probably belong in `$(libexecdir)' instead. The value of `libdir' should normally be `/usr/local/lib', but write it as `$(exec_prefix)/lib'.

`infodir'

The directory for installing the Info files for this package. By default, it should be `/usr/local/info', but it should be written as `$(prefix)/info'.

`includedir'

The directory for installing header files to be included by user programs with the C `#include' preprocessor directive. This should normally be `/usr/local/include', but write it as

`$(prefix)/include'.

Most compilers other than GCC do not look for header files in `/usr/local/include'. So installing the header files this way is only useful with GCC. Sometimes this is not a problem because some libraries are only really intended to work with GCC. But some libraries are intended to work with other compilers. They should install their header files in two places, one specified by `includedir` and one specified by `oldincludedir`.

`oldincludedir'

The directory for installing `#include' header files for use with compilers other than GCC. This should normally be `/usr/include'.

The Makefile commands should check whether the value of `oldincludedir` is empty. If it is, they should not try to use it; they should cancel the second installation of the header files.

A package should not replace an existing header in this directory unless the header came from the same package. Thus, if your Foo package provides a header file `foo.h', then it should install the header file in the `oldincludedir` directory if either (1) there is no `foo.h' there or (2) the `foo.h' that exists came from the Foo package.

To tell whether `foo.h' came from the Foo package, put a magic string in the file--part of a comment--and grep for that string.

Unix-style man pages are installed in one of the following:

`mandir'

The directory for installing the man pages (if any) for this package. It should include the suffix for the proper section of the manual--usually `1' for a utility. It will normally be `/usr/local/man/man1', but you should write it as `$(prefix)/man/man1'.

`man1dir'

The directory for installing section 1 man pages.

`man2dir'

The directory for installing section 2 man pages.

`...'

Use these names instead of `mandir' if the package needs to install man pages in more than one section of the manual.

Don't make the primary documentation for any GNU software be a man page. Write a manual in Texinfo instead. Man pages are just for the sake of people running GNU software on Unix, which is a secondary application only.

`manext'

The file name extension for the installed man page. This should contain a period followed by the appropriate digit; it should normally be `.1'.

`man1ext'

The file name extension for installed section 1 man pages.

`man2ext'

The file name extension for installed section 2 man pages.

`...'

Use these names instead of `manext' if the package needs to install man pages in more than one section of the manual.

And finally, you should set the following variable:

`srcdir'

The directory for the sources being compiled. The value of this variable is normally inserted by the `configure` shell script.

For example:

```
# Common prefix for installation directories.
# NOTE: This directory must exist when you start the install.
prefix = /usr/local
exec_prefix = $(prefix)
# Where to put the executable for the command `gcc'.
bindir = $(exec_prefix)/bin
# Where to put the directories used by the compiler.
libexecdir = $(exec_prefix)/libexec
# Where to put the Info files.
infodir = $(prefix)/info
```

If your program installs a large number of files into one of the standard user-specified directories, it might be useful to group them into a subdirectory particular to that program. If you do this, you should write the `install` rule to create these subdirectories.

Do not expect the user to include the subdirectory name in the value of any of the variables listed above. The idea of having a uniform set of variable names for installation directories is to enable the user to specify the exact same values for several different GNU packages. In order for this to be useful, all the packages must be designed so that they will work sensibly when the user does so.

Go to the previous, next section.

Go to the previous, next section.

Quick Reference

This appendix summarizes the directives, text manipulation functions, and special variables which GNU `make` understands. See section Special Built-in Target Names, section Catalogue of Implicit Rules, and section Summary of Options, for other summaries.

Here is a summary of the directives GNU `make` recognizes:

`define variable`
`endef`

Define a multi-line, recursively-expanded variable.
See section Defining Canned Command Sequences.

`ifdef variable`
`ifndef variable`
`ifeq (a,b)`
`ifeq "a" "b"`
`ifeq 'a' 'b'`
`ifneq (a,b)`
`ifneq "a" "b"`
`ifneq 'a' 'b'`
`else`
`endif`

Conditionally evaluate part of the makefile.
See section Conditional Parts of Makefiles.

`include file`

Include another makefile.
See section Including Other Makefiles.

`override variable = value`
`override variable := value`
`override variable += value`
`override define variable`
`endef`

Define a variable, overriding any previous definition, even one from the command line.
See section The `override` Directive.

`export`

Tell `make` to export all variables to child processes by default.
See section Communicating Variables to a Sub-`make`.

`export variable`

`export variable = value`

`export variable := value`

`export variable += value`

`unexport variable`

Tell `make` whether or not to export a particular variable to child processes.
See section Communicating Variables to a Sub-`make`.

`vpath pattern path`

Specify a search path for files matching a `%' pattern.
See section The `vpath` Directive.

`vpath pattern`

Remove all search paths previously specified for pattern.

`vpath`

Remove all search paths previously specified in any `vpath` directive.

Here is a summary of the text manipulation functions (see section Functions for Transforming Text):

`$(subst from,to,text)`

Replace from with to in text.
See section Functions for String Substitution and Analysis.

`$(patsubst pattern,replacement,text)`

Replace words matching pattern with replacement in text.
See section Functions for String Substitution and Analysis.

`$(strip string)`

Remove excess whitespace characters from string.
See section Functions for String Substitution and Analysis.

`$(findstring find,text)`

Locate find in text.
See section Functions for String Substitution and Analysis.

`$(filter pattern...,text)`

Select words in text that match one of the pattern words.
See section Functions for String Substitution and Analysis.

`$(filter-out pattern...,text)`

Select words in text that *do not* match any of the pattern words.
See section Functions for String Substitution and Analysis.

`$(sort list)`

Sort the words in list lexicographically, removing duplicates.

See section Functions for String Substitution and Analysis.

`$(dir names...)`

Extract the directory part of each file name.
See section Functions for File Names.

`$(notdir names...)`

Extract the non-directory part of each file name.
See section Functions for File Names.

`$(suffix names...)`

Extract the suffix (the last `.' and following characters) of each file name.
See section Functions for File Names.

`$(basename names...)`

Extract the base name (name without suffix) of each file name.
See section Functions for File Names.

`$(addsuffix suffix,names...)`

Append suffix to each word in names.
See section Functions for File Names.

`$(addprefix prefix,names...)`

Prepend prefix to each word in names.
See section Functions for File Names.

`$(join list1,list2)`

Join two parallel lists of words.
See section Functions for File Names.

`$(word n,text)`

Extract the nth word (one-origin) of text.
See section Functions for File Names.

`$(words text)`

Count the number of words in text.
See section Functions for File Names.

`$(firstword names...)`

Extract the first word of names.
See section Functions for File Names.

`$(wildcard pattern...)`

Find file names matching a shell file name pattern (*not* a `%' pattern).
See section The Function `wildcard`.

`$(shell command)`

Execute a shell command and return its output.
See section The `shell` Function.

`$(origin variable)`

Return a string describing how the `make` variable variable was defined.
See section The `origin` Function.

`$(foreach var,words,text)`

Evaluate text with var bound to each word in words, and concatenate the results.
See section The `foreach` Function.

Here is a summary of the automatic variables. See section Automatic Variables, for full information.

`$@`

The file name of the target.

`$%`

The target member name, when the target is an archive member.

`$<`

The name of the first dependency.

`$?`

The names of all the dependencies that are newer than the target, with spaces between them. For dependencies which are archive members, only the member named is used (see section Using `make` to Update Archive Files).

`$^`

`$+`

The names of all the dependencies, with spaces between them. For dependencies which are archive members, only the member named is used (see section Using `make` to Update Archive Files). The value of `$^` omits duplicate dependencies, while `$+` retains them and preserves their order.

`$*`

The stem with which an implicit rule matches (see section How Patterns Match).

`$(@D)`

`$(@F)`

The directory part and the file-within-directory part of `$@`.

`$(*D)`

`$(*F)`

The directory part and the file-within-directory part of `$*`.

`$(%D)`

`$(%F)`

The directory part and the file-within-directory part of `$%`.

`$(<D)`

`$(<F)`

The directory part and the file-within-directory part of `$<`.

`$(^D)`
`$(^F)`

The directory part and the file-within-directory part of `$^`.

`$(+D)`
`$(+F)`

The directory part and the file-within-directory part of `$+`.

`$(?D)`
`$(?F)`

The directory part and the file-within-directory part of `$?`.

These variables are used specially by GNU `make`:

`MAKEFILES`

Makefiles to be read on every invocation of `make`.
See section The Variable `MAKEFILES`.

`VPATH`

Directory search path for files not found in the current directory.
See section `VPATH`: Search Path for All Dependencies.

`SHELL`

The name of the system default command interpreter, usually `` `/bin/sh' ``. You can set `SHELL` in the makefile to change the shell used to run commands. See section Command Execution.

`MAKE`

The name with which `make` was invoked. Using this variable in commands has special meaning.
See section How the `MAKE` Variable Works.

`MAKELEVEL`

The number of levels of recursion (sub-`make`s).
See section Communicating Variables to a Sub-`make`.

`MAKEFLAGS`

The flags given to `make`. You can set this in the environment or a makefile to set flags.
See section Communicating Options to a Sub-`make`.

`SUFFIXES`

The default list of suffixes before `make` reads any makefiles.

Go to the previous, next section.

Go to the previous, next section.

Complex Makefile Example

Here is the makefile for the GNU `tar` program. This is a moderately complex makefile.

Because it is the first target, the default goal is `all'. An interesting feature of this makefile is that `` `testpad.h' `` is a source file automatically created by the `testpad` program, itself compiled from `` `testpad.c' ``.

If you type `make' or `make all', then `make` creates the `` `tar' `` executable, the `` `rmt' `` daemon that provides remote tape access, and the `` `tar.info' `` Info file.

If you type `make install', then `make` not only creates `` `tar' ``, `` `rmt' ``, and `` `tar.info' ``, but also installs them.

If you type `make clean', then `make` removes the `.o' files, and the `` `tar' ``, `` `rmt' ``, `` `testpad' ``, `` `testpad.h' ``, and `` `core' `` files.

If you type `make distclean', then `make` not only removes the same files as does `make clean' but also the `` `TAGS' ``, `` `Makefile' ``, and `` `config.status' `` files. (Although it is not evident, this makefile (and `` `config.status' ``) is generated by the user with the `configure` program, which is provided in the `tar` distribution, but is not shown here.)

If you type `make realclean', then `make` removes the same files as does `make distclean' and also removes the Info files generated from `` `tar.texinfo' ``.

In addition, there are targets `shar` and `dist` that create distribution kits.

```
# Generated automatically from Makefile.in by configure.
# Un*x Makefile for GNU tar program.
# Copyright (C) 1991 Free Software Foundation, Inc.

# This program is free software; you can redistribute
# it and/or modify it under the terms of the GNU
# General Public License ...
...
...

SHELL = /bin/sh

#### Start of system configuration section. ####

srcdir = .

# If you use gcc, you should either run the
```


```
# fixincludes script that comes with it or else use
# gcc with the -traditional option.  Otherwise ioctl
# calls will be compiled incorrectly on some systems.
CC = gcc -O
YACC = bison -y
INSTALL = /usr/local/bin/install -c
INSTALLDATA = /usr/local/bin/install -c -m 644

# Things you might add to DEFS:
# -DSTDC_HEADERS        If you have ANSI C headers and
#                       libraries.
# -DPOSIX               If you have POSIX.1 headers and
#                       libraries.
# -DBSD42               If you have sys/dir.h (unless
#                       you use -DPOSIX), sys/file.h,
#                       and st_blocks in `struct stat'.
# -DUSG                 If you have System V/ANSI C
#                       string and memory functions
#                       and headers, sys/sysmacros.h,
#                       fcntl.h, getcwd, no valloc,
#                       and ndir.h (unless
#                       you use -DDIRENT).
# -DNO_MEMORY_H         If USG or STDC_HEADERS but do not
#                       include memory.h.
# -DDIRENT              If USG and you have dirent.h
#                       instead of ndir.h.
# -DSIGTYPE=int         If your signal handlers
#                       return int, not void.
# -DNO_MTIO             If you lack sys/mtio.h
#                       (magtape ioctls).
# -DNO_REMOTE           If you do not have a remote shell
#                       or rexec.
# -DUSE_REXEC           To use rexec for remote tape
#                       operations instead of
#                       forking rsh or remsh.
# -DVPRINTF_MISSING     If you lack vprintf function
#                       (but have _doprnt).
# -DDOPRNT_MISSING      If you lack _doprnt function.
#                       Also need to define
#                       -DVPRINTF_MISSING.
# -DFTIME_MISSING       If you lack ftime system call.
# -DSTRSTR_MISSING      If you lack strstr function.
# -DVALLOC_MISSING      If you lack valloc function.
# -DMKDIR_MISSING       If you lack mkdir and
#                       rmdir system calls.
# -DRENAME_MISSING      If you lack rename system call.
```


```
# -DFTRUNCATE_MISSING   If you lack ftruncate
#                       system call.
# -DV7                  On Version 7 Unix (not
#                       tested in a long time).
# -DEMUL_OPEN3          If you lack a 3-argument version
#                       of open, and want to emulate it
#                       with system calls you do have.
# -DNO_OPEN3            If you lack the 3-argument open
#                       and want to disable the tar -k
#                       option instead of emulating open.
# -DXENIX               If you have sys/inode.h
#                       and need it 94 to be included.

DEFS =  -DSIGTYPE=int -DDIRENT -DSTRSTR_MISSING \
        -DVPRINTF_MISSING -DBSD42
# Set this to rtapelib.o unless you defined NO_REMOTE,
# in which case make it empty.
RTAPELIB = rtapelib.o
LIBS =
DEF_AR_FILE = /dev/rmt8
DEFBLOCKING = 20

CDEBUG = -g
CFLAGS = $(CDEBUG) -I. -I$(srcdir) $(DEFS) \
        -DDEF_AR_FILE=\"$(DEF_AR_FILE)\" \
        -DDEFBLOCKING=$(DEFBLOCKING)
LDFLAGS = -g

prefix = /usr/local
# Prefix for each installed program,
# normally empty or `g'.
binprefix =

# The directory to install tar in.
bindir = $(prefix)/bin

# The directory to install the info files in.
infodir = $(prefix)/info

#### End of system configuration section. ####

SRC1 =  tar.c create.c extract.c buffer.c \
        getoldopt.c update.c gnu.c mangle.c
SRC2 =  version.c list.c names.c diffarch.c \
        port.c wildmat.c getopt.c
SRC3 =  getopt1.c regex.c getdate.y
```

TOC

```
SRCS =  $(SRC1) $(SRC2) $(SRC3)
OBJ1 =  tar.o create.o extract.o buffer.o \
        getoldopt.o update.o gnu.o mangle.o
OBJ2 =  version.o list.o names.o diffarch.o \
        port.o wildmat.o getopt.o
OBJ3 =  getopt1.o regex.o getdate.o $(RTAPELIB)
OBJS =  $(OBJ1) $(OBJ2) $(OBJ3)
AUX =   README COPYING ChangeLog Makefile.in  \
        makefile.pc configure configure.in \
        tar.texinfo tar.info* texinfo.tex \
        tar.h port.h open3.h getopt.h regex.h \
        rmt.h rmt.c rtapelib.c alloca.c \
        msd_dir.h msd_dir.c tcexparg.c \
        level-0 level-1 backup-specs testpad.c

all:    tar rmt tar.info

tar:    $(OBJS)
        $(CC) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)

rmt:    rmt.c
        $(CC) $(CFLAGS) $(LDFLAGS) -o $@ rmt.c

tar.info: tar.texinfo
        makeinfo tar.texinfo

install: all
        $(INSTALL) tar $(bindir)/$(binprefix)tar
        -test ! -f rmt || $(INSTALL) rmt /etc/rmt
        $(INSTALLDATA) $(srcdir)/tar.info* $(infodir)

$(OBJS): tar.h port.h testpad.h
regex.o buffer.o tar.o: regex.h
# getdate.y has 8 shift/reduce conflicts.

testpad.h: testpad
        ./testpad

testpad: testpad.o
        $(CC) -o $@ testpad.o

TAGS:   $(SRCS)
        etags $(SRCS)

clean:
        rm -f *.o tar rmt testpad testpad.h core
```


```
distclean: clean
        rm -f TAGS Makefile config.status

realclean: distclean
        rm -f tar.info*

shar: $(SRCS) $(AUX)
        shar $(SRCS) $(AUX) | compress \
          > tar-`sed -e '/version_string/!d' \
                     -e 's/[^0-9.]*\([0-9.]*\).*/\1/' \
                     -e q
                     version.c`.shar.Z

dist: $(SRCS) $(AUX)
        echo tar-`sed \
             -e '/version_string/!d' \
             -e 's/[^0-9.]*\([0-9.]*\).*/\1/' \
             -e q
             version.c` > .fname
        -rm -rf `cat .fname`
        mkdir `cat .fname`
        ln $(SRCS) $(AUX) `cat .fname`
        -rm -rf `cat .fname` .fname
        tar chZf `cat .fname`.tar.Z `cat .fname`

tar.zoo: $(SRCS) $(AUX)
        -rm -rf tmp.dir
        -mkdir tmp.dir
        -rm tar.zoo
        for X in $(SRCS) $(AUX) ; do \
            echo $$X ; \
            sed 's/$$/^M/' $$X \
            > tmp.dir/$$X ; done
        cd tmp.dir ; zoo aM ../tar.zoo *
        -rm -rf tmp.dir
```

Go to the previous, next section.

Go to the previous, next section.

Index of Concepts

TOC

?

@

[

\

_

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

TOC

r

S

t

u

v

W

y

~

Go to the previous, next section.

Go to the previous section.

Index of Functions, Variables, & Directives

$

%

*

+

.

/

<

?

@

^

a

b

c

d

e

f

g

i

j

l

m

n

o

p

r

s

t

u

v

w

y

Go to the previous section.

Gnus 5.3 Manual

Go to the next section.

Starting Gnus

If your system administrator has set things up properly, starting Gnus and reading news is extremely easy--you just type M-x gnus in your Emacs.

If you want to start Gnus in a different frame, you can use the command M-x gnus-other-frame instead.

If things do not go smoothly at startup, you have to twiddle some variables.

Finding the News

The `gnus-select-method` variable says where Gnus should look for news. This variable should be a list where the first element says how and the second element says where. This method is your native method. All groups that are not fetched with this method are foreign groups.

For instance, if the `news.somewhere.edu' NNTP server is where you want to get your daily dosage of news from, you'd say:

```
(setq gnus-select-method '(nntp "news.somewhere.edu"))
```

If you want to read directly from the local spool, say:

```
(setq gnus-select-method '(nnspool ""))
```

If you can use a local spool, you probably should, as it will almost certainly be much faster.

If this variable is not set, Gnus will take a look at the `NNTPSERVER` environment variable. If that variable isn't set, Gnus will see whether `gnus-nntpserver-file` (`/etc/nntpserver' by default) has any opinions on the matter. If that fails as well, Gnus will will try to use the machine that is running Emacs as an NNTP server. That's a long-shot, though.

If `gnus-nntp-server` is set, this variable will override `gnus-select-method`. You should therefore set `gnus-nntp-server` to `nil`, which is what it is by default.

You can also make Gnus prompt you interactively for the name of an NNTP server. If you give a non-numerical prefix to `gnus` (i.e., C-u M-x gnus), Gnus will let you choose between the servers in the `gnus-secondary-servers` list (if any). You can also just type in the name of any server you feel like visiting.

However, if you use one NNTP server regularly and are just interested in a couple of groups from a different server, you would be better served by using the B command in the group buffer. It will let you have a look at what groups are available, and you can subscribe to any of the groups you want to. This also makes `.newsrc' maintenance much tidier. See section Foreign Groups.

A slightly different approach to foreign groups is to set the `gnus-secondary-select-methoc` variable. The select methods listed in this variable are in many ways just as native as the `gnus-select-method` server. They will also be queried for active files during startup (if that's required), and new newsgroups that appear on these servers will be subscribed (or not) just as native groups are.

For instance, if you use the `nnmbox` backend to read your mail, you would typically set this variable to

```
(setq gnus-secondary-select-methods '((nnmbox "")))
```

The First Time

If no startup files exist, Gnus will try to determine what groups should be subscribed by default.

If the variable `gnus-default-subscribed-newsgroups` is set, Gnus will subscribe you to just those groups in that list, leaving the rest killed. Your system administrator should have set this variable to something useful.

Since she hasn't, Gnus will just subscribe you to a few arbitrarily picked groups (i.e., `*.newusers'). (Arbitrary is here defined as whatever Lars thinks you should read.)

You'll also be subscribed to the Gnus documentation group, which should help you with most common problems.

If `gnus-default-subscribed-newsgroups` is `t`, Gnus will just use the normal functions for handling new groups, and not do anything special.

The Server is Down

If the default server is down, Gnus will understandably have some problems starting. However, if you have some mail groups in addition to the news groups, you may want to start Gnus anyway.

Gnus, being the trusting sort of program, will ask whether to proceed without a native select method if that server can't be contacted. This will happen whether the server doesn't actually exist (i.e., you have given the wrong address) or the server has just momentarily taken ill for some reason or other. If you decide to continue and have no foreign groups, you'll find it difficult to actually do anything in the group buffer. But, hey, that's your problem. Blllrph!

If you know that the server is definitely down, or you just want to read your mail without bothering with the server at all, you can use the `gnus-no-server` command to start Gnus. That might come in handy if you're in a hurry as well.

Slave Gnusiï

You might want to run more than one Emacs with more than one Gnus at the same time. If you are using different `.newsrc' files (eg., if you are using the two different Gnusiï to read from two different servers), that is no problem whatsoever. You just do it.

The problem appears when you want to run two Gnusiï that use the same `.newsrc` file.

To work around that problem some, we here at the Think-Tank at the Gnus Towers have come up with a new concept: Masters and servants. (We have applied for a patent on this concept, and have taken out a copyright on those words. If you wish to use those words in conjunction with each other, you have to send $1 per usage instance to me. Usage of the patent (Master/Slave Relationships In Computer Applications) will be much more expensive, of course.)

Anyways, you start one Gnus up the normal way with M-x gnus (or however you do it). Each subsequent slave Gnusiï should be started with M-x gnus-slave. These slaves won't save normal `.newsrc' files, but instead save slave files that contains information only on what groups have been read in the slave session. When a master Gnus starts, it will read (and delete) these slave files, incorporating all information from them. (The slave files will be read in the sequence they were created, so the latest changes will have precedence.)

Information from the slave files has, of course, precedence over the information in the normal (i. e., master) `.newsrc` file.

Fetching a Group

It it sometime convenient to be able to just say "I want to read this group and I don't care whether Gnus has been started or not". This is perhaps more useful for people who write code than for users, but the command `gnus-fetch-group` provides this functionality in any case. It takes the group name as a parameter.

New Groups

What Gnus does when it encounters a new group is determined by the `gnus-subscribe-newsgroup-method` variable.

This variable should contain a function. Some handy pre-fab values are:

`gnus-subscribe-zombies`

> Make all new groups zombies. You can browse the zombies later (with A z) and either kill them all off properly, or subscribe to them. This is the default.

`gnus-subscribe-randomly`

> Subscribe all new groups randomly.

`gnus-subscribe-alphabetically`

> Subscribe all new groups alphabetically.

`gnus-subscribe-hierarchically`

Subscribe all new groups hierarchically.

`gnus-subscribe-interactively`

Subscribe new groups interactively. This means that Gnus will ask you about **all** new groups.

`gnus-subscribe-killed`

Kill all new groups.

A closely related variable is `gnus-subscribe-hierarchical-interactive`. (That's quite a mouthful.) If this variable is non-`nil`, Gnus will ask you in a hierarchical fashion whether to subscribe to new groups or not. Gnus will ask you for each sub-hierarchy whether you want to descend the hierarchy or not.

One common mistake is to set the variable a few paragraphs above to `gnus-subscribe-hierarchical-interactive`. This is an error. This will not work. This is ga-ga. So don't do it.

A nice and portable way to control which new newsgroups should be subscribed (or ignored) is to put an options line at the start of the `` `.newsrc' `` file. Here's an example:

```
options -n !alt.all !rec.all sci.all
```

This line obviously belongs to a serious-minded intellectual scientific person (or she may just be plain old boring), because it says that all groups that have names beginning with `alt' and `rec' should be ignored, and all groups with names beginning with `sci' should be subscribed. Gnus will not use the normal subscription method for subscribing these groups. `gnus-subscribe-options-newsgroup-method` is used instead. This variable defaults to `gnus-subscribe-alphabetically`.

If you don't want to mess with your `` `.newsrc' `` file, you can just set the two variables `gnus-options-subscribe` and `gnus-options-not-subscribe`. These two variables do exactly the same as the `` `.newsrc' ``options -n' trick. Both are regexps, and if the the new group matches the former, it will be unconditionally subscribed, and if it matches the latter, it will be ignored.

Yet another variable that meddles here is `gnus-auto-subscribed-groups`. It works exactly like `gnus-options-subscribe`, and is therefore really superfluous, but I thought it would be nice to have two of these. This variable is more meant for setting some ground rules, while the other variable is used more for user fiddling. By default this variable makes all new groups that come from mail backends (`nnml`, `nnbabyl`, `nnfolder`, `nnmbox`, and `nnmh`) subscribed. If you don't like that, just set this variable to `nil`.

If you are satisfied that you really never want to see any new groups, you could set `gnus-check-new-newsgroups` to `nil`. This will also save you some time at startup. Even if this variable is `nil`, you can always subscribe to the new groups just by pressing U in the group buffer (see section Group Maintenance). This variable is `t` by default.

Gnus normally determines whether a group is new or not by comparing the list of groups from the active file(s) with the lists of subscribed and dead groups. This isn't a particularly fast method. If

`gnus-check-new-newsgroups` is `ask-server`, Gnus will ask the server for new groups since the last time. This is both faster & cheaper. This also means that you can get rid of the list of killed groups altogether, so you may set `gnus-save-killed-list` to `nil`, which will save time both at startup, at exit, and all over. Saves disk space, too. Why isn't this the default, then? Unfortunately, not all servers support this command.

I bet I know what you're thinking now: How do I find out whether my server supports `ask-server`? No? Good, because I don't have a fail-safe answer. I would suggest just setting this variable to `ask-server` and see whether any new groups appear within the next few days. If any do, then it works. If any don't, then it doesn't work. I could write a function to make Gnus guess whether the server supports `ask-server`, but it would just be a guess. So I won't. You could `telnet` to the server and say `HELP` and see whether it lists `NEWGROUPS' among the commands it understands. If it does, then it might work. (But there are servers that lists `NEWGROUPS' without supporting the function properly.)

This variable can also be a list of select methods. If so, Gnus will issue an `ask-server` command to each of the select methods, and subscribe them (or not) using the normal methods. This might be handy if you are monitoring a few servers for new groups. A side effect is that startup will take much longer, so you can meditate while waiting. Use the mantra "dingnusdingnusdingnus" to achieve permanent bliss.

Startup Files

Now, you all know about the `` `.newsrc' `` file. All subscription information is traditionally stored in this file.

Things got a bit more complicated with GNUS. In addition to keeping the `` `.newsrc' `` file updated, it also used a file called `` `.newsrc.el' `` for storing all the information that didn't fit into the `` `.newsrc' `` file. (Actually, it also duplicated everything in the `` `.newsrc' `` file.) GNUS would read whichever one of these files was the most recently saved, which enabled people to swap between GNUS and other newsreaders.

That was kinda silly, so Gnus went one better: In addition to the `` `.newsrc' `` and `` `.newsrc.el' `` files, Gnus also has a file called `` `.newsrc.eld' ``. It will read whichever of these files that are most recent, but it will never write a `` `.newsrc.el' `` file.

You can turn off writing the `` `.newsrc' `` file by setting `gnus-save-newsrc-file` to `nil`, which means you can delete the file and save some space, as well as making exit from Gnus faster. However, this will make it impossible to use other newsreaders than Gnus. But hey, who would want to, right?

If `gnus-save-killed-list` (default `t`) is `nil`, Gnus will not save the list of killed groups to the startup file. This will save both time (when starting and quitting) and space (on disk). It will also means that Gnus has no record of what groups are new or old, so the automatic new groups subscription methods become meaningless. You should always set `gnus-check-new-newsgroups` to `nil` or `ask-server` if you set this variable to `nil` (see section New Groups).

The `gnus-startup-file` variable says where the startup files are. The default value is `` `~/.newsrc' ``, with the Gnus (El Dingo) startup file being whatever that one is with a `.eld' appended.

`gnus-save-newsrc-hook` is called before saving any of the newsrc files, while `gnus-save-quick-newsrc-hook` is called just before saving the `.newsrc.eld' file, and `gnus-save-standard-newsrc-hook` is called just before saving the `.newsrc' file. The latter two are commonly used to turn version control on or off. Version control is off by default when saving the startup files.

Auto Save

Whenever you do something that changes the Gnus data (reading articles, catching up, killing/subscribing groups), the change is added to a special dribble buffer. This buffer is auto-saved the normal Emacs way. If your Emacs should crash before you have saved the `.newsrc' files, all changes you have made can be recovered from this file.

If Gnus detects this file at startup, it will ask the user whether to read it. The auto save file is deleted whenever the real startup file is saved.

If `gnus-use-dribble-file` is `nil`, Gnus won't create and maintain a dribble buffer. The default is `t`.

Gnus will put the dribble file(s) in `gnus-dribble-directory`. If this variable is `nil`, which it is by default, Gnus will dribble into the directory where the `.newsrc' file is located. (This is normally the user's home directory.) The dribble file will get the same file permissions as the `.newsrc` file.

The Active File

When Gnus starts, or indeed whenever it tries to determine whether new articles have arrived, it reads the active file. This is a very large file that lists all the active groups and articles on the server.

Before examining the active file, Gnus deletes all lines that match the regexp `gnus-ignored-newsgroups`. This is done primarily to reject any groups with bogus names, but you can use this variable to make Gnus ignore hierarchies you aren't ever interested in. However, this is not recommended. In fact, it's highly discouraged. Instead, see section New Groups for an overview of other variables that can be used instead.

The active file can be rather Huge, so if you have a slow network, you can set `gnus-read-active-file` to `nil` to prevent Gnus from reading the active file. This variable is `t` by default.

Gnus will try to make do by getting information just on the groups that you actually subscribe to.

Note that if you subscribe to lots and lots of groups, setting this variable to `nil` will probably make Gnus slower, not faster. At present, having this variable `nil` will slow Gnus down considerably, unless you read news over a 2400 baud modem.

This variable can also have the value `some`. Gnus will then attempt to read active info only on the subscribed groups. On some servers this is quite fast (on sparkling, brand new INN servers that support the `LIST ACTIVE group` command), on others this isn't fast at all. In any case, `some` should be

faster than `nil`, and is certainly faster than `t` over slow lines.

If this variable is `nil`, Gnus will ask for group info in total lock-step, which isn't very fast. If it is `some` and you use an NNTP server, Gnus will pump out commands as fast as it can, and read all the replies in one swoop. This will normally result in better performance, but if the server does not support the aforementioned `LIST ACTIVE group` command, this isn't very nice to the server.

In any case, if you use `some` or `nil`, you should definitely kill all groups that you aren't interested in to speed things up.

Startup Variables

`gnus-load-hook`

A hook that is run while Gnus is being loaded. Note that this hook will normally be run just once in each Emacs session, no matter how many times you start Gnus.

`gnus-startup-hook`

A hook that is run after starting up Gnus successfully.

`gnus-check-bogus-newsgroups`

If non-`nil`, Gnus will check for and delete all bogus groups at startup. A bogus group is a group that you have in your `` `.newsrc' `` file, but doesn't exist on the news server. Checking for bogus groups can take quite a while, so to save time and resources it's best to leave this option off, and do the checking for bogus groups once in a while from the group buffer instead (see section Group Maintenance).

`gnus-inhibit-startup-message`

If non-`nil`, the startup message won't be displayed. That way, your boss might not notice that you are reading news instead of doing your job as easily.

`gnus-no-groups-message`

Message displayed by Gnus when no groups are available.

Go to the next section.

Go to the previous, next section.

The Group Buffer

The group buffer lists all (or parts) of the available groups. It is the first buffer shown when Gnus starts, and will never be killed as long as Gnus is active.

Group Buffer Format

Group Line Specification

The default format of the group buffer is nice and dull, but you can make it as exciting and ugly as you feel like.

Here's a couple of example group lines:

```
     25: news.announce.newusers
 *    0: alt.fan.andrea-dworkin
```

Quite simple, huh?

You can see that there are 25 unread articles in `news.announce.newusers'. There are no unread articles, but some ticked articles, in `alt.fan.andrea-dworkin' (see that little asterisk at the beginning of the line?)

You can change that format to whatever you want by fiddling with the `gnus-group-line-format` variable. This variable works along the lines of a `format` specification, which is pretty much the same as a `printf` specifications, for those of you who use (feh!) C. See section Formatting Variables.

The default value that produced those lines above is `%M%S%5y: %(%g%)\n'.

There should always be a colon on the line; the cursor always moves to the colon after performing an operation. Nothing else is required--not even the group name. All displayed text is just window dressing, and is never examined by Gnus. Gnus stores all real information it needs using text properties.

(Note that if you make a really strange, wonderful, spreadsheet-like layout, everybody will believe you are hard at work with the accounting instead of wasting time reading news.)

Here's a list of all available format characters:

`M'

Only marked articles.

`S'

Whether the group is subscribed.

`L'

Level of subscribedness.

`N'
: Number of unread articles.

`I'
: Number of dormant articles.

`T'
: Number of ticked articles.

`R'
: Number of read articles.

`t'
: Total number of articles.

`y'
: Number of unread, unticked, non-dormant articles.

`i'
: Number of ticked and dormant articles.

`g'
: Full group name.

`G'
: Group name.

`D'
: Newsgroup description.

`o'
: `m' if moderated.

`O'
: `(m)' if moderated.

`s'
: Select method.

`n'
: Select from where.

`z'
: A string that looks like `<%s:%n>' if a foreign select method is used.

`P'
: Indentation based on the level of the topic (see section Group Topics).

`c'
: Short (collapsed) group name. The `gnus-group-uncollapsed-levels` variable says how many levels to leave at the end of the group name. The default is `1`.

`u'
: User defined specifier. The next character in the format string should be a letter. GNUS will call

the function `gnus-user-format-function-`X', where `X' is the letter following `%u'. The function will be passed the current headers as argument. The function should return a string, which will be inserted into the buffer just like information from any other specifier.

All the "number-of" specs will be filled with an asterisk (`*') if no info is available--for instance, if it is a non-activated foreign group, or a bogus (or semi-bogus) native group.

Group Modeline Specification

The mode line can be changed by setting (`gnus-group-mode-line-format`). It doesn't understand that many format specifiers:

`S'

The native news server.

`M'

The native select method.

Group Highlighting

Highlighting in the group buffer is controlled by the `gnus-group-highlight` variable. This is an alist with elements that look like (form . face). If form evaluates to something non-`nil`, the face will be used on the line.

Here's an example value for this variable that might look nice if the background is dark:

```
(setq gnus-group-highlight
    `(((> unread 200) .
       ,(custom-face-lookup "Red" nil nil t nil nil))
      ((and (< level 3) (zerop unread)) .
       ,(custom-face-lookup "SeaGreen" nil nil t nil nil))
      ((< level 3) .
       ,(custom-face-lookup "SpringGreen" nil nil t nil nil))
      ((zerop unread) .
       ,(custom-face-lookup "SteelBlue" nil nil t nil nil))
      (t .
       ,(custom-face-lookup "SkyBlue" nil nil t nil nil))
      ))
```

Variables that are dynamically bound when the forms are evaluated include:

`group`

The group name.

`unread`

The number of unread articles in the group.

`method`

The select method.

`mailp`
: Whether the group is a mail group.

`level`
: The level of the group.

`score`
: The score of the group.

`ticked`
: The number of ticked articles in the group.

`topic`
: When using the topic minor mode, this variable is bound to the current topic being inserted.

When the forms are `eval`ed, point is at the beginning of the line of the group in question, so you can use many of the normal Gnus functions for snarfing info on the group.

`gnus-group-update-hook` is called when a group line is changed. It will not be called when `gnus-visual` is `nil`. This hook calls `gnus-group-highlight-line` by default.

Group Maneuvering

All movement commands understand the numeric prefix and will behave as expected, hopefully.

n
: Go to the next group that has unread articles (`gnus-group-next-unread-group`).

p

DEL
: Go to the previous group group that has unread articles (`gnus-group-prev-unread-group`).

N
: Go to the next group (`gnus-group-next-group`).

P
: Go to the previous group (`gnus-group-prev-group`).

M-p
: Go to the next unread group on the same level (or lower) (`gnus-group-next-unread-group-same-level`).

M-n
: Go to the previous unread group on the same level (or lower) (`gnus-group-prev-unread-group-same-level`).

Three commands for jumping to groups:

j
: Jump to a group (and make it visible if it isn't already) (`gnus-group-jump-to-group`).

Killed groups can be jumped to, just like living groups.

,

Jump to the unread group with the lowest level (`gnus-group-best-unread-group`).

.

Jump to the first group with unread articles (`gnus-group-first-unread-group`).

If `gnus-group-goto-unread` is `nil`, all the movement commands will move to the next group, not the next unread group. Even the commands that say they move to the next unread group. The default is `t`.

Selecting a Group

SPACE

Select the current group, switch to the summary buffer and display the first unread article (`gnus-group-read-group`). If there are no unread articles in the group, or if you give a non-numerical prefix to this command, Gnus will offer to fetch all the old articles in this group from the server. If you give a numerical prefix N, Gnus will fetch N number of articles. If N is positive, fetch the N newest articles, if N is negative, fetch the abs(N) oldest articles.

RET

Select the current group and switch to the summary buffer (`gnus-group-select-group`). Takes the same arguments as `gnus-group-read-group`---the only difference is that this command does not display the first unread article automatically upon group entry.

M-RET

This does the same as the command above, but tries to do it with the minimum amount off fuzz (`gnus-group-quick-select-group`). No scoring/killing will be performed, there will be no highlights and no expunging. This might be useful if you're in a real hurry and have to enter some humongous group.

M-SPACE

This is yet one more command that does the same as the one above, but this one does it without expunging and hiding dormants (`gnus-group-visible-select-group`).

c

Mark all unticked articles in this group as read (`gnus-group-catchup-current`). `gnus-group-catchup-group-hook` is when catching up a group from the group buffer.

C

Mark all articles in this group, even the ticked ones, as read (`gnus-group-catchup-current-all`).

The `gnus-large-newsgroup` variable says what Gnus should consider to be a big group. This is 200 by default. If the group has more unread articles than this, Gnus will query the user before entering the group. The user can then specify how many articles should be fetched from the server. If the user specifies a negative number (`-n`), the `n` oldest articles will be fetched. If it is positive, the `n` articles that have arrived most recently will be fetched.

`gnus-auto-select-first` control whether any articles are selected automatically when entering a group.

`nil`

Don't select any articles when entering the group. Just display the full summary buffer.

`t`

Select the first unread article when entering the group.

`best`

Select the most high-scored article in the group when entering the group.

If you want to prevent automatic selection in some group (say, in a binary group with Huge articles) you can set this variable to `nil` in `gnus-select-group-hook`, which is called when a group is selected.

Subscription Commands

S t

u

Toggle subscription to the current group (`gnus-group-unsubscribe-current-group`).

S s

U

Prompt for a group to subscribe, and then subscribe it. If it was subscribed already, unsubscribe it instead (`gnus-group-unsubscribe-group`).

S k

C-k

Kill the current group (`gnus-group-kill-group`).

S y

C-y

Yank the last killed group (`gnus-group-yank-group`).

C-x C-t

Transpose two groups (`gnus-group-transpose-groups`). This isn't really a subscription command, but you can use it instead of a kill-and-yank sequence sometimes.

S w

C-w

Kill all groups in the region (`gnus-group-kill-region`).

S z

Kill all zombie groups (`gnus-group-kill-all-zombies`).

S C-k

Kill all groups on a certain level (`gnus-group-kill-level`). These groups can't be yanked back after killing, so this command should be used with some caution. The only thing where this

command comes in really handy is when you have a `.newsrc' with lots of unsubscribed groups that you want to get rid off. S C-k on level 7 will kill off all unsubscribed groups that do not have message numbers in the `.newsrc' file.

Also see section Group Levels.

Group Levels

All groups have a level of subscribedness. For instance, if a group is on level 2, it is more subscribed than a group on level 5. You can ask Gnus to just list groups on a given level or lower (see section Listing Groups), or to just check for new articles in groups on a given level or lower (see section Scanning New Messages).

Remember: The higher the level of the group, the less important it is.

S l

Set the level of the current group. If a numeric prefix is given, the next n groups will have their levels set. The user will be prompted for a level.

Gnus considers groups on between levels 1 and `gnus-level-subscribed` (inclusive) (default 5) to be subscribed, `gnus-level-subscribed` (exclusive) and `gnus-level-unsubscribed` (inclusive) (default 7) to be unsubscribed, `gnus-level-zombie` to be zombies (walking dead) (default 8) and `gnus-level-killed` to be killed (default 9), completely dead. Gnus treats subscribed and unsubscribed groups exactly the same, but zombie and killed groups have no information on what articles you have read, etc, stored. This distinction between dead and living groups isn't done because it is nice or clever, it is done purely for reasons of efficiency.

It is recommended that you keep all your mail groups (if any) on quite low levels (eg. 1 or 2).

If you want to play with the level variables, you should show some care. Set them once, and don't touch them ever again. Better yet, don't touch them at all unless you know exactly what you're doing.

Two closely related variables are `gnus-level-default-subscribed` (default 3) and `gnus-level-default-unsubscribed` (default 6), which are the levels that new groups will be put on if they are (un)subscribed. These two variables should, of course, be inside the relevant legal ranges.

If `gnus-keep-same-level` is non-`nil`, some movement commands will only move to groups that are of the same level (or lower). In particular, going from the last article in one group to the next group will go to the next group of the same level (or lower). This might be handy if you want to read the most important groups before you read the rest.

All groups with a level less than or equal to `gnus-group-default-list-level` will be listed in the group buffer by default.

If `gnus-group-list-inactive-groups` is non-`nil`, non-active groups will be listed along with the unread groups. This variable is `t` by default. If it is `nil`, inactive groups won't be listed.

If `gnus-group-use-permanent-levels` is non-`nil`, once you give a level prefix to g or l, all

subsequent commands will use this level as the "work" level.

Gnus will normally just activate groups that are on level `gnus-activate-level` or less. If you don't want to activate unsubscribed groups, for instance, you might set this variable to `5`.

Group Score

You would normally keep important groups on high levels, but that scheme is somewhat restrictive. Don't you wish you could have Gnus sort the group buffer according to how often you read groups, perhaps? Within reason?

This is what group score is for. You can assign a score to each group. You can then sort the group buffer based on this score. Alternatively, you can sort on score and then level. (Taken together, the level and the score is called the rank of the group. A group that is on level 4 and has a score of 1 has a higher rank than a group on level 5 that has a score of 300. (The level is the most significant part and the score is the least significant part.)

If you want groups you read often to get higher scores than groups you read seldom you can add the `gnus-summary-bubble-group` function to the `gnus-summary-exit-hook` hook. This will result (after sorting) in a bubbling sort of action. If you want to see that in action after each summary exit, you can add `gnus-group-sort-groups-by-rank` or `gnus-group-sort-groups-by-score` to the same hook, but that will slow things down somewhat.

Marking Groups

If you want to perform some command on several groups, and they appear subsequently in the group buffer, you would normally just give a numerical prefix to the command. Most group commands will then do your bidding on those groups.

However, if the groups are not in sequential order, you can still perform a command on several groups. You simply mark the groups first with the process mark and then execute the command.

#

M m

Set the mark on the current group (`gnus-group-mark-group`).

M-#

M u

Remove the mark from the current group (`gnus-group-unmark-group`).

M U

Remove the mark from all groups (`gnus-group-unmark-all-groups`).

M w

Mark all groups between point and mark (`gnus-group-mark-region`).

M b

Mark all groups in the buffer (`gnus-group-mark-buffer`).

M r

Mark all groups that match some regular expression (`gnus-group-mark-regexp`).

Also see section Process/Prefix.

If you want to execute some command on all groups that have been marked with the process mark, you can use the M-& (`gnus-group-universal-argument`) command. It will prompt you for the command to be executed.

Foreign Groups

Here are some group mode commands for making and editing general foreign groups, as well as commands to ease the creation of a few special-purpose groups:

G m

Make a new group (`gnus-group-make-group`). Gnus will prompt you for a name, a method and possibly an address. For an easier way to subscribe to NNTP groups, see section Browse Foreign Server.

G r

Rename the current group to something else (`gnus-group-rename-group`). This is legal only on some groups -- mail groups mostly. This command might very well be quite slow on some backends.

G e

Enter a buffer where you can edit the select method of the current group (`gnus-group-edit-group-method`).

G p

Enter a buffer where you can edit the group parameters (`gnus-group-edit-group-parameters`).

G E

Enter a buffer where you can edit the group info (`gnus-group-edit-group`).

G d

Make a directory group. You will be prompted for a directory name (`gnus-group-make-directory-group`).

G h

Make the Gnus help group (`gnus-group-make-help-group`).

G a

Make a Gnus archive group (`gnus-group-make-archive-group`). By default a group pointing to the most recent articles will be created (`gnus-group-recent-archive-directory`), but given a prefix, a full group will be created from from `gnus-group-archive-directory`.

G k

Make a kiboze group. You will be prompted for a name, for a regexp to match groups to be "included" in the kiboze group, and a series of strings to match on headers (`gnus-group-make-kiboze-group`). See section Kibozed Groups

G D

Read an arbitrary directory as if with were a newsgroup with the `nneething` backend (`gnus-group-enter-directory`).

G f

Make a group based on some file or other (`gnus-group-make-doc-group`). If you give a prefix to this command, you will be prompted for a file name and a file type. Currently supported types are `babyl`, `mbox`, `digest`, `mmdf`, `news`, `rnews`, `clari-briefs`, and `forward`. If you run this command without a prefix, Gnus will guess at the file type.

G DEL

This function will delete the current group (`gnus-group-delete-group`). If given a prefix, this function will actually delete all the articles in the group, and forcibly remove the group itself from the face of the Earth. Use a prefix only if you are absolutely sure of what you are doing.

G V

Make a new, fresh, empty `nnvirtual` group (`gnus-group-make-empty-virtual`).

G v

Add the current group to an `nnvirtual` group (`gnus-group-add-to-virtual`). Uses the process/prefix convention.

See section Select Methods for more information on the various select methods.

If the `gnus-activate-foreign-newsgroups` is a positive number, Gnus will check all foreign groups with this level or lower at startup. This might take quite a while, especially if you subscribe to lots of groups from different NNTP servers.

Group Parameters

Gnus stores all information on a group in a list that is usually known as the group info. This list has from three to six elements. Here's an example info.

```
("nnml:mail.ding" 3 ((1 . 232) 244 (256 . 270)) ((tick 246 249))
                  (nnml "private") ((to-address . "ding@ifi.uio.no")))
```

The first element is the group name, as Gnus knows the group, anyway. The second element is the subscription level, which normally is a small integer. The third element is a list of ranges of read articles. The fourth element is a list of lists of article marks of various kinds. The fifth element is the select method (or virtual server, if you like). The sixth element is a list of group parameters, which is what this section is about.

Any of the last three elements may be missing if they are not required. In fact, the vast majority of groups

will normally only have the first three elements, which saves quite a lot of cons cells.

The group parameters store information local to a particular group:

`to-address`

If the group parameter list contains an element that looks like `(to-address . "some@where.com")`, that address will be used by the backend when doing followups and posts. This is primarily useful in mail groups that represent closed mailing lists--mailing lists where it's expected that everybody that writes to the mailing list is subscribed to it. Since using this parameter ensures that the mail only goes to the mailing list itself, it means that members won't receive two copies of your followups.

Using `to-address` will actually work whether the group is foreign or not. Let's say there's a group on the server that is called `fa.4ad-l'. This is a real newsgroup, but the server has gotten the articles from a mail-to-news gateway. Posting directly to this group is therefore impossible--you have to send mail to the mailing list address instead.

`to-list`

If the group parameter list has an element that looks like `(to-list . "some@where.com")`, that address will be used when doing a a in any group. It is totally ignored when doing a followup--except that if it is present in a news group, you'll get mail group semantics when doing f.

`broken-reply-to`

Elements like `(broken-reply-to . t)` signals that `Reply-To` headers in this group are to be ignored. This can be useful if you're reading a mailing list group where the listserv has inserted `Reply-To` headers that point back to the listserv itself. This is broken behavior. So there!

`to-group`

If the group parameter list contains an element like `(to-group . "some.group.name")`, all posts will be sent to that group.

`auto-expire`

If this symbol is present in the group parameter list, all articles that are read will be marked as expirable. For an alternative approach, see section Expiring Mail.

`total-expire`

If this symbol is present, all read articles will be put through the expiry process, even if they are not marked as expirable. Use with caution.

`expiry-wait`

If the group parameter has an element that looks like `(expiry-wait . 10)`, this value will override any `nnmail-expiry-wait` and `nnmail-expiry-wait-function` when expiring expirable messages. The value can either be a number of days (not necessarily an integer) or the symbols `never` or `immediate`.

`score-file`

Elements that look like `(score-file . "file")` will make `file' into the current score file for the group in question. This means that all score commands you issue will end up in that file.

`admin-address`

When unsubscribing to a mailing list you should never send the unsubscription notice to the mailing list itself. Instead, you'd send messages to the administrative address. This parameter allows you to put the admin address somewhere convenient.

`comment`

This parameter allows you to enter a arbitrary comment on the group.

`(variable form)`

You can use the group parameters to set variables local to the group you are entering. Say you want to turn threading off in `news.answers'. You'd then put `(gnus-show-threads nil)` in the group parameters of that group. `gnus-show-threads` will be made into a local variable in the summary buffer you enter, and the form `nil` will be `eval`ed there.

This can also be used as a group-specific hook function, if you'd like. If you want to hear a beep when you enter the group `alt.binaries.pictures.furniture', you could put something like `(dummy-variable (ding))` in the parameters of that group. `dummy-variable` will be set to the result of the `(ding)` form, but who cares?

If you want to change the group info you can use the G E command to enter a buffer where you can edit it.

You usually don't want to edit the entire group info, so you'd be better off using the G p command to just edit the group parameters.

Listing Groups

These commands all list various slices of the groups that are available.

l

A s

List all groups that have unread articles (`gnus-group-list-groups`). If the numeric prefix is used, this command will list only groups of level ARG and lower. By default, it only lists groups of level five or lower (i.e., just subscribed groups).

L

A u

List all groups, whether they have unread articles or not (`gnus-group-list-all-groups`). If the numeric prefix is used, this command will list only groups of level ARG and lower. By default, it lists groups of level seven or lower (i.e., just subscribed and unsubscribed groups).

A l

List all unread groups on a specific level (`gnus-group-list-level`). If given a prefix, also list the groups with no unread articles.

A k

List all killed groups (`gnus-group-list-killed`). If given a prefix argument, really list all groups that are available, but aren't currently (un)subscribed. This could entail reading the active file from the server.

A z

List all zombie groups (`gnus-group-list-zombies`).

A m

List all subscribed groups with unread articles that match a regexp (`gnus-group-list-matching`).

A M

List groups that match a regexp (`gnus-group-list-all-matching`).

A A

List absolutely all groups that are in the active file(s) of the server(s) you are connected to (`gnus-group-list-active`). This might very well take quite a while. It might actually be a better idea to do a A m to list all matching, and just give `.' as the thing to match on.

A a

List all groups that have names that match a regexp (`gnus-group-apropos`).

A d

List all groups that have names or descriptions that match a regexp (`gnus-group-description-apropos`).

Groups that match the `gnus-permanently-visible-groups` regexp will always be shown, whether they have unread articles or not. You can also add the `visible` element to the group parameters in question to get the same effect.

Groups that have just ticked articles in it are normally listed in the group buffer. If `gnus-list-groups-with-ticked-articles` is `nil`, these groups will be treated just like totally empty groups. It is `t` by default.

Sorting Groups

The C-c C-s (`gnus-group-sort-groups`) command sorts the group buffer according to the function(s) given by the `gnus-group-sort-function` variable. Available sorting functions include:

`gnus-group-sort-by-alphabet`

Sort the group names alphabetically. This is the default.

`gnus-group-sort-by-level`

Sort by group level.

`gnus-group-sort-by-score`

Sort by group score.

`gnus-group-sort-by-rank`

Sort by group score and then the group level. The level and the score are, when taken together, the group's rank.

`gnus-group-sort-by-unread`

Sort by number of unread articles.

`gnus-group-sort-by-method`

Sort by alphabetically on the select method.

`gnus-group-sort-function` can also be a list of sorting functions. In that case, the most significant sort key function must be the last one.

There are also a number of commands for sorting directly according to some sorting criteria:

G S a

Sort the group buffer alphabetically by group name (`gnus-group-sort-groups-by-alphabet`).

G S u

Sort the group buffer by the number of unread articles (`gnus-group-sort-groups-by-unread`).

G S l

Sort the group buffer by group level (`gnus-group-sort-groups-by-level`).

G S v

Sort the group buffer by group score (`gnus-group-sort-groups-by-score`).

G S r

Sort the group buffer by group level (`gnus-group-sort-groups-by-rank`).

G S m

Sort the group buffer alphabetically by backend name (`gnus-group-sort-groups-by-method`).

When given a prefix, all these commands will sort in reverse order.

Group Maintenance

b

Find bogus groups and delete them (`gnus-group-check-bogus-groups`).

F

Find new groups and process them (`gnus-find-new-newsgroups`). If given a prefix, use the `ask-server` method to query the server for new groups.

C-c C-x

Run all expirable articles in the current group through the expiry process (if any) (`gnus-group-expire-articles`).

C-c M-C-x

Run all articles in all groups through the expiry process (`gnus-group-expire-all-groups`).

Browse Foreign Server

B

You will be queried for a select method and a server name. Gnus will then attempt to contact this server and let you browse the groups there (`gnus-group-browse-foreign-server`).

A new buffer with a list of available groups will appear. This buffer will be use the `gnus-browse-mode`. This buffer looks a bit (well, a lot) like a normal group buffer, but with one major difference - you can't enter any of the groups. If you want to read any of the news available on that server, you have to subscribe to the groups you think may be interesting, and then you have to exit this buffer. The new groups will be added to the group buffer, and then you can read them as you would any other group.

Future versions of Gnus may possibly permit reading groups straight from the browse buffer.

Here's a list of keystrokes available in the browse mode:

n

Go to the next group (`gnus-group-next-group`).

p

Go to the previous group (`gnus-group-prev-group`).

SPACE

Enter the current group and display the first article (`gnus-browse-read-group`).

RET

Enter the current group (`gnus-browse-select-group`).

u

Unsubscribe to the current group, or, as will be the case here, subscribe to it (`gnus-browse-unsubscribe-current-group`).

l

q

Exit browse mode (`gnus-browse-exit`).

?

Describe browse mode briefly (well, there's not much to describe, is there) (`gnus-browse-describe-briefly`).

Exiting Gnus

Yes, Gnus is ex(c)iting.

z

Suspend Gnus (`gnus-group-suspend`). This doesn't really exit Gnus, but it kills all buffers except the Group buffer. I'm not sure why this is a gain, but then who am I to judge?

q
: Quit Gnus (`gnus-group-exit`).

Q
: Quit Gnus without saving any startup files (`gnus-group-quit`).

`gnus-suspend-gnus-hook` is called when you suspend Gnus and `gnus-exit-gnus-hook` is called when you quit Gnus, while `gnus-after-exiting-gnus-hook` is called as the final item when exiting Gnus.

If you wish to completely unload Gnus and all its adherents, you can use the `gnus-unload` command. This command is also very handy when trying to customize meta-variables.

Note:

> Miss Lisa Cannifax, while sitting in English class, feels her feet go numbly heavy and herself fall into a hazy trance as the boy sitting behind her drew repeated lines with his pencil across the back of her plastic chair.

Group Topics

If you read lots and lots of groups, it might be convenient to group them hierarchically according to topics. You put your Emacs groups over here, your sex groups over there, and the rest (what, two groups or so?) you put in some misc section that you never bother with anyway. You can even group the Emacs sex groups as a sub-topic to either the Emacs groups or the sex groups--or both! Go wild!

To get this *fab* functionality you simply turn on (ooh!) the `gnus-topic` minor mode--type t in the group buffer. (This is a toggling command.)

Go ahead, just try it. I'll still be here when you get back. La de dum... Nice tune, that... la la la... What, you're back? Yes, and now press l. There. All your groups are now listed under `misc'. Doesn't that make you feel all warm and fuzzy? Hot and bothered?

If you want this permanently enabled, you should add that minor mode to the hook for the group mode:

```
(add-hook 'gnus-group-mode-hook 'gnus-topic-mode)
```

Topic Variables

Now, if you select a topic, if will fold/unfold that topic, which is really neat, I think.

The topic lines themselves are created according to the `gnus-topic-line-format` variable. See section Formatting Variables. Elements allowed are:

`i'
: Indentation.

`n'
: Topic name.

`v'

Visibility.

`l'

Level.

`g'

Number of groups in the topic.

`a'

Number of unread articles in the topic.

`A'

Number of unread articles in the topic and all its subtopics.

Each sub-topic (and the groups in the sub-topics) will be indented with `gnus-topic-indent-level` times the topic level number of spaces. The default is `2`.

`gnus-topic-mode-hook` is called in topic minor mode buffers.

Topic Commands

When the topic minor mode is turned on, a new T submap will be available. In addition, a few of the standard keys change their definitions slightly.

T n

Prompt for a new topic name and create it (`gnus-topic-create-topic`).

T m

Move the current group to some other topic (`gnus-topic-move-group`). This command understands the process/prefix convention (see section Process/Prefix).

T c

Copy the current group to some other topic (`gnus-topic-copy-group`). This command understands the process/prefix convention (see section Process/Prefix).

T D

Remove a group from the current topic (`gnus-topic-remove-group`). This command understands the process/prefix convention (see section Process/Prefix).

T M

Move all groups that match some regular expression to a topic (`gnus-topic-move-matching`).

T C

Copy all groups that match some regular expression to a topic (`gnus-topic-copy-matching`).

T #

Mark all groups in the current topic with the process mark (`gnus-topic-mark-topic`).

T M-#

Remove the process mark from all groups in the current topic (`gnus-topic-unmark-top`).

RET

SPACE

Either select a group or fold a topic (`gnus-topic-select-group`). When you perform this command on a group, you'll enter the group, as usual. When done on a topic line, the topic will be folded (if it was visible) or unfolded (if it was folded already). So it's basically a toggling command on topics. In addition, if you give a numerical prefix, group on that level (and lower) will be displayed.

T TAB

"Indent" the current topic so that it becomes a sub-topic of the previous topic (`gnus-topic-indent`). If given a prefix, "un-indent" the topic instead.

C-k

Kill a group or topic (`gnus-topic-kill-group`).

C-y

Yank the previously killed group or topic (`gnus-topic-yank-group`). Note that all topics will be yanked before all groups.

T r

Rename a topic (`gnus-topic-rename`).

T DEL

Delete an empty topic (`gnus-topic-delete`).

A T

List all groups that Gnus knows about in a topics-ified way (`gnus-topic-list-active`).

Topic Topology

So, let's have a look at an example group buffer:

```
Gnus
  Emacs -- I wuw it!
       3: comp.emacs
       2: alt.religion.emacs
    Naughty Emacs
     452: alt.sex.emacs
       0: comp.talk.emacs.recovery
  Misc
       8: comp.binaries.fractals
      13: comp.sources.unix
```

So, here we have one top-level topic, two topics under that, and one sub-topic under one of the sub-topics. (There is always just one (1) top-level topic). This topology can be expressed as follows:


```
(("Gnus" visible)
 (("Emacs -- I wuw it!" visible)
  (("Naughty Emacs" visible)))
 (("Misc" visible)))
```

This is in fact how the variable `gnus-topic-topology` would look for the display above. That variable is saved in the `.newsrc.eld' file, and shouldn't be messed with manually--unless you really want to. Since this variable is read from the `.newsrc.eld' file, setting it in any other startup files will have no effect.

This topology shows what topics are sub-topics of what topics (right), and which topics are visible. Two settings are currently allowed---`visible` and `invisible`.

Misc Group Stuff

^

Enter the server buffer (`gnus-group-enter-server-mode`). See section The Server Buffer.

a

Post an article to a group (`gnus-group-post-news`). The current group name will be used as the default.

m

Mail a message somewhere (`gnus-group-mail`).

Variables for the group buffer:

`gnus-group-mode-hook`

`gnus-group-mode-hook` is called after the group buffer has been created.

`gnus-group-prepare-hook`

`gnus-group-prepare-hook` is called after the group buffer is generated. It may be used to modify the buffer in some strange, unnatural way.

`gnus-permanently-visible-groups`

Groups matching this regexp will always be listed in the group buffer, whether they are empty or not.

Scanning New Messages

g

Check the server(s) for new articles. If the numerical prefix is used, this command will check only groups of level arg and lower (`gnus-group-get-new-news`). If given a non-numerical prefix, this command will force a total rereading of the active file(s) from the backend(s).

M-g

Check whether new articles have arrived in the current group (`gnus-group-get-new-news-this-group`). The

`gnus-goto-next-group-when-activating` variable controls whether this command moves point to the next group or not. It is `t` by default.

C-c M-g

Activate absolutely all groups (`gnus-activate-all-groups`).

R

Restart Gnus (`gnus-group-restart`).

`gnus-get-new-news-hook` is run just before checking for new news.

`gnus-after-getting-new-news-hook` is run after checking for new news.

Group Information

M-f

Try to fetch the FAQ for the current group (`gnus-group-fetch-faq`). Gnus will try to get the FAQ from `gnus-group-faq-directory`, which is usually a directory on a remote machine. `ange-ftp` will be used for fetching the file.

D

Describe the current group (`gnus-group-describe-group`). If given a prefix, force Gnus to re-read the description from the server.

M-d

Describe all groups (`gnus-group-describe-all-groups`). If given a prefix, force Gnus to re-read the description file from the server.

V

Display current Gnus version numbers (`gnus-version`).

?

Give a very short help message (`gnus-group-describe-briefly`).

C-c C-i

Go to the Gnus info node (`gnus-info-find-node`).

File Commands

r

Read the init file (`gnus-init-file`, which defaults to `` `~/.gnus' ``) (`gnus-group-read-init-file`).

s

Save the `` `.newsrc.eld' `` file (and `` `.newsrc' `` if wanted) (`gnus-group-save-newsrc`). If given a prefix, force saving the file(s) whether Gnus thinks it is necessary or not.

Go to the previous, next section.

Go to the previous, next section.

The Summary Buffer

A line for each article is displayed in the summary buffer. You can move around, read articles, post articles and reply to articles.

Summary Buffer Format

Gnus will use the value of the `gnus-extract-address-components` variable as a function for getting the name and address parts of a `From` header. Two pre-defined function exist: `gnus-extract-address-components`, which is the default, quite fast, and too simplistic solution; and `mail-extract-address-components`, which works very nicely, but is slower. The default function will return the wrong answer in 5% of the cases. If this is unacceptable to you, use the other function instead.

`gnus-summary-same-subject` is a string indicating that the current article has the same subject as the previous. This string will be used with those specs that require it. The default is `'.

Summary Buffer Lines

You can change the format of the lines in the summary buffer by changing the `gnus-summary-line-format` variable. It works along the same lines a a normal `format` string, with some extensions.

The default string is `%U%R%z%I%(%[%4L: %-20,20n%]%) %s\n'.

The following format specification characters are understood:

`N'
: Article number.

`S'
: Subject string.

`s'
: Subject if the article is the root, `gnus-summary-same-subject` otherwise.

`F'
: Full `From` line.

`n'
: The name (from the `From` header).

`a'
: The name (from the `From` header). This differs from the `n` spec in that it uses `gnus-extract-address-components`, which is slower, but may be more thorough.

`A'

The address (from the `From` header). This works the same way as the `a` spec.

`L'

Number of lines in the article.

`c'

Number of characters in the article.

`I'

Indentation based on thread level (see section Customizing Threading).

`T'

Nothing if the article is a root and lots of spaces if it isn't (it pushes everything after it off the screen).

`\['

Opening bracket, which is normally `\[', but can also be `<' for adopted articles.

`\]'

Closing bracket, which is normally `\]', but can also be `>' for adopted articles.

`>'

One space for each thread level.

`<'

Twenty minus thread level spaces.

`U'

Unread.

`R'

Replied.

`i'

Score as a number.

`z'

Zcore, `+' if above the default level and `-' if below the default level. If the difference between `gnus-summary-default-level` and the score is less than `gnus-summary-zcore-fuzz`, this spec will not be used.

`V'

Total thread score.

`x'

`Xref`.

`D'

`Date`.

`M'

`Message-ID`.

`r'

`References`.

`t'

Number of articles in the current sub-thread. Using this spec will slow down summary buffer generation somewhat.

`e'

A single character will be displayed if the article has any children.

`u'

User defined specifier. The next character in the format string should be a letter. GNUS will call the function `gnus-user-format-function-`X', where `X' is the letter following `%u'. The function will be passed the current header as argument. The function should return a string, which will be inserted into the summary just like information from any other summary specifier.

The `%U' (status), `%R' (replied) and `%z' (zcore) specs have to be handled with care. For reasons of efficiency, Gnus will compute what column these characters will end up in, and "hard-code" that. This means that it is illegal to have these specs after a variable-length spec. Well, you might not be arrested, but your summary buffer will look strange, which is bad enough.

The smart choice is to have these specs as far to the left as possible. (Isn't that the case with everything, though? But I digress.)

This restriction may disappear in later versions of Gnus.

Summary Buffer Mode Line

You can also change the format of the summary mode bar. Set `gnus-summary-mode-line-format` to whatever you like. Here are the elements you can play with:

`G'

Group name.

`p'

Unprefixed group name.

`A'

Current article number.

`V'

Gnus version.

`U'

Number of unread articles in this group.

`e'

Number of unselected articles in this group.

`Z'

A string with the number of unread and unselected articles represented either as `<%U(+%u) more>' if there are both unread and unselected articles, and just as `<%U more>' if there are just unread articles and no unselected ones.

`g'

Shortish group name. For instance, `rec.arts.anime' will be shortened to `r.a.anime'.

`S'

Subject of the current article.

`u'

Used-defined spec.

`s'

Name of the current score file.

`d'

Number of dormant articles.

`t'

Number of ticked articles.

`r'

Number of articles that have been marked as read in this session.

`E'

Number of articles expunged by the score files.

Summary Highlighting

`gnus-visual-mark-article-hook`

This hook is run after selecting an article. It is meant to be used for highlighting the article in some way. It is not run if `gnus-visual` is `nil`.

`gnus-summary-update-hook`

This hook is called when a summary line is changed. It is not run if `gnus-visual` is `nil`.

`gnus-summary-selected-face`

This is the face (or font as some people call it) that is used to highlight the current article in the summary buffer.

`gnus-summary-highlight`

Summary lines are highlighted according to this variable, which is a list where the elements are on the format `(FORM . FACE)`. If you would, for instance, like ticked articles to be italic and high-scored articles to be bold, you could set this variable to something like

```
(((eq mark gnus-ticked-mark) . italic)
 ((> score default) . bold))
```

As you may have guessed, if FORM returns a non-`nil` value, FACE will be applied to the line.

Summary Maneuvering

All the straight movement commands understand the numeric prefix and behave pretty much as you'd expect.

None of these commands select articles.

G M-n

M-n

Go to the next summary line of an unread article (`gnus-summary-next-unread-subject`).

G M-p

M-p

Go to the previous summary line of an unread article (`gnus-summary-prev-unread-subject`).

G j

j

Ask for an article number and then go that article (`gnus-summary-goto-article`).

G g

Ask for an article number and then go the summary line of that article (`gnus-summary-goto-subject`).

If Gnus asks you to press a key to confirm going to the next group, you can use the C-n and C-p keys to move around the group buffer, searching for the next group to read without actually returning to the group buffer.

Variables related to summary movement:

`gnus-auto-select-next`

If you are at the end of the group and issue one of the movement commands, Gnus will offer to go to the next group. If this variable is `t` and the next group is empty, Gnus will exit summary mode and return to the group buffer. If this variable is neither `t` nor `nil`, Gnus will select the next group, no matter whether it has any unread articles or not. As a special case, if this variable is `quietly`, Gnus will select the next group without asking for confirmation. If this variable is `almost-quietly`, the same will happen only if you are located on the last article in the group. Finally, if this variable is `slightly-quietly`, the Z n command will go to the next group without confirmation. Also see section Group Levels.

`gnus-auto-select-same`

If non-`nil`, all the movement commands will try to go to the next article with the same subject as the current. This variable is not particularly useful if you use a threaded display.

`gnus-summary-check-current`

If non-`nil`, all the "unread" movement commands will not proceed to the next (or previous) article if the current article is unread. Instead, they will choose the current article.

`gnus-auto-center-summary`

If non-`nil`, Gnus will keep the point in the summary buffer centered at all times. This makes things quite tidy, but if you have a slow network connection, or simply do not like this un-Emacsism, you can set this variable to `nil` to get the normal Emacs scrolling action. This will also inhibit horizontal re-centering of the summary buffer, which might make it more inconvenient to read extremely long threads.

Choosing Articles

None of the following movement commands understand the numeric prefix, and they all select and display an article.

SPACE

Select the current article, or, if that one's read already, the next unread article (`gnus-summary-next-page`).

G n

n

Go to next unread article (`gnus-summary-next-unread-article`).

G p

p

Go to previous unread article (`gnus-summary-prev-unread-article`).

G N

N

Go to the next article (`gnus-summary-next-article`).

G P

P

Go to the previous article (`gnus-summary-prev-article`).

G C-n

Go to the next article with the same subject (`gnus-summary-next-same-subject`).

G C-p

Go to the previous article with the same subject (`gnus-summary-prev-same-subject`).

G f

.

Go to the first unread article (`gnus-summary-first-unread-article`).

G b

,

Go to the article with the highest score (`gnus-summary-best-unread-article`).

G l

l

Go to the previous article read (`gnus-summary-goto-last-article`).

G p

Pop an article off the summary history and go to this article (`gnus-summary-pop-article`). This command differs from the command above in that you can pop as many previous articles off the history as you like.

Some variables that are relevant for moving and selecting articles:

`gnus-auto-extend-newsgroup`

All the movement commands will try to go to the previous (or next) article, even if that article isn't displayed in the Summary buffer if this variable is non-`nil`. Gnus will then fetch the article from the server and display it in the article buffer.

`gnus-select-article-hook`

This hook is called whenever an article is selected. By default it exposes any threads hidden under the selected article.

`gnus-mark-article-hook`

This hook is called whenever an article is selected. It is intended to be used for marking articles as read. The default value is `gnus-summary-mark-read-and-unread-as-read`, and will change the mark of almost any article you read to `gnus-unread-mark`. The only articles not affected by this function are ticked, dormant, and expirable articles. If you'd instead like to just have unread articles marked as read, you can use `gnus-summary-mark-unread-as-read` instead. It will leave marks like `gnus-low-score-mark`, `gnus-del-mark` (and so on) alone.

Scrolling the Article

SPACE

Pressing SPACE will scroll the current article forward one page, or, if you have come to the end of the current article, will choose the next article (`gnus-summary-next-page`).

DEL

Scroll the current article back one page (`gnus-summary-prev-page`).

RET

Scroll the current article one line forward (`gnus-summary-scroll-up`).

A g

g

(Re)fetch the current article (`gnus-summary-show-article`). If given a prefix, fetch the current article, but don't run any of the article treatment functions. This will give you a "raw" article, just the way it came from the server.

A <

<

Scroll to the beginning of the article (`gnus-summary-beginning-of-article`).

A >

>

Scroll to the end of the article (`gnus-summary-end-of-article`).

A s

Perform an isearch in the article buffer (`gnus-summary-isearch-article`).

Reply, Followup and Post

Summary Mail Commands

Commands for composing a mail message:

S r

r

Mail a reply to the author of the current article (`gnus-summary-reply`).

S R

R

Mail a reply to the author of the current article and include the original message (`gnus-summary-reply-with-original`). This command uses the process/prefix convention.

S o m

Forward the current article to some other person (`gnus-summary-mail-forward`).

S o p

Forward the current article to a newsgroup (`gnus-summary-post-forward`).

S m

m

Send a mail to some other person (`gnus-summary-mail-other-window`).

S D b

If you have sent a mail, but the mail was bounced back to you for some reason (wrong address, transient failure), you can use this command to resend that bounced mail (`gnus-summary-resend-bounced-mail`). You will be popped into a mail buffer where you can edit the headers before sending the mail off again. If you give a prefix to this command, and the bounced mail is a reply to some other mail, Gnus will try to fetch that mail and display it for easy perusal of its headers. This might very well fail, though.

S D r

Not to be confused with the previous command, `gnus-summary-resend-message` will prompt you for an address to send the current message off to, and then send it to that place. The headers of the message won't be altered--but lots of headers that say `Resent-To`, `Resent-From` and so on will be added. This means that you actually send a mail to someone that has a `To` header that (probably) points to yourself. This will confuse people. So, natcherly you'll only do that if you're really eVIl.

This command is mainly used if you have several accounts and want to ship a mail to a different account of yours. (If you're both `root` and `postmaster` and get a mail for `postmaster` to the `root` account, you may want to resend it to `postmaster`. Ordnung muss sein!

S O m

Digest the current series and forward the result using mail (`gnus-uu-digest-mail-forward`). This command uses the process/prefix convention (see section Process/Prefix).

S O p

Digest the current series and forward the result to a newsgroup (`gnus-uu-digest-mail-forward`).

Summary Post Commands

Commands for posting an article:

S p

a

Post an article to the current group (`gnus-summary-post-news`).

S f

f

Post a followup to the current article (`gnus-summary-followup`).

S F

F

Post a followup to the current article and include the original message (`gnus-summary-followup-with-original`). This command uses the process/prefix convention.

S u

Uuencode a file, split it into parts, and post it as a series (`gnus-uu-post-news`). (see section Uuencoding and Posting).

Canceling Articles

Have you ever written something, and then decided that you really, really, really wish you hadn't posted that?

Well, you can't cancel mail, but you can cancel posts.

Find the article you wish to cancel (you can only cancel your own articles, so don't try any funny stuff). Then press C or S c (`gnus-summary-cancel-article`). Your article will be canceled--machines all over the world will be deleting your article.

Be aware, however, that not all sites honor cancels, so your article may live on here and there, while most sites will delete the article in question.

If you discover that you have made some mistakes and want to do some corrections, you can post a superseding article that will replace your original article.

Go to the original article and press S s (`gnus-summary-supersede-article`). You will be put in a buffer where you can edit the article all you want before sending it off the usual way.

The same goes for superseding as for canceling, only more so: Some sites do not honor superseding. On those sites, it will appear that you have posted almost the same article twice.

If you have just posted the article, and change your mind right away, there is a trick you can use to cancel/supersede the article without waiting for the article to appear on your site first. You simply return to the post buffer (which is called `*post-buf*`). There you will find the article you just posted, with all the headers intact. Change the `Message-ID` header to a `Cancel` or `Supersedes` header by substituting one of those words for `Message-ID`. Then just press C-c C-c to send the article as you would do normally. The previous article will be canceled/superseded.

Just remember, kids: There is no 'c' in 'supersede'.

Marking Articles

There are several marks you can set on an article.

You have marks that decide the readedness (whoo, neato-keano neologism ohoy!) of the article. Alphabetic marks generally mean read, while non-alphabetic characters generally mean unread.

In addition, you also have marks that do not affect readedness.

Unread Articles

The following marks mark articles as unread, in one form or other.

`!'

Ticked articles are articles that will remain visible always. If you see an article that you find interesting, or you want to put off reading it, or replying to it, until sometime later, you'd typically tick it. However, articles can be expired, so if you want to keep an article forever, you'll have to save it. Ticked articles have a `!' (`gnus-ticked-mark`) in the first column.

`?'

A dormant article is marked with a `?' (`gnus-dormant-mark`), and will only appear in the summary buffer if there are followups to it.

`SPACE'

An unread article is marked with a `SPACE' (`gnus-unread-mark`). These are articles that haven't been read at all yet.

Read Articles

All the following marks mark articles as read.

`r'

Articles that are marked as read. They have a `r' (`gnus-del-mark`) in the first column. These are articles that the user has marked as read more or less manually.

`R'

Articles that are actually read are marked with `R' (`gnus-read-mark`).

`O'

Articles that were marked as read in previous sessions are now old and marked with `O' (`gnus-ancient-mark`).

`K'

Marked as killed (`gnus-killed-mark`).

`X'

Marked as killed by kill files (`gnus-kill-file-mark`).

`Y'

Marked as read by having a too low score (`gnus-low-score-mark`).

`C'

Marked as read by a catchup (`gnus-catchup-mark`).

`G'

Canceled article (`gnus-canceled-mark`)

`F'

SOUPed article (`gnus-souped-mark`).

`Q'

Sparsely reffed article (`gnus-sparse-mark`).

All these marks just mean that the article is marked as read, really. They are interpreted differently by the adaptive scoring scheme, however.

One more special mark, though:

`E'

You can also mark articles as expirable (or have them marked as such automatically). That doesn't make much sense in normal groups, because a user does not control the expiring of news articles, but in mail groups, for instance, articles that are marked as expirable can be deleted by Gnus at any time. Expirable articles are marked with `E' (`gnus-expirable-mark`).

Other Marks

There are some marks that have nothing to do with whether the article is read or not.

- You can set a bookmark in the current article. Say you are reading a long thesis on cats' urinary tracts, and have to go home for dinner before you've finished reading the thesis. You can then set a bookmark in the article, and Gnus will jump to this bookmark the next time it encounters the article.
- All articles that you have replied to or made a followup to (i.e., have answered) will be marked with an `A' in the second column (`gnus-replied-mark`).
- Articles that are stored in the article cache will be marked with an `*' in the second column (`gnus-cached-mark`).
- Articles that are "saved" (in some manner or other; not necessarily religiously) are marked with an `S' in the second column (`gnus-saved-mark`.
- It the `%e' spec is used, the presence of threads or not will be marked with `gnus-not-empty-thread-mark` and `gnus-empty-thread-mark` in the third column, respectively.
- Finally we have the process mark (`gnus-process-mark`. A variety of commands react to the presence of the process mark. For instance, X u (`gnus-uu-decode-uu`) will uudecode and view all articles that have been marked with the process mark. Articles marked with the process mark have a `#' in the second column.

You might have noticed that most of these "non-readedness" marks appear in the second column by default. So if you have a cached, saved, replied article that you have process-marked, what will that look like?

Nothing much. The precedence rules go as follows: process -> cache -> replied -> saved. So if the article is in the cache and is replied, you'll only see the cache mark and not the replied mark.

Setting Marks

All the marking commands understand the numeric prefix.

M t

!

Tick the current article (`gnus-summary-tick-article-forward`).

M ?

?

Mark the current article as dormant (`gnus-summary-mark-as-dormant`).

M d

d

Mark the current article as read (`gnus-summary-mark-as-read-forward`).

M k

k

Mark all articles that have the same subject as the current one as read, and then select the next unread article (`gnus-summary-kill-same-subject-and-select`).

M K

C-k

Mark all articles that have the same subject as the current one as read (`gnus-summary-kill-same-subject`).

M C

Mark all unread articles in the group as read (`gnus-summary-catchup`).

M C-c

Mark all articles in the group as read--even the ticked and dormant articles (`gnus-summary-catchup-all`).

M H

Catchup the current group to point (`gnus-summary-catchup-to-here`).

C-w

Mark all articles between point and mark as read (`gnus-summary-mark-region-as-read`).

M V k

Kill all articles with scores below the default score (or below the numeric prefix) (`gnus-summary-kill-below`).

M c

M-u

Clear all readedness-marks from the current article (`gnus-summary-clear-mark-forward`).

M e

E

Mark the current article as expirable (`gnus-summary-mark-as-expirable`).

M b

Set a bookmark in the current article (`gnus-summary-set-bookmark`).

M B

Remove the bookmark from the current article (`gnus-summary-remove-bookmark`).

M V c

Clear all marks from articles with scores over the default score (or over the numeric prefix) (`gnus-summary-clear-above`).

M V u

Tick all articles with scores over the default score (or over the numeric prefix) (`gnus-summary-tick-above`).

M V m

Prompt for a mark, and mark all articles with scores over the default score (or over the numeric prefix) with this mark (`gnus-summary-clear-above`).

The `gnus-summary-goto-unread` variable controls what action should be taken after setting a mark. If non-`nil`, point will move to the next/previous unread article. If `nil`, point will just move one line up or down. As a special case, if this variable is `never`, all the marking commands as well as other commands (like SPACE) will move to the next article, whether it is unread or not. The default is `t`.

Setting Process Marks

M P p

#

Mark the current article with the process mark (`gnus-summary-mark-as-processable`).

M P u

M-#

Remove the process mark, if any, from the current article (`gnus-summary-unmark-as-processable`).

M P U

Remove the process mark from all articles (`gnus-summary-unmark-all-processable`).

M P R

Mark articles by a regular expression (`gnus-uu-mark-by-regexp`).

M P r

Mark articles in region (`gnus-uu-mark-region`).

M P t

Mark all articles in the current (sub)thread (`gnus-uu-mark-thread`).

M P T

Unmark all articles in the current (sub)thread (`gnus-uu-unmark-thread`).

M P v

Mark all articles that have a score above the prefix argument (`gnus-uu-mark-over`).

M P s

Mark all articles in the current series (`gnus-uu-mark-series`).

M P S

Mark all series that have already had some articles marked (`gnus-uu-mark-sparse`).

M P a

Mark all articles in series order (`gnus-uu-mark-series`).

M P b

Mark all articles in the buffer in the order they appear (`gnus-uu-mark-buffer`).

Limiting

It can be convenient to limit the summary buffer to just show some subset of the articles currently in the group. The effect most limit commands have is to remove a few (or many) articles from the summary buffer.

/ /

/ s

Limit the summary buffer to articles that match some subject (`gnus-summary-limit-to-subject`).

/ a

Limit the summary buffer to articles that match some author (`gnus-summary-limit-to-author`).

/ u

x

Limit the summary buffer to articles that are not marked as read (`gnus-summary-limit-to-unread`). If given a prefix, limit the buffer to articles that are strictly unread. This means that ticked and dormant articles will also be excluded.

/ m

Ask for a mark and then limit to all articles that have not been marked with that mark (`gnus-summary-limit-to-marks`).

/ n

Limit the summary buffer to the current article (`gnus-summary-limit-to-articles`). Uses the process/prefix convention (see section Process/Prefix).

/ w

Pop the previous limit off the stack and restore it (`gnus-summary-pop-limit`). If given a prefix, pop all limits off the stack.

/ v

Limit the summary buffer to articles that have a score at or above some score (`gnus-summary-limit-to-score`).

/ E

M S

Display all expunged articles (`gnus-summary-limit-include-expunged`).

/ D

Display all dormant articles (`gnus-summary-limit-include-dormant`).

/ d

Hide all dormant articles (`gnus-summary-limit-exclude-dormant`).

/ c

Hide all dormant articles that have no children (`gnus-summary-limit-exclude-childless-dormant`).

/ C

Mark all excluded unread articles as read (`gnus-summary-limit-mark-excluded-as-read`). If given a prefix, also mark excluded ticked and dormant articles as read.

Threading

Gnus threads articles by default. To thread is to put replies to articles directly after the articles they reply to--in a hierarchical fashion.

Customizing Threading

`gnus-show-threads`

If this variable is `nil`, no threading will be done, and all of the rest of the variables here will have no effect. Turning threading off will speed group selection up a bit, but it is sure to make reading slower and more awkward.

`gnus-fetch-old-headers`

If non-`nil`, Gnus will attempt to build old threads by fetching more old headers--headers to articles that are marked as read. If you would like to display as few summary lines as possible, but still connect as many loose threads as possible, you should set this variable to `some` or a number. If you set it to a number, no more than that number of extra old headers will be fetched. In either case, fetching old headers only works if the backend you are using carries overview files--this would normally be `nntp`, `nnspool` and `nnml`. Also remember that if the root of the thread has been expired by the server, there's not much Gnus can do about that.

`gnus-build-sparse-threads`

Fetching old headers can be slow. A low-rent similar effect can be gotten by setting this variable to `some`. Gnus will then look at the complete `References` headers of all articles and try to string articles that belong in the same thread together. This will leave gaps in the threading display where Gnus guesses that an article is missing from the thread. (These gaps appear like normal summary lines. If you select a gap, Gnus will try to fetch the article in question.) If this variable is `t`, Gnus will display all these "gaps" without regard for whether they are useful for completing the thread or not. Finally, if this variable is `more`, Gnus won't cut off sparse leaf nodes that don't lead anywhere. This variable is `nil` by default.

`gnus-summary-gather-subject-limit`

Loose threads are gathered by comparing subjects of articles. If this variable is `nil`, Gnus requires an exact match between the subjects of the loose threads before gathering them into one big super-thread. This might be too strict a requirement, what with the presence of stupid newsreaders that chop off long subjects lines. If you think so, set this variable to, say, 20 to require that only the first 20 characters of the subjects have to match. If you set this variable to a really low number, you'll find that Gnus will gather everything in sight into one thread, which isn't very helpful.

If you set this variable to the special value `fuzzy`, Gnus will use a fuzzy string comparison algorithm on the subjects.

`gnus-simplify-subject-fuzzy-regexp`

This can either be a regular expression or list of regular expressions that match strings that will be

removed from subjects if fuzzy subject simplification is used.

`gnus-simplify-ignored-prefixes`

If you set `gnus-summary-gather-subject-limit` to something as low as 10, you might consider setting this variable to something sensible:

```
(setq gnus-simplify-ignored-prefixes
      (concat
       "\\`\\[?\\("
       (mapconcat 'identity
                  '("looking"
                    "wanted" "followup" "summary\\( of\\)?"
                    "help" "query" "problem" "question"
                    "answer" "reference" "announce"
                    "How can I" "How to" "Comparison of"
                    ;; ...
                    )
                  "\\|")
       "\\)\\s *\\("
       (mapconcat 'identity
                  '("for" "for reference" "with" "about")
                  "\\|")
       "\\)?\\]?:?[ \t]*"))
```

All words that match this regexp will be removed before comparing two subjects.

`gnus-summary-gather-exclude-subject`

Since loose thread gathering is done on subjects only, that might lead to many false hits, especially with certain common subjects like `' and `(none)'. To make the situation slightly better, you can use the regexp `gnus-summary-gather-exclude-subject` to say what subjects should be excluded from the gathering process. The default is `^ *$\\|^(none)$'.

`gnus-summary-thread-gathering-function`

Gnus gathers threads by looking at `Subject` headers. This means that totally unrelated articles may end up in the same "thread", which is confusing. An alternate approach is to look at all the `Message-ID`s in all the `References` headers to find matches. This will ensure that no gathered threads ever includes unrelated articles, but it's also means that people who have posted with broken newsreaders won't be gathered properly. The choice is yours--plague or cholera:

`gnus-gather-threads-by-subject`

This function is the default gathering function and looks at `Subject`s exclusively.

`gnus-gather-threads-by-references`

This function looks at `References` headers exclusively.

If you want to test gathering by `References`, you could say something like:

```
(setq gnus-summary-thread-gathering-function
      'gnus-gather-threads-by-references)
```

- gnus-summary-make-false-root If non-`nil`, Gnus will gather all loose subtrees into one big tree and

create a dummy root at the top. (Wait a minute. Root at the top? Yup.) Loose subtrees occur when the real root has expired, or you've read or killed the root in a previous session.

When there is no real root of a thread, Gnus will have to fudge something. This variable says what fudging method Gnus should use. There are four possible values:

`adopt`

Gnus will make the first of the orphaned articles the parent. This parent will adopt all the other articles. The adopted articles will be marked as such by pointy brackets (`<>') instead of the standard square brackets (`[]'). This is the default method.

`dummy`

Gnus will create a dummy summary line that will pretend to be the parent. This dummy line does not correspond to any real article, so selecting it will just select the first real article after the dummy article. `gnus-summary-dummy-line-format` is used to specify the format of the dummy roots. It accepts only one format spec: `S', which is the subject of the article. See section Formatting Variables.

`empty`

Gnus won't actually make any article the parent, but simply leave the subject field of all orphans except the first empty. (Actually, it will use `gnus-summary-same-subject` as the subject (see section Summary Buffer Format).)

`none`

Don't make any article parent at all. Just gather the threads and display them after one another.

`nil`

Don't gather loose threads.

- gnus-thread-hide-subtree If non-`nil`, all threads will be hidden when the summary buffer is generated.
- gnus-thread-hide-killed if you kill a thread and this variable is non-`nil`, the subtree will be hidden.
- gnus-thread-ignore-subject Sometimes somebody changes the subject in the middle of a thread. If this variable is non-`nil`, the subject change is ignored. If it is `nil`, which is the default, a change in the subject will result in a new thread.
- gnus-thread-indent-level This is a number that says how much each sub-thread should be indented. The default is `4`.

Thread Commands

T k

M-C-k

Mark all articles in the current sub-thread as read (`gnus-summary-kill-thread`). If the prefix argument is positive, remove all marks instead. If the prefix argument is negative, tick articles instead.

T l

M-C-l

Lower the score of the current thread (`gnus-summary-lower-thread`).

T i

Increase the score of the current thread (`gnus-summary-raise-thread`).

T #

Set the process mark on the current thread (`gnus-uu-mark-thread`).

T M-#

Remove the process mark from the current thread (`gnus-uu-unmark-thread`).

T T

Toggle threading (`gnus-summary-toggle-threads`).

T s

Expose the thread hidden under the current article, if any (`gnus-summary-show-thread`).

T h

Hide the current (sub)thread (`gnus-summary-hide-thread`).

T S

Expose all hidden threads (`gnus-summary-show-all-threads`).

T H

Hide all threads (`gnus-summary-hide-all-threads`).

T t

Re-thread the thread the current article is part of (`gnus-summary-rethread-current`). This works even when the summary buffer is otherwise unthreaded.

T ^

Make the current article the child of the marked (or previous) article (`gnus-summary-reparent-thread`.

The following commands are thread movement commands. They all understand the numeric prefix.

T n

Go to the next thread (`gnus-summary-next-thread`).

T p

Go to the previous thread (`gnus-summary-prev-thread`).

T d

Descend the thread (`gnus-summary-down-thread`).

T u

Ascend the thread (`gnus-summary-up-thread`).

T o

Go to the top of the thread (`gnus-summary-top-thread`).

If you ignore subject while threading, you'll naturally end up with threads that have several different subjects in them. If you then issue a command like `T k' (`gnus-summary-kill-thread`) you might not wish to kill the entire thread, but just those parts of the thread that have the same subject as the current article. If you like this idea, you can fiddle with `gnus-thread-operation-ignore-subject`. If is non-`nil` (which it is by default), subjects will be ignored when doing thread commands. If this variable is `nil`, articles in the same thread with different subjects will not be included in the operation in question. If this variable is `fuzzy`, only articles that have subjects that are fuzzily equal will be included.

Sorting

If you are using a threaded summary display, you can sort the threads by setting `gnus-thread-sort-functions`, which is a list of functions. By default, sorting is done on article numbers. Ready-made sorting predicate functions include `gnus-thread-sort-by-number`, `gnus-thread-sort-by-author`, `gnus-thread-sort-by-subject`, `gnus-thread-sort-by-date`, `gnus-thread-sort-by-score`, and `gnus-thread-sort-by-total-score`.

Each function takes two threads and return non-`nil` if the first thread should be sorted before the other. Note that sorting really is normally done by looking only at the roots of each thread. If you use more than one function, the primary sort key should be the last function in the list. You should probably always include `gnus-thread-sort-by-number` in the list of sorting functions--preferably first. This will ensure that threads that are equal with respect to the other sort criteria will be displayed in ascending article order.

If you would like to sort by score, then by subject, and finally by number, you could do something like:

```
(setq gnus-thread-sort-functions
      '(gnus-thread-sort-by-number
        gnus-thread-sort-by-subject
        gnus-thread-sort-by-score))
```

The threads that have highest score will be displayed first in the summary buffer. When threads have the same score, they will be sorted alphabetically. The threads that have the same score and the same subject will be sorted by number, which is (normally) the sequence in which the articles arrived.

If you want to sort by score and then reverse arrival order, you could say something like:

```
(setq gnus-thread-sort-functions
      '((lambda (t1 t2)
          (not (gnus-thread-sort-by-number t1 t2)))
        gnus-thread-sort-by-score))
```

The function in the `gnus-thread-score-function` variable (default `+`) is used for calculating the total score of a thread. Useful functions might be `max`, `min`, or squared means, or whatever tickles your fancy.

If you are using an unthreaded display for some strange reason or other, you have to fiddle with the `gnus-article-sort-functions` variable. It is very similar to the `gnus-thread-sort-functions`, except that is uses slightly different functions for article comparison. Available sorting predicate functions are `gnus-article-sort-by-number`, `gnus-article-sort-by-author`, `gnus-article-sort-by-subject`, `gnus-article-sort-by-date`, and `gnus-article-sort-by-score`.

If you want to sort an unthreaded summary display by subject, you could say something like:

```
(setq gnus-article-sort-functions
      '(gnus-article-sort-by-number
```


```
gnus-article-sort-by-subject))
```

Asynchronous Article Fetching

If you read your news from an NNTP server that's far away, the network latencies may make reading articles a chore. You have to wait for a while after pressing n to go to the next article before the article appears. Why can't Gnus just go ahead and fetch the article while you are reading the previous one? Why not, indeed.

First, some caveats. There are some pitfalls to using asynchronous article fetching, especially the way Gnus does it.

Let's say you are reading article 1, which is short, and article 2 is quite long, and you are not interested in reading that. Gnus does not know this, so it goes ahead and fetches article 2. You decide to read article 3, but since Gnus is in the process of fetching article 2, the connection is blocked.

To avoid these situations, Gnus will open two (count 'em two) connections to the server. Some people may think this isn't a very nice thing to do, but I don't see any real alternatives. Setting up that extra connection takes some time, so Gnus startup will be slower.

Gnus will fetch more articles than you will read. This will mean that the link between your machine and the NNTP server will become more loaded than if you didn't use article pre-fetch. The server itself will also become more loaded--both with the extra article requests, and the extra connection.

Ok, so now you know that you shouldn't really use this thing... unless you really want to.

Here's how: Set `gnus-asynchronous` to `t`. The rest should happen automatically.

You can control how many articles that are to be pre-fetched by setting `nntp-async-number`. This is five by default, which means that when you read an article in the group, `nntp` will pre-fetch the next five articles. If this variable is `t`, `nntp` will pre-fetch all the articles that it can without bound. If it is `nil`, no pre-fetching will be made.

You may wish to create some sort of scheme for choosing which articles that `nntp` should consider as candidates for pre-fetching. For instance, you may wish to pre-fetch all articles with high scores, and not pre-fetch low-scored articles. You can do that by setting the `gnus-asynchronous-article-function`, which will be called with an alist where the keys are the article numbers. Your function should return an alist where the articles you are not interested in have been removed. You could also do sorting on article score and the like.

Article Caching

If you have an *extremely* slow NNTP connection, you may consider turning article caching on. Each article will then be stored locally under your home directory. As you may surmise, this could potentially use *huge* amounts of disk space, as well as eat up all your inodes so fast it will make your head swim. In vodka.

Used carefully, though, it could be just an easier way to save articles.

To turn caching on, set `gnus-use-cache` to `t`. By default, all articles that are ticked or marked as dormant will then be copied over to your local cache (`gnus-cache-directory`). Whether this cache is

flat or hierarchal is controlled by the `gnus-use-long-file-name` variable, as usual.

When re-select a ticked or dormant article, it will be fetched from the cache instead of from the server. As articles in your cache will never expire, this might serve as a method of saving articles while still keeping them where they belong. Just mark all articles you want to save as dormant, and don't worry.

When an article is marked as read, is it removed from the cache.

The entering/removal of articles from the cache is controlled by the `gnus-cache-enter-articles` and `gnus-cache-remove-articles` variables. Both are lists of symbols. The first is `(ticked dormant)` by default, meaning that ticked and dormant articles will be put in the cache. The latter is `(read)` by default, meaning that articles that are marked as read are removed from the cache. Possibly symbols in these two lists are `ticked`, `dormant`, `unread` and `read`.

So where does the massive article-fetching and storing come into the picture? The `gnus-jog-cache` command will go through all subscribed newsgroups, request all unread articles, and store them in the cache. You should only ever, ever ever ever, use this command if 1) your connection to the NNTP server is really, really, really slow and 2) you have a really, really, really huge disk. Seriously.

It is likely that you do not want caching on some groups. For instance, if your `nnml` mail is located under your home directory, it makes no sense to cache it somewhere else under your home directory. Unless you feel that it's neat to use twice as much space. To limit the caching, you could set the `gnus-uncacheable-groups` regexp to `^nnml', for instance. This variable is `nil` by default.

The cache stores information on what articles it contains in its active file (`gnus-cache-active-file`). If this file (or any other parts of the cache) becomes all messed up for some reason or other, Gnus offers two functions that will try to set things right. M-x gnus-cache-generate-nov-databases will (re)build all the NOV files, and gnus-cache-generate-active will (re)generate the active file.

Persistent Articles

Closely related to article caching, we have persistent articles. In fact, it's just a different way of looking at caching, and much more useful in my opinion.

Say you're reading a newsgroup, and you happen on to some valuable gem that you want to keep and treasure forever. You'd normally just save it (using one of the many saving commands) in some file. The problem with that is that it's just, well, yucky. Ideally you'd prefer just having the article remain in the group where you found it forever; untouched by the expiry going on at the news server.

This is what a persistent article is--an article that just won't be deleted. It's implemented using the normal cache functions, but you use two explicit commands for managing persistent articles:

*
: Make the current article persistent (`gnus-cache-enter-article`).

M-*
: Remove the current article from the persistent articles (`gnus-cache-remove-article`). This will normally delete the article.

Both these commands understand the process/prefix convention.

To avoid having all ticked articles (and stuff) entered into the cache, you should set `gnus-use-cache passive` if you're just interested in persistent articles:

```
(setq gnus-use-cache 'passive)
```

Article Backlog

If you have a slow connection, but the idea of using caching seems unappealing to you (and it is, really), you can help the situation some by switching on the backlog. This is where Gnus will buffer already read articles so that it doesn't have to re-fetch articles you've already read. This only helps if you are in the habit of re-selecting articles you've recently read, of course. If you never do that, turning the backlog on will slow Gnus down a little bit, and increase memory usage some.

If you set `gnus-keep-backlog` to a number n, Gnus will store at most n old articles in a buffer for later re-fetching. If this variable is non-`nil` and is not a number, Gnus will store *all* read articles, which means that your Emacs will grow without bound before exploding and taking your machine down with you. I put that in there just to keep y'all on your toes.

This variable is `nil` by default.

Saving Articles

Gnus can save articles in a number of ways. Below is the documentation for saving articles in a fairly straight-forward fashion (i.e., little processing of the article is done before it is saved). For a different approach (uudecoding, unsharing) you should use `gnus-uu` (see section Decoding Articles).

If `gnus-save-all-headers` is non-`nil`, Gnus will not delete unwanted headers before saving the article.

If the preceding variable is `nil`, all headers that match the `gnus-saved-headers` regexp will be kept, while the rest will be deleted before saving.

O o

o

Save the current article using the default article saver (`gnus-summary-save-article`).

O m

Save the current article in mail format (`gnus-summary-save-article-mail`).

O r

Save the current article in rmail format (`gnus-summary-save-article-rmail`).

O f

Save the current article in plain file format (`gnus-summary-save-article-file`).

O b

Save the current article body in plain file format (`gnus-summary-save-article-body-file`).

O h

Save the current article in mh folder format (`gnus-summary-save-article-folder`).

O v

Save the current article in a VM folder (`gnus-summary-save-article-vm`).

O p

Save the current article in a pipe. Uhm, like, what I mean is--Pipe the current article to a process (`gnus-summary-pipe-output`).

All these commands use the process/prefix convention (see section Process/Prefix). If you save bunches of articles using these functions, you might get tired of being prompted for files to save each and every article in. The prompting action is controlled by the `gnus-prompt-before-saving` variable, which is `always` by default, giving you that excessive prompting action you know and loathe. If you set this variable to `t` instead, you'll be prompted just once for each series of articles you save. If you like to really have Gnus do all your thinking for you, you can even set this variable to `nil`, which means that you will never be prompted for files to save articles in. Gnus will simply save all the articles in the default files.

You can customize the `gnus-default-article-saver` variable to make Gnus do what you want it to. You can use any of the four ready-made functions below, or you can create your own.

`gnus-summary-save-in-rmail`

This is the default format, babyl. Uses the function in the `gnus-rmail-save-name` variable to get a file name to save the article in. The default is `gnus-plain-save-name`.

`gnus-summary-save-in-mail`

Save in a Unix mail (mbox) file. Uses the function in the `gnus-mail-save-name` variable to get a file name to save the article in. The default is `gnus-plain-save-name`.

`gnus-summary-save-in-file`

Append the article straight to an ordinary file. Uses the function in the `gnus-file-save-name` variable to get a file name to save the article in. The default is `gnus-numeric-save-name`.

`gnus-summary-save-body-in-file`

Append the article body to an ordinary file. Uses the function in the `gnus-file-save-name` variable to get a file name to save the article in. The default is `gnus-numeric-save-name`.

`gnus-summary-save-in-folder`

Save the article to an MH folder using `rcvstore` from the MH library. Uses the function in the `gnus-folder-save-name` variable to get a file name to save the article in. The default is `gnus-folder-save-name`, but you can also use `gnus-Folder-save-name`. The former creates capitalized names, and the latter does not.

`gnus-summary-save-in-vm`

Save the article in a VM folder. You have to have the VM mail reader to use this setting.

All of these functions, except for the last one, will save the article in the `gnus-article-save-directory`, which is initialized from the `SAVEDIR` environment variable. This is `` `~/News/' `` by default.

As you can see above, the functions use different functions to find a suitable name of a file to save the article in. Below is a list of available functions that generate names:

`gnus-Numeric-save-name`

Generates file names that look like `~/News/Alt.andrea-dworkin/45'.

`gnus-numeric-save-name`

Generates file names that look like `~/News/alt.andrea-dworkin/45'.

`gnus-Plain-save-name`

Generates file names that look like `~/News/Alt.andrea-dworkin'.

`gnus-plain-save-name`

Generates file names that look like `~/News/alt.andrea-dworkin'.

You can have Gnus suggest where to save articles by plonking a regexp into the `gnus-split-methods` alist. For instance, if you would like to save articles related to Gnus in the file `gnus-stuff', and articles related to VM in `vm-stuff`, you could set this variable to something like:

```
(("^Subject:.*gnus\\|^Newsgroups:.*gnus" "gnus-stuff")
 ("^Subject:.*vm\\|^Xref:.*vm" "vm-stuff")
 (my-choosing-function "../other-dir/my-stuff")
 ((equal gnus-newsgroup-name "mail.misc") "mail-stuff"))
```

We see that this is a list where each element is a list that has two elements--the match and the file. The match can either be a string (in which case it is used as a regexp to match on the article head); it can be a symbol (which will be called as a function with the group name as a parameter); or it can be a list (which will be `eval`ed). If any of these actions have a non-`nil` result, the file will be used as a default prompt. In addition, the result of the operation itself will be used if the function or form called returns a string or a list of strings.

You basically end up with a list of file names that might be used when saving the current article. (All "matches" will be used.) You will then be prompted for what you really want to use as a name, with file name completion over the results from applying this variable.

This variable is `((gnus-article-archive-name))` by default, which means that Gnus will look at the articles it saves for an `Archive-name` line and use that as a suggestion for the file name.

Finally, you have the `gnus-use-long-file-name` variable. If it is `nil`, all the preceding functions will replace all periods (`.') in the group names with slashes (`/')---which means that the functions will generate hierarchies of directories instead of having all the files in the toplevel directory (`~/News/alt/andrea-dworkin' instead of `~/News/alt.andrea-dworkin'.) This variable is `t` by default on most systems. However, for historical reasons, this is `nil` on Xenix and usg-unix-v machines by default.

This function also affects kill and score file names. If this variable is a list, and the list contains the element `not-score`, long file names will not be used for score files, if it contains the element `not-save`, long file names will not be used for saving, and if it contains the element `not-kill`, long file names will not be used for kill files.

If you'd like to save articles in a hierarchy that looks something like a spool, you could

```
(setq gnus-use-long-file-name '(not-save)) ; to get a hierarchy
(setq gnus-default-article-save 'gnus-summary-save-in-file) ; no encoding
```

Then just save with o. You'd then read this hierarchy with ephemeral `nneething` groups---G D in the

group buffer, and the toplevel directory as the argument (`~/News/'). Then just walk around to the groups/directories with nneething.

Decoding Articles

Sometime users post articles (or series of articles) that have been encoded in some way or other. Gnus can decode them for you.

All these functions use the process/prefix convention (see section Process/Prefix) for finding out what articles to work on, with the extension that a "single article" means "a single series". Gnus can find out by itself what articles belong to a series, decode all the articles and unpack/view/save the resulting file(s).

Gnus guesses what articles are in the series according to the following simplish rule: The subjects must be (nearly) identical, except for the last two numbers of the line. (Spaces are largely ignored, however.)

For example: If you choose a subject called `cat.gif (2/3)', Gnus will find all the articles that match the regexp `^cat.gif ([0-9]+/[0-9]+).*$'.

Subjects that are nonstandard, like `cat.gif (2/3) Part 6 of a series', will not be properly recognized by any of the automatic viewing commands, and you have to mark the articles manually with #.

Uuencoded Articles

X u
: Uudecodes the current series (gnus-uu-decode-uu).

X U
: Uudecodes and saves the current series (gnus-uu-decode-uu-and-save).

X v u
: Uudecodes and views the current series (gnus-uu-decode-uu-view).

X v U
: Uudecodes, views and saves the current series (gnus-uu-decode-uu-and-save-view).

Remember that these all react to the presence of articles marked with the process mark. If, for instance, you'd like to decode and save an entire newsgroup, you'd typically do M P a (gnus-uu-mark-all) and then X U (gnus-uu-decode-uu-and-save).

All this is very much different from how gnus-uu worked with GNUS 4.1, where you had explicit keystrokes for everything under the sun. This version of gnus-uu generally assumes that you mark articles in some way (see section Setting Process Marks) and then press X u.

Note: When trying to decode articles that have names matching gnus-uu-notify-files, which is hard-coded to `[Cc][Ii][Nn][Dd][Yy][0-9]+.\\(gif\\|jpg\\)', gnus-uu will automatically post an article on `comp.unix.wizards' saying that you have just viewed the file in question. This feature can't be turned off.

Shared Articles

X s

Unshars the current series (`gnus-uu-decode-unshar`).

X S

Unshars and saves the current series (`gnus-uu-decode-unshar-and-save`).

X v s

Unshars and views the current series (`gnus-uu-decode-unshar-view`).

X v S

Unshars, views and saves the current series (`gnus-uu-decode-unshar-and-save-view`).

PostScript Files

X p

Unpack the current PostScript series (`gnus-uu-decode-postscript`).

X P

Unpack and save the current PostScript series (`gnus-uu-decode-postscript-and-save`).

X v p

View the current PostScript series (`gnus-uu-decode-postscript-view`).

X v P

View and save the current PostScript series (`gnus-uu-decode-postscript-and-save-view`).

Decoding Variables

Adjective, not verb.

Rule Variables

Gnus uses rule variables to decide how to view a file. All these variables are on the form

```
(list '(regexp1 command2)
      '(regexp2 command2)
      ...)
```

`gnus-uu-user-view-rules`

This variable is consulted first when viewing files. If you wish to use, for instance, `sox` to convert an `.au' sound file, you could say something like:

```
(setq gnus-uu-user-view-rules
  (list '(\"\\\\.au$\" \"sox %s -t .aiff > /dev/audio\")))
```

`gnus-uu-user-view-rules-end`

This variable is consulted if Gnus couldn't make any matches from the user and default view rules.

`gnus-uu-user-archive-rules`

This variable can be used to say what commands should be used to unpack archives.

Other Decode Variables

`gnus-uu-grabbed-file-functions`

All functions in this list will be called right each file has been successfully decoded--so that you can move or view files right away, and don't have to wait for all files to be decoded before you can do anything. Ready-made functions you can put in this list are:

`gnus-uu-grab-view`

View the file.

`gnus-uu-grab-move`

Move the file (if you're using a saving function.)

- gnus-uu-ignore-files-by-name Files with name matching this regular expression won't be viewed.
- gnus-uu-ignore-files-by-type Files with a MIME type matching this variable won't be viewed. Note that Gnus tries to guess what type the file is based on the name. `gnus-uu` is not a MIME package (yet), so this is slightly kludgey.
- gnus-uu-tmp-dir Where `gnus-uu` does its work.
- gnus-uu-do-not-unpack-archives Non-`nil` means that `gnus-uu` won't peek inside archives looking for files to display.
- gnus-uu-view-and-save Non-`nil` means that the user will always be asked to save a file after viewing it.
- gnus-uu-ignore-default-view-rules Non-`nil` means that `gnus-uu` will ignore the default viewing rules.
- gnus-uu-ignore-default-archive-rules Non-`nil` means that `gnus-uu` will ignore the default archive unpacking commands.
- gnus-uu-kill-carriage-return Non-`nil` means that `gnus-uu` will strip all carriage returns from articles.
- gnus-uu-unmark-articles-not-decoded Non-`nil` means that `gnus-uu` will mark articles that were unsuccessfully decoded as unread.
- gnus-uu-correct-stripped-uucode Non-`nil` means that `gnus-uu` will *try* to fix uuencoded files that have had trailing spaces deleted.
- gnus-uu-view-with-metamail Non-`nil` means that `gnus-uu` will ignore the viewing commands defined by the rule variables and just fudge a MIME content type based on the file name. The result will be fed to `metamail` for viewing.
- gnus-uu-save-in-digest Non-`nil` means that `gnus-uu`, when asked to save without decoding, will save in digests. If this variable is `nil`, `gnus-uu` will just save everything in a file without any embellishments. The digesting almost conforms to RFC1153--no easy way to specify any meaningful volume and issue numbers were found, so I simply dropped them.

Uuencoding and Posting

`gnus-uu-post-include-before-composing`

Non-`nil` means that `gnus-uu` will ask for a file to encode before you compose the article. If this variable is `t`, you can either include an encoded file with C-c C-i or have one included for you when

you post the article.

`gnus-uu-post-length`

Maximum length of an article. The encoded file will be split into how many articles it takes to post the entire file.

`gnus-uu-post-threaded`

Non-`nil` means that `gnus-uu` will post the encoded file in a thread. This may not be smart, as no other decoder I have seen are able to follow threads when collecting uuencoded articles. (Well, I have seen one package that does that---`gnus-uu`, but somehow, I don't think that counts...) Default is `nil`.

`gnus-uu-post-separate-description`

Non-`nil` means that the description will be posted in a separate article. The first article will typically be numbered (0/x). If this variable is `nil`, the description the user enters will be included at the beginning of the first article, which will be numbered (1/x). Default is `t`.

Viewing Files

After decoding, if the file is some sort of archive, Gnus will attempt to unpack the archive and see if any of the files in the archive can be viewed. For instance, if you have a gzipped tar file `` `pics.tar.gz' `` containing the files `` `pic1.jpg' `` and `` `pic2.gif' ``, Gnus will uncompress and de-tar the main file, and then view the two pictures. This unpacking process is recursive, so if the archive contains archives of archives, it'll all be unpacked.

Finally, Gnus will normally insert a pseudo-article for each extracted file into the summary buffer. If you go to these "articles", you will be prompted for a command to run (usually Gnus will make a suggestion), and then the command will be run.

If `gnus-view-pseudo-asynchronously` is `nil`, Emacs will wait until the viewing is done before proceeding.

If `gnus-view-pseudos` is `automatic`, Gnus will not insert the pseudo-articles into the summary buffer, but view them immediately. If this variable is `not-confirm`, the user won't even be asked for a confirmation before viewing is done.

If `gnus-view-pseudos-separately` is non-`nil`, one pseudo-article will be created for each file to be viewed. If `nil`, all files that use the same viewing command will be given as a list of parameters to that command.

If `gnus-insert-pseudo-articles` is non-`nil`, insert pseudo-articles when decoding. It is `t` by default.

So; there you are, reading your *pseudo-articles* in your *virtual newsgroup* from the *virtual server*; and you think: Why isn't anything real anymore? How did we get here?

Article Treatment

Reading through this huge manual, you may have quite forgotten that the object of newsreaders are to actually, like, read what people have written. Reading articles. Unfortunately, people are quite bad at writing, so there are tons of functions and variables to make reading these articles easier.

Article Highlighting

Not only do you want your article buffer to look like fruit salad, but you want it to look like technicolor fruit salad.

W H a

Highlight the current article (`gnus-article-highlight`).

W H h

Highlight the headers (`gnus-article-highlight-headers`). The highlighting will be done according to the `gnus-header-face-alist` variable, which is a list where each element has the form (regexp name content). regexp is a regular expression for matching the header, name is the face used for highlighting the header name and content is the face for highlighting the header value. The first match made will be used. Note that regexp shouldn't have `^' prepended--Gnus will add one.

W H c

Highlight cited text (`gnus-article-highlight-citation`).

Some variables to customize the citation highlights:

`gnus-cite-parse-max-size`

If the article size if bigger than this variable (which is 25000 by default), no citation highlighting will be performed.

`gnus-cite-prefix-regexp`

Regexp matching the longest possible citation prefix on a line.

`gnus-cite-max-prefix`

Maximum possible length for a citation prefix (default 20).

`gnus-cite-face-list`

List of faces used for highlighting citations. When there are citations from multiple articles in the same message, Gnus will try to give each citation from each article its own face. This should make it easier to see who wrote what.

`gnus-supercite-regexp`

Regexp matching normal Supercite attribution lines.

`gnus-supercite-secondary-regexp`

Regexp matching mangled Supercite attribution lines.

`gnus-cite-minimum-match-count`

Minimum number of identical prefixes we have to see before we believe that it's a citation.

`gnus-cite-attribution-prefix`

Regexp matching the beginning of an attribution line.

`gnus-cite-attribution-suffix`

Regexp matching the end of an attribution line.

`gnus-cite-attribution-face`

Face used for attribution lines. It is merged with the face for the cited text belonging to the attribution.

- W H s Highlight the signature (`gnus-article-highlight-signature`). Everything after `gnus-signature-separator` in an article will be considered a signature and will be highlighted with `gnus-signature-face`, which is `italic` by default.

Article Hiding

Or rather, hiding certain things in each article. There usually is much too much cruft in most articles.

W W a

Do maximum hiding on the summary buffer (gnus-article-hide).

W W h

Hide headers (`gnus-article-hide-headers`). See section Hiding Headers.

W W b

Hide headers that aren't particularly interesting (`gnus-article-hide-boring-headers`). See section Hiding Headers.

W W s

Hide signature (`gnus-article-hide-signature`).

W W p

Hide PGP signatures (`gnus-article-hide-pgp`).

W W c

Hide citation (`gnus-article-hide-citation`). Some variables for customizing the hiding:

`gnus-cite-hide-percentage`

If the cited text is of a bigger percentage than this variable (default 50), hide the cited text.

`gnus-cite-hide-absolute`

The cited text must be have at least this length (default 10) before it is hidden.

`gnus-cited-text-button-line-format`

Gnus adds buttons show where the cited text has been hidden, and to allow toggle hiding the text. The format of the variable is specified by this format-like variable. These specs are legal:

`b'

Start point of the hidden text.

`e'

End point of the hidden text.

`l'

Length of the hidden text.

- gnus-cited-lines-visible The number of lines at the beginning of the cited text to leave shown.

- W W C Hide cited text in articles that aren't roots (`gnus-article-hide-citation-in-followups`). This isn't very useful as an interactive command, but might be a handy function to stick in `gnus-article-display-hook` (see section Customizing Articles).

All these "hiding" commands are toggles, but if you give a negative prefix to these commands, they will

show what they have previously hidden. If you give a positive prefix, they will always hide.

Also see section Article Highlighting for further variables for citation customization.

`gnus-signature-limit` provides a limit to what is considered a signature. If it is a number, no signature may not be longer (in characters) than that number. If it is a function, the function will be called without any parameters, and if it returns `nil`, there is no signature in the buffer. If it is a string, it will be used as a regexp. If it matches, the text in question is not a signature.

Article Washing

We call this "article washing" for a really good reason. Namely, the A key was taken, so we had to use the W key instead.

Washing is defined by us as "changing something from something to something else", but normally results in something looking better. Cleaner, perhaps.

W l

Remove page breaks from the current article (`gnus-summary-stop-page-breaking`).

W r

Do a Caesar rotate (rot13) on the article buffer (`gnus-summary-caesar-message`).

W t

Toggle whether to display all headers in the article buffer (`gnus-summary-toggle-header`).

W v

Toggle whether to display all headers in the article buffer permanently (`gnus-summary-verbose-header`).

W m

Toggle whether to run the article through MIME before displaying (`gnus-summary-toggle-mime`).

W o

Treat overstrike (`gnus-article-treat-overstrike`).

W w

Do word wrap (`gnus-article-fill-cited-article`).

W c

Remove CR (`gnus-article-remove-cr`).

W L

Remove all blank lines at the end of the article (`gnus-article-remove-trailing-blank-lines`).

W q

Treat quoted-printable (`gnus-article-de-quoted-unreadable`).

W f

Look for and display any X-Face headers (`gnus-article-display-x-face`). The command executed by this function is given by the `gnus-article-x-face-command` variable. If this

variable is a string, this string will be executed in a sub-shell. If it is a function, this function will be called with the face as the argument. If the `gnus-article-x-face-too-ugly` (which is a regexp) matches the `From` header, the face will not be shown. The default action under Emacs is to fork off an `xv` to view the face; under XEmacs the default action is to display the face before the `From` header. (It's nicer if XEmacs has been compiled with X-Face support -- that will make display somewhat faster. If there's no native X-Face support, Gnus will try to convert the `X-Face` header using external programs from the `pbmplus` package and friends.) If you want to have this function in the display hook, it should probably come last.

W b

Add clickable buttons to the article (`gnus-article-add-buttons`).

W B

Add clickable buttons to the article headers (`gnus-article-add-buttons-to-head`).

Article Buttons

People often include references to other stuff in articles, and it would be nice if Gnus could just fetch whatever it is that people talk about with the minimum of fuzz.

Gnus adds buttons to certain standard references by default: Well-formed URLs, mail addresses and Message-IDs. This is controlled by two variables, one that handles article bodies and one that handles article heads:

`gnus-button-alist`

This is an alist where each entry has this form:

```
(REGEXP BUTTON-PAR USE-P FUNCTION DATA-PAR)
```

regexp

All text that match this regular expression will be considered an external reference. Here's a typical regexp that match embedded URLs: `<URL:\\([^\n\r>]*\\)>'.

button-par

Gnus has to know which parts of the match is to be highlighted. This is a number that says what sub-expression of the regexp that is to be highlighted. If you want it all highlighted, you use `0` here.

use-p

This form will be `eval`ed, and if the result is non-`nil`, this is considered a match. This is useful if you want extra sifting to avoid false matches.

function

This function will be called when you click on this button.

data-par

As with button-par, this is a sub-expression number, but this one says which part of the match is to be sent as data to function.

So the full entry for buttonizing URLs is then


```
("<URL:\\([^\n\r>]*\\)>" 0 t gnus-button-url 1)
```

- gnus-header-button-alist This is just like the other alist, except that it is applied to the article head only, and that each entry has an additional element that is used to say what headers to apply the buttonize coding to:

```
(HEADER REGEXP BUTTON-PAR USE-P FUNCTION DATA-PAR)
```

header is a regular expression.

- gnus-button-url-regexp A regular expression that matches embedded URLs. It is used in the default values of the variables above.
- gnus-article-button-face Face used on bottons.
- gnus-article-mouse-face Face is used when the mouse cursor is over a button.

Article Date

The date is most likely generated in some obscure timezone you've never heard of, so it's quite nice to be able to find out what the time was when the article was sent.

W T u
: Display the date in UT (aka. GMT, aka ZULU) (`gnus-article-date-ut`).

W T l
: Display the date in the local timezone (`gnus-article-date-local`).

W T e
: Say how much time has (e)lapsed between the article was posted and now (`gnus-article-date-lapsed`).

W T o
: Display the original date (`gnus-article-date-original`). This can be useful if you normally use some other conversion function and is worried that it might be doing something totally wrong. Say, claiming that the article was posted in 1854. Although something like that is *totally* impossible. Don't you trust me? *titter*

Summary Sorting

You can have the summary buffer sorted in various ways, even though I can't really see why you'd want that.

C-c C-s C-n
: Sort by article number (`gnus-summary-sort-by-number`).

C-c C-s C-a
: Sort by author (`gnus-summary-sort-by-author`).

C-c C-s C-s
: Sort by subject (`gnus-summary-sort-by-subject`).

C-c C-s C-d
: Sort by date (`gnus-summary-sort-by-date`).

C-c C-s C-i

Sort by score (`gnus-summary-sort-by-score`).

These functions will work both when you use threading and when you don't use threading. In the latter case, all summary lines will be sorted, line by line. In the former case, sorting will be done on a root-by-root basis, which might not be what you were looking for. To toggle whether to use threading, type T T (see section Thread Commands).

Finding the Parent

If you'd like to read the parent of the current article, and it is not displayed in the summary buffer, you might still be able to. That is, if the current group is fetched by NNTP, the parent hasn't expired and the `References` in the current article are not mangled, you can just press ^ or A r (`gnus-summary-refer-parent-article`). If everything goes well, you'll get the parent. If the parent is already displayed in the summary buffer, point will just move to this article.

You can have Gnus fetch all articles mentioned in the `References` header of the article by pushing A R (`gnus-summary-refer-references`).

You can also ask the NNTP server for an arbitrary article, no matter what group it belongs to. M-^ (`gnus-summary-refer-article`) will ask you for a `Message-ID`, which is one of those long thingies that look something like `<38o6up$6f2@hymir.ifi.uio.no>'. You have to get it all exactly right. No fuzzy searches, I'm afraid.

If the group you are reading is located on a backend that does not support fetching by `Message-ID` very well (like `nnspool`), you can set `gnus-refer-article-method` to an NNTP method. It would, perhaps, be best if the NNTP server you consult is the same as the one that keeps the spool you are reading from updated, but that's not really necessary.

Most of the mail backends support fetching by `Message-ID`, but do not do a particularly excellent job of it. That is, `nnmbox` and `nnbabyl` are able to locate articles from any groups, while `nnml` and `nnfolder` are only able to locate articles that have been posted to the current group. (Anything else would be too time consuming.) `nnmh` does not support this at all.

Alternative Approaches

Different people like to read news using different methods. This being Gnus, we offer a small selection of minor modes for the summary buffers.

Pick and Read

Some newsreaders (like `nn` and, uhm, `nn`) use a two-phased reading interface. The user first marks the articles she wants to read from a summary buffer. Then she starts reading the articles with just an article buffer displayed.

Gnus provides a summary buffer minor mode that allows this---`gnus-pick-mode`. This basically means that a few process mark commands become one-keystroke commands to allow easy marking, and it makes

one additional command for switching to the summary buffer available.

Here are the available keystrokes when using pick mode:

SPACE
: Pick the article (`gnus-summary-mark-as-processable`).

u
: Unpick the article (`gnus-summary-unmark-as-processable`).

U
: Unpick all articles (`gnus-summary-unmark-all-processable`).

t
: Pick the thread (`gnus-uu-mark-thread`).

T
: Unpick the thread (`gnus-uu-unmark-thread`).

r
: Pick the region (`gnus-uu-mark-region`).

R
: Unpick the region (`gnus-uu-unmark-region`).

e
: Pick articles that match a regexp (`gnus-uu-mark-by-regexp`).

E
: Unpick articles that match a regexp (`gnus-uu-unmark-by-regexp`).

b
: Pick the buffer (`gnus-uu-mark-buffer`).

B
: Unpick the buffer (`gnus-uu-unmark-buffer`).

RET
: Start reading the picked articles (`gnus-pick-start-reading`). If given a prefix, mark all unpicked articles as read first. If `gnus-pick-display-summary` is non-`nil`, the summary buffer will still be visible when you are reading.

If this sounds like a good idea to you, you could say:

```
(add-hook 'gnus-summary-mode-hook 'gnus-pick-mode)
```

`gnus-pick-mode-hook` is run in pick minor mode buffers.

Binary Groups

If you spend much time in binary groups, you may grow tired of hitting X u, n, RET all the time. M-x gnus-binary-mode is a minor mode for summary buffers that makes all ordinary Gnus article selection functions uudecode series of articles and display the result instead of just displaying the articles the normal way.

In fact, the only way to see the actual articles if you have turned this mode on is the g command (`gnus-binary-show-article`).

`gnus-binary-mode-hook` is called in binary minor mode buffers.

Tree Display

If you don't like the normal Gnus summary display, you might try setting `gnus-use-trees` to `t`. This will create (by default) an additional tree buffer. You can execute all summary mode commands in the tree buffer.

There are a few variables to customize the tree display, of course:

`gnus-tree-mode-hook`

A hook called in all tree mode buffers.

`gnus-tree-mode-line-format`

A format string for the mode bar in the tree mode buffers. The default is `Gnus: %%b [%A] %Z'. For a list of legal specs, see section Summary Buffer Mode Line.

`gnus-selected-tree-face`

Face used for highlighting the selected article in the tree buffer. The default is `modeline`.

`gnus-tree-line-format`

A format string for the tree nodes. The name is a bit of a misnomer, though--it doesn't define a line, but just the node. The default value is `%(%[%3,3n%]%)', which displays the first three characters of the name of the poster. It is vital that all nodes are of the same length, so you *must* use `%4,4n'-like specifiers.

Legal specs are:

`n'

The name of the poster.

`f'

The `From` header.

`N'

The number of the article.

`['

The opening bracket.

`]'

The closing bracket.

`s'

The subject.

See section Formatting Variables.

Variables related to the display are:

`gnus-tree-brackets`

This is used for differentiating between "real" articles and "sparse" articles. The format is ((real-open . real-close) (sparse-open . sparse-close) (dummy-open . dummy-close)), and the default is `((?[ . ?]) (?( . ?)) (?{ . ?}))`.

`gnus-tree-parent-child-edges`

This is a list that contains the characters used for connecting parent nodes to their children. The default is `(?- ?\\ ?|)`.

- gnus-tree-minimize-window If this variable is non-`nil`, Gnus will try to keep the tree buffer as small as possible to allow more room for the other Gnus windows. If this variable is a number, the tree buffer will never be higher than that number. The default is `t`.
- gnus-generate-tree-function The function that actually generates the thread tree. Two predefined functions are available: `gnus-generate-horizontal-tree` and `gnus-generate-vertical-tree` (which is the default).

Here's and example from a horizontal tree buffer:

```
{***}-(***)-[odd]-[Gun]
     |     \[Jan]
     |     \[odd]-[Eri]
     |     \(***)-[Eri]
     |           \[odd]-[Paa]
     \[Bjo]
     \[Gun]
     \[Gun]-[Jor]
```

Here's the same thread displayed in a vertical tree buffer:

```
{***}
  |--------------------------\-----\-----\
(***)                         [Bjo] [Gun] [Gun]
  |--\-----\-----\                           |
[odd] [Jan] [odd] (***)                    [Jor]
  |           |     |--\
[Gun]       [Eri] [Eri] [odd]
                          |
                        [Paa]
```

Mail Group Commands

Some commands only make sense in mail groups. If these commands are illegal in the current group, they will raise a hell and let you know.

All these commands (except the expiry and edit commands) use the process/prefix convention (see section Process/Prefix).

B e

Expire all expirable articles in the group (`gnus-summary-expire-articles`).

B M-C-e

Expunge all the expirable articles in the group (`gnus-summary-expire-articles-now`). This means that **all** articles that are eligible for expiry in the current group will disappear forever into that big `/dev/null' in the sky.

B DEL

Delete the mail article. This is "delete" as in "delete it from your disk forever and ever, never to return again." Use with caution. (`gnus-summary-delete-article`).

B m

Move the article from one mail group to another (`gnus-summary-move-article`).

B c

Copy the article from one group (mail group or not) to a mail group (`gnus-summary-copy-article`).

B C

Crosspost the current article to some other group (`gnus-summary-crosspost-article`). This will create a new copy of the article in the other group, and the Xref headers of the article will be properly updated.

B i

Import an arbitrary file into the current mail newsgroup (`gnus-summary-import-article`). You will be prompted for a file name, a `From` header and a `Subject` header.

B r

Respool the mail article (`gnus-summary-move-article`).

B w

e

Edit the current article (`gnus-summary-edit-article`). To finish editing and make the changes permanent, type C-c C-c (gnus-summary-edit-article-done).

B q

If you want to re-spool an article, you might be curious as to what group the article will end up in before you do the re-spooling. This command will tell you (`gnus-summary-respool-query`).

If you move (or copy) articles regularly, you might wish to have Gnus suggest where to put the articles. `gnus-move-split-methods` is a variable that uses the same syntax as `gnus-split-methods` (see section Saving Articles). You may customize that variable to create suggestions you find reasonable.

Various Summary Stuff

`gnus-summary-mode-hook`

This hook is called when creating a summary mode buffer.

`gnus-summary-generate-hook`

This is called as the last thing before doing the threading and the generation of the summary buffer. It's quite convenient for customizing the threading variables based on what data the newsgroup has. This

hook is called from the summary buffer after most summary buffer variables has been set.

`gnus-summary-prepare-hook`

Is is called after the summary buffer has been generated. You might use it to, for instance, highlight lines or modify the look of the buffer in some other ungodly manner. I don't care.

Summary Group Information

H f

Try to fetch the FAQ (list of frequently asked questions) for the current group (`gnus-summary-fetch-faq`). Gnus will try to get the FAQ from `gnus-group-faq-directory`, which is usually a directory on a remote machine. This variable can also be a list of directories. In that case, giving a prefix to this command will allow you to choose between the various sites. `ange-ftp` probably will be used for fetching the file.

H d

Give a brief description of the current group (`gnus-summary-describe-group`). If given a prefix, force rereading the description from the server.

H h

Give a very brief description of the most important summary keystrokes (`gnus-summary-describe-briefly`).

H i

Go to the Gnus info node (`gnus-info-find-node`).

Searching for Articles

M-s

Search through all subsequent articles for a regexp (`gnus-summary-search-article-forward`).

M-r

Search through all previous articles for a regexp (`gnus-summary-search-article-backward`).

&

This command will prompt you for a header field, a regular expression to match on this field, and a command to be executed if the match is made (`gnus-summary-execute-command`).

M-&

Perform any operation on all articles that have been marked with the process mark (`gnus-summary-universal-argument`).

Really Various Summary Commands

A D

If the current article is a collection of other articles (for instance, a digest), you might use this command to enter a group based on the that article (`gnus-summary-enter-digest-group`).

Gnus will try to guess what article type is currently displayed unless you give a prefix to this command, which forces a "digest" interpretation. Basically, whenever you see a message that is a collection of other messages on some format, you A D and read these messages in a more convenient fashion.

C-t

Toggle truncation of summary lines (`gnus-summary-toggle-truncation`).

=

Expand the summary buffer window (`gnus-summary-expand-window`). If given a prefix, force an `article` window configuration.

Exiting the Summary Buffer

Exiting from the summary buffer will normally update all info on the group and return you to the group buffer.

Z Z

q

Exit the current group and update all information on the group (`gnus-summary-exit`). `gnus-summary-prepare-exit-hook` is called before doing much of the exiting, and calls `gnus-summary-expire-articles` by default. `gnus-summary-exit-hook` is called after finishing the exiting process.

Z E

Q

Exit the current group without updating any information on the group (`gnus-summary-exit-no-update`).

Z c

c

Mark all unticked articles in the group as read and then exit (`gnus-summary-catchup-and-exit`).

Z C

Mark all articles, even the ticked ones, as read and then exit (`gnus-summary-catchup-all-and-exit`).

Z n

Mark all articles as read and go to the next group (`gnus-summary-catchup-and-goto-next-group`).

Z R

Exit this group, and then enter it again (`gnus-summary-reselect-current-group`). If given a prefix, select all articles, both read and unread.

Z G

M-g

Exit the group, check for new articles in the group, and select the group (`gnus-summary-rescan-group`). If given a prefix, select all articles, both read and unread.

Z N

Exit the group and go to the next group (`gnus-summary-next-group`).

Z P

Exit the group and go to the previous group (`gnus-summary-prev-group`).

`gnus-exit-group-hook` is called when you exit the current group.

If you're in the habit of exiting groups, and then changing your mind about it, you might set `gnus-kill-summary-on-exit` to `nil`. If you do that, Gnus won't kill the summary buffer when you exit it. (Quelle surprise!) Instead it will change the name of the buffer to something like `*Dead Summary ... *' and install a minor mode called `gnus-dead-summary-mode`. Now, if you switch back to this buffer, you'll find that all keys are mapped to a function called `gnus-summary-wake-up-the-dead`. So tapping any keys in a dead summary buffer will result in a live, normal summary buffer.

There will never be more than one dead summary buffer at any one time.

The data on the current group will be updated (which articles you have read, which articles you have replied to, etc.) when you exit the summary buffer. If the `gnus-use-cross-reference` variable is `t` (which is the default), articles that are cross-referenced to this group and are marked as read, will also be marked as read in the other subscribed groups they were cross-posted to. If this variable is neither `nil` nor `t`, the article will be marked as read in both subscribed and unsubscribed groups.

Marking cross-posted articles as read ensures that you'll never have to read the same article more than once. Unless, of course, somebody has posted it to several groups separately. Posting the same article to several groups (not cross-posting) is called spamming, and you are by law required to send nasty-grams to anyone who perpetrates such a heinous crime.

Remember: Cross-posting is kinda ok, but posting the same article separately to several groups is not. Massive cross-posting (aka. velveeta) is to be avoided.

One thing that may cause Gnus to not do the cross-posting thing correctly is if you use an NNTP server that supports XOVER (which is very nice, because it speeds things up considerably) which does not include the `Xref` header in its NOV lines. This is Evil, but all too common, alas, alack. Gnus tries to Do The Right Thing even with XOVER by registering the `Xref` lines of all articles you actually read, but if you kill the articles, or just mark them as read without reading them, Gnus will not get a chance to snoop the `Xref` lines out of these articles, and will be unable to use the cross reference mechanism.

To check whether your NNTP server includes the `Xref` header in its overview files, try `telnet your.nntp.server nntp', `MODE READER' on `inn` servers, and then say `LIST overview.fmt'. This may not work, but if it does, and the last line you get does not read `Xref:full', then you should shout and whine at your news admin until she includes the `Xref` header in the overview files.

If you want Gnus to get the `Xref`s right all the time, you have to set `gnus-nov-is-evil` to `t`, which slows things down considerably.

C'est la vie.

Go to the previous, next section.

Go to the previous, next section.

The Article Buffer

The articles are displayed in the article buffer, of which there is only one. All the summary buffers share the same article buffer unless you tell Gnus otherwise.

Hiding Headers

The top section of each article is the head. (The rest is the body, but you may have guessed that already.)

There is a lot of useful information in the head: the name of the person who wrote the article, the date it was written and the subject of the article. That's well and nice, but there's also lots of information most people do not want to see--what systems the article has passed through before reaching you, the `Message-ID`, the `References`, etc. ad nauseum--and you'll probably want to get rid of some of those lines. If you want to keep all those lines in the article buffer, you can set `gnus-show-all-headers` to `t`.

Gnus provides you with two variables for sifting headers:

`gnus-visible-headers`

> If this variable is non-`nil`, it should be a regular expression that says what headers you wish to keep in the article buffer. All headers that do not match this variable will be hidden.
>
> For instance, if you only want to see the name of the person who wrote the article and the subject, you'd say:
>
> ```
> (setq gnus-visible-headers "^From:\\|^Subject:")
> ```
>
> This variable can also be a list of regexps to match headers that are to remain visible.

`gnus-ignored-headers`

> This variable is the reverse of `gnus-visible-headers`. If this variable is set (and `gnus-visible-headers` is `nil`), it should be a regular expression that matches all lines that you want to hide. All lines that do not match this variable will remain visible.
>
> For instance, if you just want to get rid of the `References` line and the `Xref` line, you might say:
>
> ```
> (setq gnus-ignored-headers "^References:\\|^Xref:")
> ```
>
> This variable can also be a list of regexps to match headers that are to be removed.
>
> Note that if `gnus-visible-headers` is non-`nil`, this variable will have no effect.

Gnus can also sort the headers for you. (It does this by default.) You can control the sorting by setting the `gnus-sorted-header-list` variable. It is a list of regular expressions that says in what order the

headers are to be displayed.

For instance, if you want the name of the author of the article first, and then the subject, you might say something like:

```
(setq gnus-sorted-header-list '("^From:" "^Subject:"))
```

Any headers that are to remain visible, but are not listed in this variable, will be displayed in random order after all the headers that are listed in this variable.

You can hide further boring headers by entering `gnus-article-hide-boring-headers` into `gnus-article-display-hook`. What this function does depends on the `gnus-boring-article-headers` variable. It's a list, but this list doesn't actually contain header names. Instead is lists various boring conditions that Gnus can check and remove from sight.

These conditions are:

`empty`
: Remove all empty headers.

`newsgroups`
: Remove the `Newsgroups` header if it only contains the current group name.

`followup-to`
: Remove the `Followup-To` header if it is identical to the `Newsgroups` header.

`reply-to`
: Remove the `Reply-To` header if it lists the same address as the `From` header.

`date`
: Remove the `Date` header if the article is less than three days old.

To include the four first elements, you could say something like;

```
(setq gnus-boring-article-headers
      '(empty newsgroups followup-to reply-to))
```

This is also the default value for this variable.

Using MIME

Mime is a standard for waving your hands through the air, aimlessly, while people stand around yawning.

MIME, however, is a standard for encoding your articles, aimlessly, while all newsreaders die of fear.

MIME may specify what character set the article uses, the encoding of the characters, and it also makes it possible to embed pictures and other naughty stuff in innocent-looking articles.

Gnus handles MIME by shoving the articles through `gnus-show-mime-method`, which is

`metamail-buffer` by default. Set `gnus-show-mime` to `t` if you want to use MIME all the tim
However, if `gnus-strict-mime` is non-`nil`, the MIME method will only be used if there are M
headers in the article.

It might be best to just use the toggling functions from the summary buffer to avoid getting nasty surprises. (For instance, you enter the group `alt.sing-a-long' and, before you know it, MIME has decoded the sound file in the article and some horrible sing-a-long song comes streaming out out your speakers, and you can't find the volume button, because there isn't one, and people are starting to look at you, and you try to stop the program, but you can't, and you can't find the program to control the volume, and everybody else in the room suddenly decides to look at you disdainfully, and you'll feel rather stupid.)

Any similarity to real events and people is purely coincidental. Ahem.

Customizing Articles

The `gnus-article-display-hook` is called after the article has been inserted into the article buffer. It is meant to handle all treatment of the article before it is displayed.

By default it contains `gnus-article-hide-headers`, `gnus-article-treat-overstrike`, and `gnus-article-maybe-highlight`, but there are thousands, nay millions, of functions you can put in this hook. For an overview of functions see section Article Highlighting, see section Article Hiding, see section Article Washing, see section Article Buttons and see section Article Date.

You can, of course, write your own functions. The functions are called from the article buffer, and you can do anything you like, pretty much. There is no information that you have to keep in the buffer--you can change everything. However, you shouldn't delete any headers. Instead make them invisible if you want to make them go away.

Article Keymap

Most of the keystrokes in the summary buffer can also be used in the article buffer. They should behave as if you typed them in the summary buffer, which means that you don't actually have to have a summary buffer displayed while reading. You can do it all from the article buffer.

A few additional keystrokes are available:

SPACE
: Scroll forwards one page (`gnus-article-next-page`).

DEL
: Scroll backwards one page (`gnus-article-prev-page`).

C-c ^
: If point is in the neighborhood of a `Message-ID` and you press r, Gnus will try to get that article from the server (`gnus-article-refer-article`).

C-c C-m

Send a reply to the address near point (`gnus-article-mail`). If given a prefix, include th mail.

s

Reconfigure the buffers so that the summary buffer becomes visible (`gnus-article-show-summary`).

?

Give a very brief description of the available keystrokes (`gnus-article-describe-briefly`).

TAB

Go to the next button, if any (`gnus-article-next-button`. This only makes sense if you have buttonizing turned on.

M-TAB

Go to the previous button, if any (`gnus-article-prev-button`.

Misc Article

`gnus-single-article-buffer`

If non-`nil`, use the same article buffer for all the groups. (This is the default.) If `nil`, each group will have its own article buffer.

`gnus-article-prepare-hook`

This hook is called right after the article has been inserted into the article buffer. It is mainly intended for functions that do something depending on the contents; it should probably not be used for changing the contents of the article buffer.

`gnus-article-display-hook`

This hook is called as the last thing when displaying an article, and is intended for modifying the contents of the buffer, doing highlights, hiding headers, and the like.

`gnus-article-mode-hook`

Hook called in article mode buffers.

`gnus-article-mode-line-format`

This variable is a format string along the same lines as `gnus-summary-mode-line-format`. It accepts exactly the same format specifications as that variable.

`gnus-break-pages`

Controls whether page breaking is to take place. If this variable is non-`nil`, the articles will be divided into pages whenever a page delimiter appears in the article. If this variable is `nil`, paging will not be done.

`gnus-page-delimiter`

This is the delimiter mentioned above. By default, it is `^L' (form linefeed).

Go to the previous, next section.

Go to the previous, next section.

Composing Messages

All commands for posting and mailing will put you in a message buffer where you can edit the article all you like, before you send the article by pressing C-c C-c. See section 'Top' in The Message Manual. If you are in a foreign news group, and you wish to post the article using the foreign server, you can give a prefix to C-c C-c to make Gnus try to post using the foreign server.

Also see see section Canceling Articles for information on how to remove articles you shouldn't have posted.

Mail

Variables for customizing outgoing mail:

`gnus-uu-digest-headers`

> List of regexps to match headers included in digested messages. The headers will be included in the sequence they are matched.

Post

Variables for composing news articles:

`gnus-sent-message-ids-file`

> Gnus will keep a `Message-ID` history file of all the mails it has sent. If it discovers that it has already sent a mail, it will ask the user whether to re-send the mail. (This is primarily useful when dealing with SOUP packets and the like where one is apt to sent the same packet multiple times.) This variable says what the name of this history file is. It is `` `~/News/Sent-Message-IDs' `` by default. Set this variable to `nil` if you don't want Gnus to keep a history file.

`gnus-sent-message-ids-length`

> This variable says how many `Message-ID`s to keep in the history file. It is 1000 by default.

Posting Server

When you press those magical C-c C-c keys to ship off your latest (extremely intelligent, of course) article, where does it go?

Thank you for asking. I hate you.

It can be quite complicated. Normally, Gnus will use the same native server. However. If your native server doesn't allow posting, just reading, you probably want to use some other server to post your (extremely intelligent and fabulously interesting) articles. You can then set the `gnus-post-method`

to some other method:

```
(setq gnus-post-method '(nnspool ""))
```

Now, if you've done this, and then this server rejects your article, or this server is down, what do you do then? To override this variable you can use a non-zero prefix to the C-c C-c command to force using the "current" server for posting.

If you give a zero prefix (i. e., C-u 0 C-c C-c) to that command, Gnus will prompt you for what method to use for posting.

You can also set `gnus-post-method` to a list of select methods. If that's the case, Gnus will always prompt you for what method to use for posting.

Mail and Post

Here's a list of variables that are relevant to both mailing and posting:

`gnus-mailing-list-groups`

> If your news server offers groups that are really mailing lists that are gatewayed to the NNTP server, you can read those groups without problems, but you can't post/followup to them without some difficulty. One solution is to add a `to-address` to the group parameters (see section Group Parameters). An easier thing to do is set the `gnus-mailing-list-groups` to a regexp that match the groups that really are mailing lists. Then, at least, followups to the mailing lists will work most of the time. Posting to these groups (a) is still a pain, though.

You may want to do spell-checking on messages that you send out. Or, if you don't want to spell-check by hand, you could add automatic spell-checking via the `ispell` package:

```
(add-hook 'message-send-hook 'ispell-message)
```

Archived Messages

Gnus provides a few different methods for storing the mail you send. The default method is to use the archive virtual server to store the mail. If you want to disable this completely, you should set `gnus-message-archive-group` to `nil`.

`gnus-message-archive-method` says what virtual server Gnus is to use to store sent messages. It is `(nnfolder "archive" (nnfolder-directory "~/Mail/archive/"))` by default, but you can use any mail select method (`nnml`, `nnmbox`, etc.). However, `nnfolder` is a quite likeable select method for doing this sort of thing. If you don't like the default directory chosen, you could say something like:

```
(setq gnus-message-archive-method
      '(nnfolder "archive"
```


```
                (nnfolder-inhibit-expiry t)
                (nnfolder-active-file "~/News/sent-mail/active")
                (nnfolder-directory "~/News/sent-mail/")))
```

Gnus will insert `Gcc` headers in all outgoing messages that point to one or more group(s) on that server. Which group to use is determined by the `gnus-message-archive-group` variable.

This variable can be:

- a string Messages will be saved in that group.
- a list of strings Messages will be saved in all those groups.
- an alist of regexps, functions and forms When a key "matches", the result is used.

Let's illustrate:

Just saving to a single group called `MisK':

```
(setq gnus-message-archive-group "MisK")
```

Saving to two groups, `MisK' and `safe':

```
(setq gnus-message-archive-group '("MisK" "safe"))
```

Save to different groups based on what group you are in:

```
(setq gnus-message-archive-group
      '(("^alt" "sent-to-alt")
        ("mail" "sent-to-mail")
        (".*" "sent-to-misc")))
```

More complex stuff:

```
(setq gnus-message-archive-group
      '((if (message-news-p)
            "misc-news"
          "misc-mail")))
```

This is the default.

How about storing all news messages in one file, but storing all mail messages in one file per month:

```
(setq gnus-message-archive-group
      '((if (message-news-p)
            "misc-news"
          (concat "mail." (format-time-string
                           "%Y-%m" (current-time))))))
```

Now, when you send a message off, it will be stored in the appropriate group. (If you want to disable

storing for just one particular message, you can just remove the `Gcc` header that has been inserted.) The archive group will appear in the group buffer the next time you start Gnus, or the next time you press g in the group buffer. You can enter it and read the articles in it just like you'd read any other group. If the group gets really big and annoying, you can simply rename if (using G r in the group buffer) to something nice -- `misc-mail-september-1995', or whatever. New messages will continue to be stored in the old (now empty) group.

That's the default method of archiving sent mail. Gnus also offers two other variables for the people who don't like the default method. In that case you should set `gnus-message-archive-group` to `nil`; this will disable archiving.

XEmacs 19.13 doesn't have `format-time-string`, so you'll have to use a different value for `gnus-message-archive-group` there.

`gnus-outgoing-message-group`

> All outgoing messages will be put in this group. If you want to store all your outgoing mail and articles in the group `nnml:archive', you set this variable to that value. This variable can also be a list of group names.
>
> If you want to have greater control over what group to put each message in, you can set this variable to a function that checks the current newsgroup name and then returns a suitable group name (or list of names).

Go to the previous, next section.

Go to the previous, next section.

Select Methods

A foreign group is a group that is not read by the usual (or default) means. It could be, for instance, a group from a different NNTP server, it could be a virtual group, or it could be your own personal mail group.

A foreign group (or any group, really) is specified by a name and a select method. To take the latter first, a select method is a list where the first element says what backend to use (eg. `nntp`, `nnspool`, `nnml`) and the second element is the server name. There may be additional elements in the select method, where the value may have special meaning for the backend in question.

One could say that a select method defines a virtual server---so we do just that (see section The Server Buffer).

The name of the group is the name the backend will recognize the group as.

For instance, the group `soc.motss' on the NNTP server `some.where.edu' will have the name `soc.motss' and select method `(nntp "some.where.edu")`. Gnus will call this group, in all circumstances, `nntp+some.where.edu:soc.motss', even though the `nntp` backend just knows this group as `soc.motss'.

The different methods all have their peculiarities, of course.

The Server Buffer

Traditionally, a server is a machine or a piece of software that one connects to, and then requests information from. Gnus does not connect directly to any real servers, but does all transactions through one backend or other. But that's just putting one layer more between the actual media and Gnus, so we might just as well say that each backend represents a virtual server.

For instance, the `nntp` backend may be used to connect to several different actual NNTP servers, or, perhaps, to many different ports on the same actual NNTP server. You tell Gnus which backend to use, and what parameters to set by specifying a select method.

These select methods specifications can sometimes become quite complicated--say, for instance, that you want to read from the NNTP server `news.funet.fi' on port number `13`, which hangs if queried for NOV headers and has a buggy select. Ahem. Anyways, if you had to specify that for each group that used this server, that would be too much work, so Gnus offers a way of naming select methods, which is what you do in the server buffer.

To enter the server buffer, user the ^ (`gnus-group-enter-server-mode`) command in the group buffer.

`gnus-server-mode-hook` is run when creating the server buffer.

Server Buffer Format

You can change the look of the server buffer lines by changing the `gnus-server-line-format` variable. This is a `format`-like variable, with some simple extensions:

`h'
: How the news is fetched--the backend name.

`n'
: The name of this server.

`w'
: Where the news is to be fetched from--the address.

`s'
: The opened/closed/denied status of the server.

The mode line can also be customized by using the `gnus-server-mode-line-format` variable. The following specs are understood:

`S'
: Server name.

`M'
: Server method.

Also see section Formatting Variables.

Server Commands

a
: Add a new server (`gnus-server-add-server`).

e
: Edit a server (`gnus-server-edit-server`).

SPACE
: Browse the current server (`gnus-server-read-server`).

q
: Return to the group buffer (`gnus-server-exit`).

k
: Kill the current server (`gnus-server-kill-server`).

y
: Yank the previously killed server (`gnus-server-yank-server`).

c
: Copy the current server (`gnus-server-copy-server`).

l

List all servers (`gnus-server-list-servers`).

Example Methods

Most select methods are pretty simple and self-explanatory:

```
(nntp "news.funet.fi")
```

Reading directly from the spool is even simpler:

```
(nnspool "")
```

As you can see, the first element in a select method is the name of the backend, and the second is the address, or name, if you will.

After these two elements, there may be a arbitrary number of (variable form) pairs.

To go back to the first example--imagine that you want to read from port `15` from that machine. This is what the select method should look like then:

```
(nntp "news.funet.fi" (nntp-port-number 15))
```

You should read the documentation to each backend to find out what variables are relevant, but here's an `nnmh` example.

`nnmh` is a mail backend that reads a spool-like structure. Say you have two structures that you wish to access: One is your private mail spool, and the other is a public one. Here's the possible spec for you private mail:

```
(nnmh "private" (nnmh-directory "~/private/mail/"))
```

(This server is then called `private', but you may have guessed that.)

Here's the method for a public spool:

```
(nnmh "public"
      (nnmh-directory "/usr/information/spool/")
      (nnmh-get-new-mail nil))
```

Creating a Virtual Server

If you're saving lots of articles in the cache by using persistent articles, you may want to create a virtual server to read the cache.

First you need to add a new server. The a command does that. It would probably be best to use `nnspool` to read the cache. You could also use `nnml` or `nnmh`, though.

Type a nnspool RET cache RET.

You should now have a brand new `nnspool` virtual server called `cache'. You now need to edit it to the right definitions. Type e to edit the server. You'll be entered into a buffer that will contain the following:

```
(nnspool "cache")
```

Change that to:

```
(nnspool "cache"
         (nnspool-spool-directory "~/News/cache/")
         (nnspool-nov-directory "~/News/cache/")
         (nnspool-active-file "~/News/cache/active"))
```

Type C-c C-c to return to the server buffer. If you now press RET over this virtual server, you should be entered into a browse buffer, and you should be able to enter any of the groups displayed.

Servers and Methods

Wherever you would normally use a select method (eg. `gnus-secondary-select-method`, in the group select method, when browsing a foreign server) you can use a virtual server name instead. This could potentially save lots of typing. And it's nice all over.

Unavailable Servers

If a server seems to be unreachable, Gnus will mark that server as `denied`. That means that any subsequent attempt to make contact with that server will just be ignored. "It can't be opened," Gnus will tell you, without making the least effort to see whether that is actually the case or not.

That might seem quite naughty, but it does make sense most of the time. Let's say you have 10 groups subscribed to the server `nepholococcygia.com'. This server is located somewhere quite far away from you, the machine is quite, so it takes 1 minute just to find out that it refuses connection from you today. If Gnus were to attempt to do that 10 times, you'd be quite annoyed, so Gnus won't attempt to do that. Once it has gotten a single "connection refused", it will regard that server as "down".

So, what happens if the machine was only feeling unwell temporarily? How do you test to see whether the machine has come up again?

You jump to the server buffer (see section The Server Buffer) and poke it with the following commands:

O
: Try to establish connection to the server on the current line (`gnus-server-open-server`).

C
: Close the connection (if any) to the server (`gnus-server-close-server`).

D
: Mark the current server as unreachable (`gnus-server-deny-server`).

R

Remove all marks to whether Gnus was denied connection from all servers (`gnus-server-remove-denials`).

Getting News

A newsreader is normally used for reading news. Gnus currently provides only two methods of getting news -- it can read from an NNTP server, or it can read from a local spool.

NNTP

Subscribing to a foreign group from an NNTP server is rather easy. You just specify `nntp` as method and the address of the NNTP server as the, uhm, address.

If the NNTP server is located at a non-standard port, setting the third element of the select method to this port number should allow you to connect to the right port. You'll have to edit the group info for that (see section Foreign Groups).

The name of the foreign group can be the same as a native group. In fact, you can subscribe to the same group from as many different servers you feel like. There will be no name collisions.

The following variables can be used to create a virtual `nntp` server:

`nntp-server-opened-hook`

`nntp-server-opened-hook` is run after a connection has been made. It can be used to send commands to the NNTP server after it has been contacted. By default is sends the command `MODE READER` to the server with the `nntp-send-mode-reader` function. Another popular function is `nntp-send-authinfo`, which will prompt you for an NNTP password and stuff.

`nntp-server-action-alist`

This is an list of regexps to match on server types and actions to be taken when matches are made. For instance, if you want Gnus to beep every time you connect to innd, you could say something like:

```
(setq nntp-server-action-alist
      '(("innd" (ding))))
```

You probably don't want to do that, though.

The default value is

```
'(("nntpd 1\\.5\\.11t"
   (remove-hook 'nntp-server-opened-hook nntp-send-mode-reader)))
```

This ensures that Gnus doesn't send the `MODE READER` command to nntpd 1.5.11t, since that command chokes that server, I've been told.

`nntp-maximum-request`

If the NNTP server doesn't support NOV headers, this backend will collect headers by sending a

series of `head` commands. To speed things up, the backend sends lots of these commands without waiting for reply, and then reads all the replies. This is controlled by the `nntp-maximum-request` variable, and is 400 by default. If your network is buggy, you should set this to 1.

`nntp-connection-timeout`

If you have lots of foreign `nntp` groups that you connect to regularly, you're sure to have problems with NNTP servers not responding properly, or being too loaded to reply within reasonable time. This is can lead to awkward problems, which can be helped somewhat by setting `nntp-connection-timeout`. This is an integer that says how many seconds the `nntp` backend should wait for a connection before giving up. If it is `nil`, which is the default, no timeouts are done.

`nntp-command-timeout`

If you're running Gnus on a machine that has a dynamically assigned address, Gnus may become confused. If the address of your machine changes after connecting to the NNTP server, Gnus will simply sit waiting forever for replies from the server. To help with this unfortunate problem, you can set this command to a number. Gnus will then, if it sits waiting longer than that number of seconds for a reply from the server, shut down the connection, start a new one, and resend the command. This should hopefully be transparent to the user. A likely number is 30 seconds.

`nntp-retry-on-break`

If this variable is non-`nil`, you can also C-g if Gnus hangs. This will have much the same effect as the command timeout described above.

`nntp-server-hook`

This hook is run as the last step when connecting to an NNTP server.

`nntp-open-server-function`

This function is used to connect to the remote system. Two pre-made functions are `nntp-open-network-stream`, which is the default, and simply connects to some port or other on the remote system. The other is `nntp-open-rlogin`, which does an rlogin on the remote system, and then does a telnet to the NNTP server available there.

`nntp-rlogin-parameters`

If you use `nntp-open-rlogin` as the `nntp-open-server-function`, this list will be used as the parameter list given to `rsh`.

`nntp-end-of-line`

String to use as end-of-line markers when talking to the NNTP server. This is `\r\n' by default, but should be `\n' when using `rlogin` to talk to the server.

`nntp-rlogin-user-name`

User name on the remote system when using the `rlogin` connect function.

`nntp-address`

The address of the remote system running the NNTP server.

`nntp-port-number`

Port number to connect to when using the `nntp-open-network-stream` connect function.

`nntp-buggy-select`

Set this to non-`nil` if your select routine is buggy.

`nntp-nov-is-evil`

If the NNTP server does not support NOV, you could set this variable to `t`, but `nntp` usually checks whether NOV can be used automatically.

`nntp-xover-commands`

List of strings that are used as commands to fetch NOV lines from a server. The default value of this variable is `("XOVER" "XOVERVIEW")`.

`nntp-nov-gap`

`nntp` normally sends just one big request for NOV lines to the server. The server responds with one huge list of lines. However, if you have read articles 2-5000 in the group, and only want to read article 1 and 5001, that means that `nntp` will fetch 4999 NOV lines that you do not want, and will not use. This variable says how big a gap between two consecutive articles is allowed to be before the `XOVER` request is split into several request. Note that if your network is fast, setting this variable to a really small number means that fetching will probably be slower. If this variable is `nil`, `nntp` will never split requests.

`nntp-prepare-server-hook`

A hook run before attempting to connect to an NNTP server.

`nntp-async-number`

How many articles should be pre-fetched when in asynchronous mode. If this variable is `t`, `nntp` will pre-fetch all the articles that it can without bound. If it is `nil`, no pre-fetching will be made.

`nntp-warn-about-losing-connection`

If this variable is non-`nil`, some noise will be made when a server closes connection.

News Spool

Subscribing to a foreign group from the local spool is extremely easy, and might be useful, for instance, to speed up reading groups like `alt.binaries.pictures.furniture'.

Anyways, you just specify `nnspool` as the method and `' (or anything else) as the address.

If you have access to a local spool, you should probably use that as the native select method (see section Finding the News). It is normally faster than using an `nntp` select method, but might not be. It depends. You just have to try to find out what's best at your site.

`nnspool-inews-program`

Program used to post an article.

`nnspool-inews-switches`

Parameters given to the inews program when posting an article.

`nnspool-spool-directory`

Where `nnspool` looks for the articles. This is normally `` `/usr/spool/news/' ``.

`nnspool-nov-directory`

Where `nnspool` will look for NOV files. This is normally

`/usr/spool/news/over.view/'.

`nnspool-lib-dir`
: Where the news lib dir is (`/usr/lib/news/' by default).

`nnspool-active-file`
: The path of the active file.

`nnspool-newsgroups-file`
: The path of the group descriptions file.

`nnspool-history-file`
: The path of the news history file.

`nnspool-active-times-file`
: The path of the active date file.

`nnspool-nov-is-evil`
: If non-`nil`, `nnspool` won't try to use any NOV files that it finds.

`nnspool-sift-nov-with-sed`
: If non-`nil`, which is the default, use `sed` to get the relevant portion from the overview file. If nil, `nnspool` will load the entire file into a buffer and process it there.

Getting Mail

Reading mail with a newsreader--isn't that just plain WeIrD? But of course.

Getting Started Reading Mail

It's quite easy to use Gnus to read your new mail. You just plonk the mail backend of your choice into `gnus-secondary-select-methods`, and things will happen automatically.

For instance, if you want to use `nnml` (which is a one file per mail backend), you could put the following in your `.gnus' file:

```
(setq gnus-secondary-select-methods
      '((nnml "private")))
```

Now, the next time you start Gnus, this backend will be queried for new articles, and it will move all the messages in your spool file to its directory, which is `~/Mail/` by default. The new group that will be created (`mail.misc') will be subscribed, and you can read it like any other group.

You will probably want to split the mail into several groups, though:

```
(setq nnmail-split-methods
  '(("junk" "^From:.*Lars Ingebrigtsen")
    ("crazy" "^Subject:.*die\\|^Organization:.*flabby")
    ("other" "")))
```

This will result in three new mail groups being created: `nnml:junk', `nnml:crazy', and `nnml:other'. A the mail that doesn't fit into the first two groups will be placed in the latter group.

This should be sufficient for reading mail with Gnus. You might want to give the other sections in this part of the manual a perusal, though, especially see section Choosing a Mail Backend and see section Expiring Mail.

Splitting Mail

The `nnmail-split-methods` variable says how the incoming mail is to be split into groups.

```
(setq nnmail-split-methods
  '(("mail.junk" "^From:.*Lars Ingebrigtsen")
    ("mail.crazy" "^Subject:.*die\\|^Organization:.*flabby")
    ("mail.other" "")))
```

This variable is a list of lists, where the first element of each of these lists is the name of the mail group (they do not have to be called something beginning with `mail', by the way), and the second element is a regular expression used on the header of each mail to determine if it belongs in this mail group.

The second element can also be a function. In that case, it will be called narrowed to the headers with the first element of the rule as the argument. It should return a non-`nil` value if it thinks that the mail belongs in that group.

The last of these groups should always be a general one, and the regular expression should *always* be `' so that it matches any mails that haven't been matched by any of the other regexps.

If you like to tinker with this yourself, you can set this variable to a function of your choice. This function will be called without any arguments in a buffer narrowed to the headers of an incoming mail message. The function should return a list of groups names that it thinks should carry this mail message.

Note that the mail backends are free to maul the poor, innocent incoming headers all they want to. They all add `Lines` headers; some add `X-Gnus-Group` headers; most rename the Unix mbox `From<SPACE>` line to something else.

The mail backends all support cross-posting. If several regexps match, the mail will be "cross-posted" to all those groups. `nnmail-crosspost` says whether to use this mechanism or not. Note that no articles are crossposted to the general (`') group.

`nnmh` and `nnml` makes crossposts by creating hard links to the crossposted articles. However, not all files systems support hard links. If that's the case for you, set `nnmail-crosspost-link-function` to `copy-file`. (This variable is `add-name-to-file` by default.)

Gnus gives you all the opportunity you could possibly want for shooting yourself in the foot. Let's say you create a group that will contain all the mail you get from your boss. And then you accidentally unsubscribe from the group. Gnus will still put all the mail from your boss in the unsubscribed group, and so, when your boss mails you "Have that report ready by Monday or you're fired!", you'll never see it and, come Tuesday, you'll still believe that you're gainfully employed while you really should be out collecting empty bottles to save up for next month's rent money.

Mail Backend Variables

These variables are (for the most part) pertinent to all the various mail backends.

`nnmail-read-incoming-hook`

The mail backends all call this hook after reading new mail. You can use this hook to notify any mail watch programs, if you want to.

`nnmail-spool-file`

The backends will look for new mail in this file. If this variable is `nil`, the mail backends will never attempt to fetch mail by themselves. If you are using a POP mail server and your name is `larsi', you should set this variable to `po:larsi'. If your name is not `larsi', you should probably modify that slightly, but you may have guessed that already, you smart & handsome devil! You can also set this variable to `pop`, and Gnus will try to figure out the POP mail string by itself. In any case, Gnus will call `movemail` which will contact the POP server named in the `MAILHOST` environment variable. If the POP server needs a password, you can either set `nnmail-pop-password-required` to `t` and be prompted for the password, or set `nnmail-pop-password` to the password itself.

When you use a mail backend, Gnus will slurp all your mail from your inbox and plonk it down in your home directory. Gnus doesn't move any mail if you're not using a mail backend--you have to do a lot of magic invocations first. At the time when you have finished drawing the pentagram, lightened the candles, and sacrificed the goat, you really shouldn't be too surprised when Gnus moves your mail.

`nnmail-use-procmail`

If non-`nil`, the mail backends will look in `nnmail-procmail-directory` for incoming mail. All the files in that directory that have names ending in `nnmail-procmail-suffix` will be considered incoming mailboxes, and will be searched for new mail.

`nnmail-crash-box`

When the mail backends read a spool file, it is first moved to this file, which is `` `~/.gnus-crash-box' `` by default. If this file already exists, it will always be read (and incorporated) before any other spool files.

`nnmail-prepare-incoming-hook`

This is run in a buffer that holds all the new incoming mail, and can be used for, well, anything, really.

`nnmail-pre-get-new-mail-hook`

`nnmail-post-get-new-mail-hook`

These are two useful hooks executed when treating new incoming mail---`nnmail-pre-get-new-mail-hook` (is called just before starting to handle the new mail) and `nnmail-post-get-new-mail-hook` (is called when the mail handling is done). Here's and example of using these two hooks to change the default file modes the new mail files get:

```
(add-hook 'gnus-pre-get-new-mail-hook
```


```
            (lambda () (set-default-file-modes 511)))

    (add-hook 'gnus-post-get-new-mail-hook
            (lambda () (set-default-file-modes 551)))
```

`nnmail-tmp-directory`

This variable says where to move the incoming mail to while processing it. This is usually done in the same directory that the mail backend inhabits (i.e., `` `~/Mail/' ``), but if this variable is non-`nil`, it will be used instead.

`nnmail-movemail-program`

This program is executed to move mail from the user's inbox to her home directory. The default is `movemail'.

`nnmail-delete-incoming`

If non-`nil`, the mail backends will delete the temporary incoming file after splitting mail into the proper groups. This is `nil` by default for reasons of security.

`nnmail-use-long-file-names`

If non-`nil`, the mail backends will use long file and directory names. Groups like `mail.misc' will end up in directories like `` `mail.misc/' ``. If it is `nil`, the same group will end up in `` `mail/misc/' ``.

`nnmail-delete-file-function`

Function called to delete files. It is `delete-file` by default.

Fancy Mail Splitting

If the rather simple, standard method for specifying how to split mail doesn't allow you to do what you want, you can set `nnmail-split-methods` to `nnmail-split-fancy`. Then you can play with the `nnmail-split-fancy` variable.

Let's look at an example value of this variable first:

```
;; Messages from the mailer daemon are not crossposted to any of
;; the ordinary groups.  Warnings are put in a separate group
;; from real errors.
(| ("from" mail (| ("subject" "warn.*" "mail.warning")
                   "mail.misc"))
   ;; Non-error messages are crossposted to all relevant
   ;; groups, but we don't crosspost between the group for the
   ;; (ding) list and the group for other (ding) related mail.
   (& (| (any "ding@ifi\\.uio\\.no" "ding.list")
         ("subject" "ding" "ding.misc"))
      ;; Other mailing lists...
      (any "procmail@informatik\\.rwth-aachen\\.de" "procmail.list")
      (any "SmartList@informatik\\.rwth-aachen\\.de" "SmartList.list")
      ;; People...
```


```
    (any "larsi@ifi\\.uio\\.no" "people.Lars Magne Ingebrigtsen"))
  ;; Unmatched mail goes to the catch all group.
  "misc.misc"))")
```

This variable has the format of a split. A split is a (possibly) recursive structure where each split may contain other splits. Here are the four possible split syntaxes:

GROUP

If the split is a string, that will be taken as a group name.

(FIELD VALUE SPLIT)

If the split is a list, and the first element is a string, then that means that if header FIELD (a regexp) contains VALUE (also a regexp), then store the message as specified by SPLIT.

(| SPLIT...)

If the split is a list, and the first element is `|` (vertical bar), then process each SPLIT until one of them matches. A SPLIT is said to match if it will cause the mail message to be stored in one or more groups.

(& SPLIT...)

If the split is a list, and the first element is &, then process all SPLITs in the list.

In these splits, FIELD must match a complete field name. VALUE must match a complete word according to the fundamental mode syntax table. You can use `.*` in the regexps to match partial field names or words.

FIELD and VALUE can also be lisp symbols, in that case they are expanded as specified by the variable `nnmail-split-abbrev-alist`. This is an alist of cons cells, where the car of the cells contains the key, and the cdr contains a string.

`nnmail-split-fancy-syntax-table` is the syntax table in effect when all this splitting is performed.

Mail and Procmail

Many people use `procmail` (or some other mail filter program or external delivery agent---`slocal`, `elm`, etc) to split incoming mail into groups. If you do that, you should set `nnmail-spool-file` to `procmail` to ensure that the mail backends never ever try to fetch mail by themselves.

This also means that you probably don't want to set `nnmail-split-methods` either, which has some, perhaps, unexpected side effects.

When a mail backend is queried for what groups it carries, it replies with the contents of that variable, along with any groups it has figured out that it carries by other means. None of the backends (except `nnmh`) actually go out to the disk and check what groups actually exist. (It's not trivial to distinguish between what the user thinks is a basis for a newsgroup and what is just a plain old file or directory.)

This means that you have to tell Gnus (and the backends) what groups exist by hand.

Let's take the `nnmh` backend as an example.

The folders are located in `nnmh-directory`, say, `~/Mail/'. There are three folders, `foo', `bar' and `mail.baz'.

Go to the group buffer and type G m. When prompted, answer `foo' for the name and `nnmh' for the method. Repeat twice for the two other groups, `bar' and `mail.baz'. Be sure to include all your mail groups.

That's it. You are now set to read your mail. An active file for this method will be created automatically.

If you use `nnfolder` or any other backend that store more than a single article in each file, you should never have procmail add mails to the file that Gnus sees. Instead, procmail should put all incoming mail in `nnmail-procmail-directory`. To arrive at the file name to put the incoming mail in, append `nnmail-procmail-suffix` to the group name. The mail backends will read the mail from these files.

When Gnus reads a file called `mail.misc.spool', this mail will be put in the `mail.misc`, as one would expect. However, if you want Gnus to split the mail the normal way, you could set `nnmail-resplit-incoming` to `t`.

If you use `procmail` to split things directory into an `nnmh` directory (which you shouldn't do), you should set `nnmail-keep-last-article` to non-`nil` to prevent Gnus from ever expiring the final article in a mail newsgroup. This is quite, quite important.

Incorporating Old Mail

Most people have lots of old mail stored in various file formats. If you have set up Gnus to read mail using one of the spiffy Gnus mail backends, you'll probably wish to have that old mail incorporated into your mail groups.

Doing so can be quite easy.

To take an example: You're reading mail using `nnml` (see section Mail Spool), and have set `nnmail-split-methods` to a satisfactory value (see section Splitting Mail). You have an old Unix mbox file filled with important, but old, mail. You want to move it into your `nnml` groups.

Here's how:

1. Go to the group buffer.
2. Type `G f' and give the path of the mbox file when prompted to create an `nndoc` group from the mbox file (see section Foreign Groups).
3. Type `SPACE' to enter the newly created group.
4. Type `M P b' to process-mark all articles in this group (see section Setting Process Marks).
5. Type `B r' to respool all the process-marked articles, and answer `nnml' when prompted (see section Mail Group Commands).

All the mail messages in the mbox file will now also be spread out over all your `nnml` groups. Try entering them and check whether things have gone without a glitch. If things look ok, you may consider deleting the mbox file, but I wouldn't do that unless I was absolutely sure that all the mail has ended up where it should be.

Respooling is also a handy thing to do if you're switching from one mail backend to another. Just resp all the mail in the old mail groups using the new mail backend.

Expiring Mail

Traditional mail readers have a tendency to remove mail articles when you mark them as read, in some way. Gnus takes a fundamentally different approach to mail reading.

Gnus basically considers mail just to be news that has been received in a rather peculiar manner. It does not think that it has the power to actually change the mail, or delete any mail messages. If you enter a mail group, and mark articles as "read", or kill them in some other fashion, the mail articles will still exist on the system. I repeat: Gnus will not delete your old, read mail. Unless you ask it to, of course.

To make Gnus get rid of your unwanted mail, you have to mark the articles as expirable. This does not mean that the articles will disappear right away, however. In general, a mail article will be deleted from your system if, 1) it is marked as expirable, AND 2) it is more than one week old. If you do not mark an article as expirable, it will remain on your system until hell freezes over. This bears repeating one more time, with some spurious capitalizations: IF you do NOT mark articles as EXPIRABLE, Gnus will NEVER delete those ARTICLES.

You do not have to mark articles as expirable by hand. Groups that match the regular expression `gnus-auto-expirable-newsgroups` will have all articles that you read marked as expirable automatically. All articles that are marked as expirable have an `E' in the first column in the summary buffer.

Let's say you subscribe to a couple of mailing lists, and you want the articles you have read to disappear after a while:

```
(setq gnus-auto-expirable-newsgroups
      "mail.nonsense-list\\|mail.nice-list")
```

Another way to have auto-expiry happen is to have the element `auto-expire` in the group parameters of the group.

The `nnmail-expiry-wait` variable supplies the default time an expirable article has to live. The default is seven days.

Gnus also supplies a function that lets you fine-tune how long articles are to live, based on what group they are in. Let's say you want to have one month expiry period in the `mail.private' group, a one day expiry period in the `mail.junk' group, and a six day expiry period everywhere else:

```
(setq nnmail-expiry-wait-function
      (lambda (group)
       (cond ((string= group "mail.private")
              31)
             ((string= group "mail.junk")
              1)
             ((string= group "important")
```


```
          'never)
        (t
          6))))
```

The group names that this function is fed are "unadorned" group names--no `nnml:' prefixes and the like.

The `nnmail-expiry-wait` variable and `nnmail-expiry-wait-function` function can be either a number (not necessarily an integer) or the symbols `immediate` or `never`.

You can also use the `expiry-wait` group parameter to selectively change the expiry period (see section Group Parameters).

If `nnmail-keep-last-article` is non-`nil`, Gnus will never expire the final article in a mail newsgroup. This is to make life easier for procmail users.

By the way, that line up there about Gnus never expiring non-expirable articles is a lie. If you put `total-expire` in the group parameters, articles will not be marked as expirable, but all read articles will be put through the expiry process. Use with extreme caution. Even more dangerous is the `gnus-total-expirable-newsgroups` variable. All groups that match this regexp will have all read articles put through the expiry process, which means that *all* old mail articles in the groups in question will be deleted after a while. Use with extreme caution, and don't come crying to me when you discover that the regexp you used matched the wrong group and all your important mail has disappeared. Be a *man*! Or a *woman*! Whatever you feel more comfortable with! So there!

Duplicates

If you are a member of a couple of mailing list, you will sometime receive two copies of the same mail. This can be quite annoying, so `nnmail` checks for and treats any duplicates it might find. To do this, it keeps a cache of old `Message-ID`s - `nnmail-message-id-cache-file`, which is `` `~/.nnmail-cache' `` by default. The approximate maximum number of `Message-ID`s stored there is controlled by the `nnmail-message-id-cache-length` variable, which is 1000 by default. (So 1000 `Message-ID`s will be stored.) If all this sounds scary to you, you can set `nnmail-treat-duplicates` to `warn` (which is what it is by default), and `nnmail` won't delete duplicate mails. Instead it will generate a brand new `Message-ID` for the mail and insert a warning into the head of the mail saying that it thinks that this is a duplicate of a different message.

This variable can also be a function. If that's the case, the function will be called from a buffer narrowed to the message in question with the `Message-ID` as a parameter. The function must return either `nil`, `warn`, or `delete`.

You can turn this feature off completely by setting the variable to `nil`.

If you want all the duplicate mails to be put into a special duplicates group, you could do that using the normal mail split methods:

```
(setq nnmail-split-fancy
      '(| ;; Messages duplicates go to a separate group.
          ("gnus-warning" "duplication of message" "duplicate")
```


```
          ;; Message from daemons, postmaster, and the like to anoth
          (any mail "mail.misc")
          ;; Other rules.
          [ ... ] ))
```

Or something like:

```
(setq nnmail-split-methods
      '(("duplicates" "^Gnus-Warning:")
        ;; Other rules.
        [...]))
```

Here's a neat feature: If you know that the recipient reads her mail with Gnus, and that she has `nnmail-treat-duplicates` set to `delete`, you can send her as many insults as you like, just by using a `Message-ID` of a mail that you know that she's already received. Think of all the fun! She'll never see any of it! Whee!

Not Reading Mail

If you start using any of the mail backends, they have the annoying habit of assuming that you want to read mail with them. This might not be unreasonable, but it might not be what you want.

If you set `nnmail-spool-file` to `nil`, none of the backends will ever attempt to read incoming mail, which should help.

This might be too much, if, for instance, you are reading mail quite happily with `nnml` and just want to peek at some old RMAIL file you have stashed away with `nnbabyl`. All backends have variables called backend-`get-new-mail`. If you want to disable the `nnbabyl` mail reading, you edit the virtual server for the group to have a setting where `nnbabyl-get-new-mail` to `nil`.

All the mail backends will call `nn*-prepare-save-mail-hook` narrowed to the article to be saved before saving it when reading incoming mail.

Choosing a Mail Backend

Gnus will read the mail spool when you activate a mail group. The mail file is first copied to your home directory. What happens after that depends on what format you want to store your mail in.

Unix Mail Box

The nnmbox backend will use the standard Un*x mbox file to store mail. `nnmbox` will add extra headers to each mail article to say which group it belongs in.

Virtual server settings:

`nnmbox-mbox-file`

The name of the mail box in the user's home directory.

`nnmbox-active-file`

The name of the active file for the mail box.

`nnmbox-get-new-mail`

If non-`nil`, `nnmbox` will read incoming mail and split it into groups.

Rmail Babyl

The nnbabyl backend will use a babyl mail box (aka. rmail mbox) to store mail. `nnbabyl` will add extra headers to each mail article to say which group it belongs in.

Virtual server settings:

`nnbabyl-mbox-file`

The name of the rmail mbox file.

`nnbabyl-active-file`

The name of the active file for the rmail box.

`nnbabyl-get-new-mail`

If non-`nil`, `nnbabyl` will read incoming mail.

Mail Spool

The nnml spool mail format isn't compatible with any other known format. It should be used with some caution.

If you use this backend, Gnus will split all incoming mail into files; one file for each mail, and put the articles into the correct directories under the directory specified by the `nnml-directory` variable. The default value is `` `~/Mail/' ``.

You do not have to create any directories beforehand; Gnus will take care of all that.

If you have a strict limit as to how many files you are allowed to store in your account, you should not use this backend. As each mail gets its own file, you might very well occupy thousands of inodes within a few weeks. If this is no problem for you, and it isn't a problem for you having your friendly systems administrator walking around, madly, shouting "Who is eating all my inodes?! Who? Who!?!", then you should know that this is probably the fastest format to use. You do not have to trudge through a big mbox file just to read your new mail.

`nnml` is probably the slowest backend when it comes to article splitting. It has to create lots of files, and it also generates NOV databases for the incoming mails. This makes is the fastest backend when it comes to reading mail.

Virtual server settings:

`nnml-directory`

All `nnml` directories will be placed under this directory.

`nnml-active-file`

The active file for the `nnml` server.

`nnml-newsgroups-file`

The `nnml` group descriptions file. See section Newsgroups File Format.

`nnml-get-new-mail`

If non-`nil`, `nnml` will read incoming mail.

`nnml-nov-is-evil`

If non-`nil`, this backend will ignore any NOV files.

`nnml-nov-file-name`

The name of the NOV files. The default is `` `.overview' ``.

`nnml-prepare-save-mail-hook`

Hook run narrowed to an article before saving.

If your `nnml` groups and NOV files get totally out of whack, you can do a complete update by typing M-x nnml-generate-nov-databases. This command will trawl through the entire `nnml` hierarchy, looking at each and every article, so it might take a while to complete.

MH Spool

`nnmh` is just like `nnml`, except that is doesn't generate NOV databases and it doesn't keep an active file. This makes `nnmh` a *much* slower backend than `nnml`, but it also makes it easier to write procmail scripts for.

Virtual server settings:

`nnmh-directory`

All `nnmh` directories will be located under this directory.

`nnmh-get-new-mail`

If non-`nil`, `nnmh` will read incoming mail.

`nnmh-be-safe`

If non-`nil`, `nnmh` will go to ridiculous lengths to make sure that the articles in the folder are actually what Gnus thinks they are. It will check date stamps and stat everything in sight, so setting this to `t` will mean a serious slow-down. If you never use anything but Gnus to read the `nnmh` articles, you do not have to set this variable to `t`.

Mail Folders

`nnfolder` is a backend for storing each mail group in a separate file. Each file is in the standard Un*x mbox format. `nnfolder` will add extra headers to keep track of article numbers and arrival dates.

Virtual server settings:

`nnfolder-directory`

All the `nnfolder` mail boxes will be stored under this directory.

`nnfolder-active-file`

The name of the active file.

`nnfolder-newsgroups-file`

The name of the group descriptions file. See section Newsgroups File Format.

`nnfolder-get-new-mail`

If non-`nil`, `nnfolder` will read incoming mail.

If you have lots of `nnfolder`-like files you'd like to read with `nnfolder`, you can use the M-x nnfolder-generate-active-file command to make `nnfolder` aware of all likely files in `nnfolder-directory`.

Other Sources

Gnus can do more than just read news or mail. The methods described below allow Gnus to view directories and files as if they were newsgroups.

Directory Groups

If you have a directory that has lots of articles in separate files in it, you might treat it as a newsgroup. The files have to have numerical names, of course.

This might be an opportune moment to mention `ange-ftp`, that most wonderful of all wonderful Emacs packages. When I wrote `nndir`, I didn't think much about it--a backend to read directories. Big deal.

`ange-ftp` changes that picture dramatically. For instance, if you enter `` `"/ftp.hpc.uh.edu:/pub/emacs/ding-list/"' `` as the the directory name, ange-ftp will actually allow you to read this directory over at `sina' as a newsgroup. Distributed news ahoy!

`nndir` will use NOV files if they are present.

`nndir` is a "read-only" backend--you can't delete or expire articles with this method. You can use `nnmh` or `nnml` for whatever you use `nndir` for, so you could switch to any of those methods if you feel the need to have a non-read-only `nndir`.

Anything Groups

From the `nndir` backend (which reads a single spool-like directory), it's just a hop and a skip to `nneething`, which pretends that any arbitrary directory is a newsgroup. Strange, but true.

When `nneething` is presented with a directory, it will scan this directory and assign article numbers to each file. When you enter such a group, `nneething` must create "headers" that Gnus can use. After all, Gnus is a newsreader, in case you're forgetting. `nneething` does this in a two-step process. First, it snoops each file in question. If the file looks like an article (i.e., the first few lines look like headers), it will use this as the head. If this is just some arbitrary file without a head (eg. a C source file), `nneething` will cobble up a header out of thin air. It will use file ownership, name and date and do whatever it can with these elements.

All this should happen automatically for you, and you will be presented with something that looks very much like a newsgroup. Totally like a newsgroup, to be precise. If you select an article, it will be displayed in the article buffer, just as usual.

If you select a line that represents a directory, Gnus will pop you into a new summary buffer for this `nneething` group. And so on. You can traverse the entire disk this way, if you feel like, but remember that Gnus is not dired, really, and does not intend to be, either.

There are two overall modes to this action--ephemeral or solid. When doing the ephemeral thing (i.e., G D from the group buffer), Gnus will not store information on what files you have read, and what files are new, and so on. If you create a solid `nneething` group the normal way with G m, Gnus will store a mapping table between article numbers and file names, and you can treat this group like any other groups. When you activate a solid `nneething` group, you will be told how many unread articles it contains, etc., etc.

Some variables:

`nneething-map-file-directory`

All the mapping files for solid `nneething` groups will be stored in this directory, which defaults to `` `~/.nneething/' ``.

`nneething-exclude-files`

All files that match this regexp will be ignored. Nice to use to exclude auto-save files and the like, which is what it does by default.

`nneething-map-file`

Name of the map files.

Document Groups

`nndoc` is a cute little thing that will let you read a single file as a newsgroup. Several files types are supported:

`babyl`

The babyl (rmail) mail box.

`mbox`

The standard Unix mbox file.

`mmdf`

The MMDF mail box format.

`news`

Several news articles appended into a file.

`rnews`

The rnews batch transport format.

`forward`

Forwarded articles.

`mime-digest`

MIME (RFC 1341) digest format.

`standard-digest`

The standard (RFC 1153) digest format.

`slack-digest`

Non-standard digest format--matches most things, but does it badly.

You can also use the special "file type" `guess`, which means that `nndoc` will try to guess what file type it is looking at. `digest` means that `nndoc` should guess what digest type the file is.

`nndoc` will not try to change the file or insert any extra headers into it--it will simply, like, let you use the file as the basis for a group. And that's it.

If you have some old archived articles that you want to insert into your new & spiffy Gnus mail backend, `nndoc` can probably help you with that. Say you have an old `` `RMAIL' `` file with mail that you now want to split into your new `nnml` groups. You look at that file using `nndoc`, set the process mark on all the articles in the buffer (M P b, for instance), and then re-spool (B r) using `nnml`. If all goes well, all the mail in the `` `RMAIL' `` file is now also stored in lots of `nnml` directories, and you can delete that pesky `` `RMAIL' `` file. If you have the guts!

Virtual server variables:

`nndoc-article-type`

This should be one of `mbox`, `babyl`, `digest`, `mmdf`, `forward`, `news`, `rnews`, `mime-digest`, `clari-briefs`, or `guess`.

`nndoc-post-type`

This variable says whether Gnus is to consider the group a news group or a mail group. There are two legal values: `mail` (the default) and `news`.

SOUP

In the PC world people often talk about "offline" newsreaders. These are thingies that are combined reader/news transport monstrosities. With built-in modem programs. Yecchh!

Of course, us Unix Weenie types of human beans use things like `uucp` and, like, `nntpd` and set up proper news and mail transport things like Ghod intended. And then we just use normal newsreaders.

However, it can sometimes be convenient to do something a that's a bit easier on the brain if you have a very slow modem, and you're not really that interested in doing things properly.

A file format called SOUP has been developed for transporting news and mail from servers to home machines and back again. It can be a bit fiddly.

1. You log in on the server and create a SOUP packet. You can either use a dedicated SOUP thingie, or you can use Gnus to create the packet with the O s command.
2. You transfer the packet home. Rail, boat, car or modem will do fine.
3. You put the packet in your home directory.
4. You fire up Gnus using the `nnsoup` backend as the native server.
5. You read articles and mail and answer and followup to the things you want.
6. You do the G s r command to pack these replies into a SOUP packet.
7. You transfer this packet to the server.

8. You use Gnus to mail this packet out with the G s s command.
9. You then repeat until you die.

So you basically have a bipartite system--you use `nnsoup` for reading and Gnus for packing/sending these SOUP packets.

SOUP Commands

G s b

Pack all unread articles in the current group (`gnus-group-brew-soup`). This command understands the process/prefix convention.

G s w

Save all data files (`gnus-soup-save-areas`).

G s s

Send all replies from the replies packet (`gnus-soup-send-replies`).

G s p

Pack all files into a SOUP packet (`gnus-soup-pack-packet`).

G s r

Pack all replies into a replies packet (`nnsoup-pack-replies`).

O s

This summary-mode command adds the current article to a SOUP packet (`gnus-soup-add-article`). It understands the process/prefix convention.

There are a few variables to customize where Gnus will put all these thingies:

`gnus-soup-directory`

Directory where Gnus will save intermediate files while composing SOUP packets. The default is `` `~/SoupBrew/' ``.

`gnus-soup-replies-directory`

This is what Gnus will use as a temporary directory while sending our reply packets. The default is `` `~/SoupBrew/SoupReplies/' ``.

`gnus-soup-prefix-file`

Name of the file where Gnus stores the last used prefix. The default is `gnus-prefix'.

`gnus-soup-packer`

A format string command for packing a SOUP packet. The default is `tar cf - %s | gzip > $HOME/Soupout%d.tgz'.

`gnus-soup-unpacker`

Format string command for unpacking a SOUP packet. The default is `gunzip -c %s | tar xvf -'.

`gnus-soup-packet-directory`

Where Gnus will look for reply packets. The default is `` `~/' ``.

`gnus-soup-packet-regexp`

TOC

Regular expression matching SOUP reply packets in `gnus-soup-packet-directory`.

SOUP Groups

`nnsoup` is the backend for reading SOUP packets. It will read incoming packets, unpack them, and put them in a directory where you can read them at leisure.

These are the variables you can use to customize its behavior:

`nnsoup-tmp-directory`

When `nnsoup` unpacks a SOUP packet, it does it in this directory. (`` `/tmp/' `` by default.)

`nnsoup-directory`

`nnsoup` then moves each message and index file to this directory. The default is `` `~/SOUP/' ``.

`nnsoup-replies-directory`

All replies will stored in this directory before being packed into a reply packet. The default is `` `~/SOUP/replies/"' ``.

`nnsoup-replies-format-type`

The SOUP format of the replies packets. The default is `?n' (rnews), and I don't think you should touch that variable. I probably shouldn't even have documented it. Drats! Too late!

`nnsoup-replies-index-type`

The index type of the replies packet. The is `?n', which means "none". Don't fiddle with this one either!

`nnsoup-active-file`

Where `nnsoup` stores lots of information. This is not an "active file" in the `nntp` sense; it's an Emacs Lisp file. If you lose this file or mess it up in any way, you're dead. The default is `` `~/SOUP/active' ``.

`nnsoup-packer`

Format string command for packing a reply SOUP packet. The default is `tar cf - %s | gzip > $HOME/Soupin%d.tgz'.

`nnsoup-unpacker`

Format string command for unpacking incoming SOUP packets. The default is `gunzip -c %s | tar xvf -'.

`nnsoup-packet-directory`

Where `nnsoup` will look for incoming packets. The default is `` `~/' ``.

`nnsoup-packet-regexp`

Regular expression matching incoming SOUP packets. The default is `Soupout'.

SOUP Replies

Just using `nnsoup` won't mean that your postings and mailings end up in SOUP reply packets automagically. You have to work a bit more for that to happen.

The `nnsoup-set-variables` command will set the appropriate variables to ensure that all your

followups and replies end up in the SOUP system.

In specific, this is what it does:

```
(setq gnus-inews-article-function 'nnsoup-request-post)
(setq send-mail-function 'nnsoup-request-mail)
```

And that's it, really. If you only want news to go into the SOUP system you just use the first line. If you only want mail to be SOUPed you use the second.

Combined Groups

Gnus allows combining a mixture of all the other group types into bigger groups.

Virtual Groups

An nnvirtual group is really nothing more than a collection of other groups.

For instance, if you are tired of reading many small group, you can put them all in one big group, and then grow tired of reading one big, unwieldy group. The joys of computing!

You specify `nnvirtual` as the method. The address should be a regexp to match component groups.

All marks in the virtual group will stick to the articles in the component groups. So if you tick an article in a virtual group, the article will also be ticked in the component group from whence it came. (And vice versa--marks from the component groups will also be shown in the virtual group.)

Here's an example `nnvirtual` method that collects all Andrea Dworkin newsgroups into one, big, happy newsgroup:

```
(nnvirtual "^alt\\.fan\\.andrea-dworkin$\\|^rec\\.dworkin.*")
```

The component groups can be native or foreign; everything should work smoothly, but if your computer explodes, it was probably my fault.

Collecting the same group from several servers might actually be a good idea if users have set the Distribution header to limit distribution. If you would like to read `soc.motss' both from a server in Japan and a server in Norway, you could use the following as the group regexp:

```
"^nntp+some.server.jp:soc.motss$\\|^nntp+some.server.no:soc.motss$"
```

This should work kinda smoothly--all articles from both groups should end up in this one, and there should be no duplicates. Threading (and the rest) will still work as usual, but there might be problems with the sequence of articles. Sorting on date might be an option here (see section Selecting a Group.

One limitation, however--all groups that are included in a virtual group has to be alive (i.e., subscribed or unsubscribed). Killed or zombie groups can't be component groups for `nnvirtual` groups.

If the `nnvirtual-always-rescan` is non-`nil`, `nnvirtual` will always scan groups for unread articles when entering a virtual group. If this variable is `nil` (which is the default) and you read articles in a component group after the virtual group has been activated, the read articles from the component group will show up when you enter the virtual group. You'll also see this effect if you have two virtual groups that contain the same component group. If that's the case, you should set this variable to `t`. Or you can just tap `M-g` on the virtual group every time before you enter it--it'll have much the same effect.

Kibozed Groups

Kibozing is defined by OED as "grepping through (parts of) the news feed". `nnkiboze` is a backend that will do this for you. Oh joy! Now you can grind any NNTP server down to a halt with useless requests! Oh happiness!

To create a kibozed group, use the G k command in the group buffer.

The address field of the `nnkiboze` method is, as with `nnvirtual`, a regexp to match groups to be "included" in the `nnkiboze` group. There most similarities between `nnkiboze` and `nnvirtual` ends.

In addition to this regexp detailing component groups, an `nnkiboze` group must have a score file to say what articles that are to be included in the group (see section Scoring).

You must run M-x nnkiboze-generate-groups after creating the `nnkiboze` groups you want to have. This command will take time. Lots of time. Oodles and oodles of time. Gnus has to fetch the headers from all the articles in all the components groups and run them through the scoring process to determine if there are any articles in the groups that are to be part of the `nnkiboze` groups.

Please limit the number of component groups by using restrictive regexps. Otherwise your sysadmin may become annoyed with you, and the NNTP site may throw you off and never let you back in again. Stranger things have happened.

`nnkiboze` component groups do not have to be alive--they can be dead, and they can be foreign. No restrictions.

The generation of an `nnkiboze` group means writing two files in `nnkiboze-directory`, which is `` `~/News/' `` by default. One contains the NOV header lines for all the articles in the group, and the other is an additional `` `.newsrc' `` file to store information on what groups that have been searched through to find component articles.

Articles that are marked as read in the `nnkiboze` group will have their NOV lines removed from the NOV file.

Go to the previous, next section.

Go to the previous, next section.

Scoring

Other people use kill files, but we here at Gnus Towers like scoring better than killing, so we'd rather switch than fight. They do something completely different as well, so sit up straight and pay attention!

All articles have a default score (`gnus-summary-default-score`), which is 0 by default. This score may be raised or lowered either interactively or by score files. Articles that have a score lower than `gnus-summary-mark-below` are marked as read.

Gnus will read any score files that apply to the current group before generating the summary buffer.

There are several commands in the summary buffer that insert score entries based on the current article. You can, for instance, ask Gnus to lower or increase the score of all articles with a certain subject.

There are two sorts of scoring entries: Permanent and temporary. Temporary score entries are self-expiring entries. Any entries that are temporary and have not been used for, say, a week, will be removed silently to help keep the sizes of the score files down.

Summary Score Commands

The score commands that alter score entries do not actually modify real score files. That would be too inefficient. Gnus maintains a cache of previously loaded score files, one of which is considered the current score file alist. The score commands simply insert entries into this list, and upon group exit, this list is saved.

The current score file is by default the group's local score file, even if no such score file actually exists. To insert score commands into some other score file (eg. `` `all.SCORE' ``), you must first make this score file the current one.

General score commands that don't actually change the score file:

V s
: Set the score of the current article (`gnus-summary-set-score`).

V S
: Display the score of the current article (`gnus-summary-current-score`).

V t
: Display all score rules that have been used on the current article (`gnus-score-find-trace`).

V R
: Run the current summary through the scoring process (`gnus-summary-rescore`). This might be useful if you're playing around with your score files behind Gnus' back and want to see the effect you're having.

V a
: Add a new score entry, and allow specifying all elements (`gnus-summary-score-entry`).

V c
: Make a different score file the current (`gnus-score-change-score-file`).

V e

Edit the current score file (`gnus-score-edit-current-scores`). You will be popped into a `gnus-score-mode` buffer (see section Score File Editing).

V f

Edit a score file and make this score file the current one (`gnus-score-edit-file`).

V F

Flush the score cahe (`gnus-score-flush-cache`). This is useful after editing score files.

V C

Customize a score file in a visually pleasing manner (`gnus-score-customize`).

I C-i

Increase the score of the current article (`gnus-summary-raise-score`).

L C-l

Lower the score of the current article (`gnus-summary-lower-score`).

The rest of these commands modify the local score file.

V m

Prompt for a score, and mark all articles with a score below this as read (`gnus-score-set-mark-below`).

V E

Expunge all articles with a score below the default score (or the numeric prefix) (`gnus-score-set-expunge-below`).

The keystrokes for actually making score entries follow a very regular pattern, so there's no need to list all the commands. (Hundreds of them.)

1. The first key is either I (upper case i) for increasing the score or L for lowering the score.
2. The second key says what header you want to score on. The following keys are available:

   a

   Score on the author name.

   s

   Score on the subject line.

   x

   Score on the Xref line--i.e., the cross-posting line.

   t

   Score on thread--the References line.

   d

   Score on the date.

   l

   Score on the number of lines.

   i

   Score on the Message-ID.

   f

Score on followups.

b

Score on the body.

h

Score on the head.

3. The third key is the match type. Which match types are legal depends on what headers you are scoring on.

`strings`

e

Exact matching.

s

Substring matching.

f

Fuzzy matching.

r

Regexp matching

- date

b

Before date.

a

At date.

n

This date.

- number

<

Less than number.

=

Equal to number.

>

Greater than number.

4. The fourth and final key says whether this is a temporary (i.e., expiring) score entry, or a permanent (i.e., non-expiring) score entry, or whether it is to be done immediately, without adding to the score file.

t

Temporary score entry.

p

Permanent score entry.

i

Immediately scoring.

So, let's say you want to increase the score on the current author with exact matching permanently: I a e p. If you want to lower the score based on the subject line, using substring matching, and make a temporary score entry:

L s s t. Pretty easy.

To make things a bit more complicated, there are shortcuts. If you use a capital letter on either the second or third keys, Gnus will use defaults for the remaining one or two keystrokes. The defaults are "substring" and "temporary". So I A is the same as I a s t, and I a R is the same as I a r t.

The `gnus-score-mimic-keymap` says whether these commands will pretend they are keymaps or not.

Group Score Commands

There aren't many of these as yet, I'm afraid.

W f

> Gnus maintains a cache of score alists to avoid having to reload them all the time. This command will flush the cache (`gnus-score-flush-cache`).

Score Variables

`gnus-use-scoring`

> If `nil`, Gnus will not check for score files, and will not, in general, do any score-related work. This is `t` by default.

`gnus-kill-killed`

> If this variable is `nil`, Gnus will never apply score files to articles that have already been through the kill process. While this may save you lots of time, it also means that if you apply a kill file to a group, and then change the kill file and want to run it over you group again to kill more articles, it won't work. You have to set this variable to `t` to do that. (It is `t` by default.)

`gnus-kill-files-directory`

> All kill and score files will be stored in this directory, which is initialized from the `SAVEDIR` environment variable by default. This is `` `~/News/' `` by default.

`gnus-score-file-suffix`

> Suffix to add to the group name to arrive at the score file name (`SCORE' by default.)

`gnus-score-uncacheable-files`

> All score files are normally cached to avoid excessive re-loading of score files. However, if this might make you Emacs grow big and bloated, so this regexp can be used to weed out score files that are unlikely to be needed again. It would be a bad idea to deny caching of `` `all.SCORE' ``, while it might be a good idea to not cache `` `comp.infosystems.www.authoring.misc.ADAPT' ``. In fact, this variable is `ADAPT$' by default, so no adaptive score files will be cached.

`gnus-save-score`

> If you have really complicated score files, and do lots of batch scoring, then you might set this variable to `t`. This will make Gnus save the scores into the `` `.newsrc.eld' `` file.

`gnus-score-interactive-default-score`

> Score used by all the interactive raise/lower commands to raise/lower score with. Default is 1000, which may seem excessive, but this is to ensure that the adaptive scoring scheme gets enough room to play with. We don't want the small changes from the adaptive scoring to overwrite manually entered data.

`gnus-summary-default-score`

Default score of an article, which is 0 by default.

`gnus-score-over-mark`

Mark (in the third column) used for articles with a score over the default. Default is `+'.

`gnus-score-below-mark`

Mark (in the third column) used for articles with a score below the default. Default is `-'.

`gnus-score-find-score-files-function`

Function used to find score files for the current group. This function is called with the name of the group as the argument.

Predefined functions available are:

`gnus-score-find-single`

Only apply the group's own score file.

`gnus-score-find-bnews`

Apply all score files that match, using bnews syntax. This is the default. For instance, if the current group is `gnu.emacs.gnus', `all.emacs.all.SCORE', `not.alt.all.SCORE' and `gnu.all.SCORE' would all apply. In short, the instances of `all' in the score file names are translated into `.*', and then a regexp match is done.

This means that if you have some score entries that you want to apply to all groups, then you put those entries in the `all.SCORE' file.

`gnus-score-find-hierarchical`

Apply all score files from all the parent groups. This means that you can't have score files like `all.SCORE' or `all.emacs.SCORE', but you can have `SCORE', `comp.SCORE' and `comp.emacs.SCORE'.

This variable can also be a list of functions. In that case, all these functions will be called, and all the returned lists of score files will be applied. These functions can also return lists of score alists directly. In that case, the functions that return these non-file score alists should probably be placed before the "real" score file functions, to ensure that the last score file returned is the local score file. Phu.

- gnus-score-expiry-days This variable says how many days should pass before an unused score file entry is expired. If this variable is `nil`, no score file entries are expired. It's 7 by default.
- gnus-update-score-entry-dates If this variable is non-`nil`, matching score entries will have their dates updated. (This is how Gnus controls expiry--all non-matching entries will become too old while matching entries will stay fresh and young.) However, if you set this variable to `nil`, even matching entries will grow old and will have to face that oh-so grim reaper.
- gnus-score-after-write-file-function Function called with the name of the score file just written.

Score File Format

A score file is an `emacs-lisp` file that normally contains just a single form. Casual users are not expected to edit these files; everything can be changed from the summary buffer.

Anyway, if you'd like to dig into it yourself, here's an example:

```
(("from"
  ("Lars Ingebrigtsen" -10000)
```


```
 ("Per Abrahamsen")
 ("larsi\\|lmi" -50000 nil R))
("subject"
 ("Ding is Badd" nil 728373))
("xref"
 ("alt.politics" -1000 728372 s))
("lines"
 (2 -100 nil <))
(mark 0)
(expunge -1000)
(mark-and-expunge -10)
(read-only nil)
(orphan -10)
(adapt t)
(files "/hom/larsi/News/gnu.SCORE")
(exclude-files "all.SCORE")
(local (gnus-newsgroup-auto-expire t)
       (gnus-summary-make-false-root 'empty))
(eval (ding)))
```

This example demonstrates absolutely everything about a score file.

Even though this looks much like lisp code, nothing here is actually `eval`ed. The lisp reader is used to read this form, though, so it has to be legal syntactically, if not semantically.

Six keys are supported by this alist:

`STRING`

If the key is a string, it is the name of the header to perform the match on. Scoring can only be performed on these eight headers: `From`, `Subject`, `References`, `Message-ID`, `Xref`, `Lines`, `Chars` and `Date`. In addition to these headers, there are three strings to tell Gnus to fetch the entire article and do the match on larger parts of the article: `Body` will perform the match on the body of the article, `Head` will perform the match on the head of the article, and `All` will perform the match on the entire article. Note that using any of these last three keys will slow down group entry *considerably*. The final "header" you can score on is `Followup`. These score entries will result in new score entries being added for all follow-ups to articles that matches these score entries.

Following this key is a arbitrary number of score entries, where each score entry has one to four elements.

The first element is the match element. On most headers this will be a string, but on the Lines and Chars headers, this must be an integer.

If the second element is present, it should be a number--the score element. This number should be an integer in the neginf to posinf interval. This number is added to the score of the article if the match is successful. If this element is not present, the `gnus-score-interactive-default-score` number will be used instead. This is 1000 by default.

If the third element is present, it should be a number--the date element. This date says when the last time this score entry matched, which provides a mechanism for expiring the score entries. It this element is not present, the score entry is permanent. The date is represented by the number of days since December 31, 1 ce.

If the fourth element is present, it should be a symbol--the type element. This element

specifies what function should be used to see whether this score entry matches the article. Wha match types that can be used depends on what header you wish to perform the match on.

From, Subject, References, Xref, Message-ID

For most header types, there are the `r` and `R` (regexp) as well as `s` and `S` (substring) types and `e` and `E` (exact match) types. If this element is not present, Gnus will assume that substring matching should be used. `R` and `S` differ from the other two in that the matches will be done in a case-sensitive manner. All these one-letter types are really just abbreviations for the `regexp`, `string` and `exact` types, which you can use instead, if you feel like.

Lines, Chars

These two headers use different match types: <, >, =, >= and <=.

Date

For the Date header we have three match types: `before`, `at` and `after`. I can't really imagine this ever being useful, but, like, it would feel kinda silly not to provide this function. Just in case. You never know. Better safe than sorry. Once burnt, twice shy. Don't judge a book by its cover. Never not have sex on a first date.

Head, Body, All

These three match keys use the same match types as the `From` (etc) header uses.

Followup

This match key will add a score entry on all articles that followup to some author. Uses the same match types as the `From` header uses.

Thread

This match key will add a score entry on all articles that are part of a thread. Uses the same match types as the `References` header uses.

- mark The value of this entry should be a number. Any articles with a score lower than this number will be marked as read.
- expunge The value of this entry should be a number. Any articles with a score lower than this number will be removed from the summary buffer.
- mark-and-expunge The value of this entry should be a number. Any articles with a score lower than this number will be marked as read and removed from the summary buffer.
- thread-mark-and-expunge The value of this entry should be a number. All articles that belong to a thread that has a total score below this number will be marked as read and removed from the summary buffer. `gnus-thread-score-function` says how to compute the total score for a thread.
- files The value of this entry should be any number of file names. These files are assumed to be score files as well, and will be loaded the same way this one was.
- exclude-files The clue of this entry should be any number of files. This files will not be loaded, even though they would normally be so, for some reason or other.
- eval The value of this entry will be `eval`el. This element will be ignored when handling global score files.
- read-only Read-only score files will not be updated or saved. Global score files should feature this atom (see section Global Score Files).
- orphan The value of this entry should be a number. Articles that do not have parents will get this number added to their scores. Imagine you follow some high-volume newsgroup, like `comp.lang.c'. Most likely you will only follow a few of the threads, also want to see any new threads.

You can do this with the following two score file entries:


```
(orphan -500)
(mark-and-expunge -100)
```

When you enter the group the first time, you will only see the new threads. You then raise the score of the threads that you find interesting (with I T or I S), and ignore (C y) the rest. Next time you enter the group, you will see new articles in the interesting threads, plus any new threads.

I.e. -- the orphan score atom is for high-volume groups where there exist a few interesting threads which can't be found automatically by ordinary scoring rules.

- adapt This entry controls the adaptive scoring. If it is `t`, the default adaptive scoring rules will be used. If it is `ignore`, no adaptive scoring will be performed on this group. If it is a list, this list will be used as the adaptive scoring rules. If it isn't present, or is something other than `t` or `ignore`, the default adaptive scoring rules will be used. If you want to use adaptive scoring on most groups, you'd set `gnus-use-adaptive-scoring` to `t`, and insert an `(adapt ignore)` in the groups where you do not want adaptive scoring. If you only want adaptive scoring in a few groups, you'd set `gnus-use-adaptive-scoring` to `nil`, and insert `(adapt t)` in the score files of the groups where you want it.
- adapt-file All adaptive score entries will go to the file named by this entry. It will also be applied when entering the group. This atom might be handy if you want to adapt on several groups at once, using the same adaptive file for a number of groups.
- local The value of this entry should be a list of `(VAR VALUE)` pairs. Each var will be made buffer-local to the current summary buffer, and set to the value specified. This is a convenient, if somewhat strange, way of setting variables in some groups if you don't like hooks much.

Score File Editing

You normally enter all scoring commands from the summary buffer, but you might feel the urge to edit them by hand as well, so we've supplied you with a mode for that.

It's simply a slightly customized `emacs-lisp` mode, with these additional commands:

C-c C-c

> Save the changes you have made and return to the summary buffer (`gnus-score-edit-done`).

C-c C-d

> Insert the current date in numerical format (`gnus-score-edit-insert-date`). This is really the day number, if you were wondering.

C-c C-p

> The adaptive score files are saved in an unformatted fashion. If you intend to read one of these files, you want to pretty print it first. This command (`gnus-score-pretty-print`) does that for you.

Type M-x gnus-score-mode to use this mode.

`gnus-score-menu-hook` is run in score mode buffers.

In the summary buffer you can use commands like V f and V e to begin editing score files.

Adaptive Scoring

If all this scoring is getting you down, Gnus has a way of making it all happen automatically--as if by magic. Or rather, as if by artificial stupidity, to be precise.

When you read an article, or mark an article as read, or kill an article, you leave marks behind. On exit from the group, Gnus can sniff these marks and add score elements depending on what marks it finds. You turn on this ability by setting `gnus-use-adaptive-scoring` to `t`.

To give you complete control over the scoring process, you can customize the `gnus-default-adaptive-score-alist` variable. For instance, it might look something like this:

```
(defvar gnus-default-adaptive-score-alist
  '((gnus-unread-mark)
    (gnus-ticked-mark (from 4))
    (gnus-dormant-mark (from 5))
    (gnus-del-mark (from -4) (subject -1))
    (gnus-read-mark (from 4) (subject 2))
    (gnus-expirable-mark (from -1) (subject -1))
    (gnus-killed-mark (from -1) (subject -3))
    (gnus-kill-file-mark)
    (gnus-ancient-mark)
    (gnus-low-score-mark)
    (gnus-catchup-mark (from -1) (subject -1))))
```

As you see, each element in this alist has a mark as a key (either a variable name or a "real" mark--a character). Following this key is a arbitrary number of header/score pairs. If there are no header/score pairs following the key, no adaptive scoring will be done on articles that have that key as the article mark. For instance, articles with `gnus-unread-mark` in the example above will not get adaptive score entries.

Each article can have only one mark, so just a single of these rules will be applied to each article.

To take `gnus-del-mark` as an example--this alist says that all articles that have that mark (i.e., are marked with `D') will have a score entry added to lower based on the `From` header by -4, and lowered by `Subject` by -1. Change this to fit your prejudices.

If you have marked 10 articles with the same subject with `gnus-del-mark`, the rule for that mark will be applied ten times. That means that that subject will get a score of ten times -1, which should be, unless I'm much mistaken, -10.

The headers you can score on are `from`, `subject`, `message-id`, `references`, `xref`, `lines`, `chars` and `date`. In addition, you can score on `followup`, which will create an adaptive score entry that matches on the `References` header using the `Message-ID` of the current article, thereby matching the following thread.

You can also score on `thread`, which will try to score all articles that appear in a thread. `thread` matches uses a `Message-ID` to match on the `References` header of the article. If the match is made, the `Message-ID` of the article is added to the `thread` rule. (Think about it. I'd recommend two aspirins afterwards.)

If you use this scheme, you should set the score file atom `mark` to something small--like -300, perhaps, to avoid having small random changes result in articles getting marked as read.

After using adaptive scoring for a week or so, Gnus should start to become properly trained and enhance the authors you like best, and kill the authors you like least, without you having to say so explicitly.

You can control what groups the adaptive scoring is to be performed on by using the score files (see section Score File Format). This will also let you use different rules in different groups.

The adaptive score entries will be put into a file where the name is the group name with `gnus-adaptive-file-suffix` appended. The default is `ADAPT'.

When doing adaptive scoring, substring or fuzzy matching would probably give you the best results in most cases. However, if the header one matches is short, the possibility for false positives is great, so if the length of the match is less than `gnus-score-exact-adapt-limit`, exact matching will be used. If this variable is `nil`, exact matching will always be used to avoid this problem.

Followups To Yourself

Gnus offers two commands for picking out the `Message-ID` header in the current buffer. Gnus will then add a score rule that scores using this `Message-ID` on the `References` header of other articles. This will, in effect, increase the score of all articles that respond to the article in the current buffer. Quite useful if you want to easily note when people answer what you've said.

`gnus-score-followup-article`

This will add a score to articles that directly follow up your own article.

`gnus-score-followup-thread`

This will add a score to all articles that appear in a thread "below" your own article.

These two functions are both primarily meant to be used in hooks like `gnus-inews-article-hook`.

Scoring Tips

Crossposts

If you want to lower the score of crossposts, the line to match on is the `Xref` header.

```
("xref" (" talk.politics.misc:" -1000))
```

Multiple crossposts

If you want to lower the score of articles that have been crossposted to more than, say, 3 groups:

```
("xref" ("[^:\n]+:[0-9]+ +[^:\n]+:[0-9]+ +[^:\n]+:[0-9]+" -1000 nil r))
```

Matching on the body

This is generally not a very good idea--it takes a very long time. Gnus actually has to fetch each individual article from the server. But you might want to anyway, I guess. Even though there are three match keys (`Head`, `Body` and `All`), you should choose one and stick with it in each score file. If you use any two, each article will be fetched *twice*. If you want to match a bit on the `Head` and a bit on the `Body`, just use `All` for all the matches.

Marking as read

You will probably want to mark articles that has a score below a certain number as read. This is most

easily achieved by putting the following in your `all.SCORE' file:

```
((mark -100))
```

You may also consider doing something similar with `expunge`.

Negated character classes

If you say stuff like `[^abcd]*`, you may get unexpected results. That will match newlines, which might lead to, well, The Unknown. Say `[^abcd\n]*` instead.

Reverse Scoring

If you want to keep just articles that have `Sex with Emacs' in the subject header, and expunge all other articles, you could put something like this in your score file:

```
(("subject"
  ("Sex with Emacs" 2))
 (mark 1)
 (expunge 1))
```

So, you raise all articles that match `Sex with Emacs' and mark the rest as read, and expunge them to boot.

Global Score Files

Sure, other newsreaders have "global kill files". These are usually nothing more than a single kill file that applies to all groups, stored in the user's home directory. Bah! Puny, weak newsreaders!

What I'm talking about here are Global Score Files. Score files from all over the world, from users everywhere, uniting all nations in one big, happy score file union! Ange-score! New and untested!

All you have to do to use other people's score files is to set the `gnus-global-score-files` variable. One entry for each score file, or each score file directory. Gnus will decide by itself what score files are applicable to which group.

Say you want to use all score files in the `/ftp@ftp.some-where:/pub/score' directory and the single score file `/ftp@ftp.ifi.uio.no:/pub/larsi/ding/score/soc.motss.SCORE':

```
(setq gnus-global-score-files
      '("/ftp@ftp.ifi.uio.no:/pub/larsi/ding/score/soc.motss.SCORE"
        "/ftp@ftp.some-where:/pub/score/"))
```

Simple, eh? Directory names must end with a `/'. These directories are typically scanned only once during each Gnus session. If you feel the need to manually re-scan the remote directories, you can use the `gnus-score-search-global-directories` command.

Note that, at present, using this option will slow down group entry somewhat. (That is--a lot.)

If you want to start maintaining score files for other people to use, just put your score file up for anonymous ftp and announce it to the world. Become a retro-moderator! Participate in the retro-moderator wars sure to ensue, where retro-moderators battle it out for the sympathy of the people, luring them to use their score files on false

premises! Yay! The net is saved!

Here are some tips for the would-be retro-moderator, off the top of my head:

- Articles that are heavily crossposted are probably junk.
- To lower a single inappropriate article, lower by `Message-ID`.
- Particularly brilliant authors can be raised on a permanent basis.
- Authors that repeatedly post off-charter for the group can safely be lowered out of existence.
- Set the `mark` and `expunge` atoms to obliterate the nastiest articles completely.
- Use expiring score entries to keep the size of the file down. You should probably have a long expiry period, though, as some sites keep old articles for a long time.

... I wonder whether other newsreaders will support global score files in the future. *Snicker*. Yup, any day now, newsreaders like Blue Wave, xrn and 1stReader are bound to implement scoring. Should we start holding our breath yet?

Kill Files

Gnus still supports those pesky old kill files. In fact, the kill file entries can now be expiring, which is something I wrote before Daniel Quinlan thought of doing score files, so I've left the code in there.

In short, kill processing is a lot slower (and I do mean *a lot*) than score processing, so it might be a good idea to rewrite your kill files into score files.

Anyway, a kill file is a normal `emacs-lisp` file. You can put any forms into this file, which means that you can use kill files as some sort of primitive hook function to be run on group entry, even though that isn't a very good idea.

XCNormal kill files look like this:

```
(gnus-kill "From" "Lars Ingebrigtsen")
(gnus-kill "Subject" "ding")
(gnus-expunge "X")
```

This will mark every article written by me as read, and remove them from the summary buffer. Very useful, you'll agree.

Other programs use a totally different kill file syntax. If Gnus encounters what looks like a `rn` kill file, it will take a stab at interpreting it.

Two summary functions for editing a GNUS kill file:

M-k

Edit this group's kill file (`gnus-summary-edit-local-kill`).

M-K

Edit the general kill file (`gnus-summary-edit-global-kill`).

Two group mode functions for editing the kill files:

M-k

Edit this group's kill file (`gnus-group-edit-local-kill`).

M-K

Edit the general kill file (`gnus-group-edit-global-kill`).

Kill file variables:

`gnus-kill-file-name`

A kill file for the group `soc.motss' is normally called `soc.motss.KILL'. The suffix appended to the group name to get this file name is detailed by the `gnus-kill-file-name` variable. The "global" kill file (not in the score file sense of "global", of course) is called just `KILL'.

`gnus-kill-save-kill-file`

If this variable is non-`nil`, Gnus will save the kill file after processing, which is necessary if you use expiring kills.

`gnus-apply-kill-hook`

A hook called to apply kill files to a group. It is `(gnus-apply-kill-file)` by default. If you want to ignore the kill file if you have a score file for the same group, you can set this hook to `(gnus-apply-kill-file-unless-scored)`. If you don't want kill files to be processed, you should set this variable to `nil`.

`gnus-kill-file-mode-hook`

A hook called in kill-file mode buffers.

GroupLens

GroupLens is a collaborative filtering system that helps you work together with other people to find the quality news articles out of the huge volume of news articles generated every day.

To accomplish this the GroupLens system combines your opinions about articles you have already read with the opinions of others who have done likewise and gives you a personalized prediction for each unread news article. Think of GroupLens as a matchmaker. GroupLens watches how you rate articles, and finds other people that rate articles the same way. Once it has found for you some people you agree with it tells you, in the form of a prediction, what they thought of the article. You can use this prediction to help you decide whether or not you want to read the article.

Using GroupLens

To use GroupLens you must register a pseudonym with your local Better Bit Bureau (BBB). At the moment the only better bit in town is at `http://www.cs.umn.edu/Research/GroupLens/bbb.html'.

Once you have registered you'll need to set a couple of variables.

`gnus-use-grouplens`

Setting this variable to a non-`nil` value will make Gnus hook into all the relevant GroupLens functions.

`grouplens-pseudonym`

This variable should be set to the pseudonum you got when registering with the Better Bit Bureau.

`grouplens-newsgroups`

A list of groups that you want to get GroupLens predictions for.

Thats the minimum of what you need to get up and running with GroupLens. Once you've registered,

TOC

GroupLens will start giving you scores for articles based on the average of what other people think. But, to the real benefit of GroupLens you need to start rating articles yourself. Then the scores GroupLens gives you will be personalized for you, based on how the people you usually agree with have already rated.

Rating Articles

In GroupLens, an article is rated on a scale from 1 to 5, inclusive. Where 1 means something like this article is a waste of bandwidth and 5 means that the article was really good. The basic question to ask yourself is, "on a scale from 1 to 5 would I like to see more articles like this one?"

There are four ways to enter a rating for an article in GroupLens.

r

This function will prompt you for a rating on a scale of one to five.

k

This function will prompt you for a rating, and rate all the articles in the thread. This is really useful for some of those long running giant threads in rec.humor.

The next two commands, n and , take a numerical prefix to be the score of the article you're reading.

1-5 n

Rate the article and go to the next unread article.

1-5 ,

Rate the article and go to the next unread article with the highest score.

If you want to give the current article a score of 4 and then go to the next article, just type 4 n.

Displaying Predictions

GroupLens makes a prediction for you about how much you will like a news article. The predictions from GroupLens are on a scale from 1 to 5, where 1 is the worst and 5 is the best. You can use the predictions from GroupLens in one of three ways controlled by the variable `gnus-grouplens-override-scoring`.

There are three ways to display predictions in grouplens. You may choose to have the GroupLens scores contribute to, or override the regular gnus scoring mechanism. override is the default; however, some people prefer to see the Gnus scores plus the grouplens scores. To get the separate scoring behavior you need to set `gnus-grouplens-override-scoring` to `'separate`. To have the GroupLens predictions combined with the grouplens scores set it to `'override` and to combine the scores set `gnus-grouplens-override-scoring` to `'combine`. When you use the combine option you will also want to set the values for `grouplens-prediction-offset` and `grouplens-score-scale-factor`.

In either case, GroupLens gives you a few choices for how you would like to see your predictions displayed. The display of predictions is controlled by the `grouplens-prediction-display` variable.

The following are legal values for that variable.

`prediction-spot`

The higher the prediction, the further to the right an `*' is displayed.

`confidence-interval`

A numeric confidence interval.

`prediction-bar`
: The higher the prediction, the longer the bar.

`confidence-bar`
: Numerical confidence.

`confidence-spot`
: The spot gets bigger with more confidence.

`prediction-num`
: Plain-old numeric value.

`confidence-plus-minus`
: Prediction +/i confidence.

GroupLens Variables

`gnus-summary-grouplens-line-format`
: The summary line format used in summary buffers that are GroupLens enhanced. It accepts the same specs as the normal summary line format (see section Summary Buffer Lines). The default is `%U%R%z%l%I%(%[%4L: %-20,20n%]%) %s\n'.

`grouplens-bbb-host`
: Host running the bbbd server. The default is `grouplens.cs.umn.edu'.

`grouplens-bbb-port`
: Port of the host running the bbbd server. The default is 9000.

`grouplens-score-offset`
: Offset the prediction by this value. In other words, subtract the prediction value by this number to arrive at the effective score. The default is 0.

`grouplens-score-scale-factor`
: This variable allows the user to magnify the effect of GroupLens scores. The scale factor is applied after the offset. The default is 1.

Go to the previous, next section.

Go to the previous, next section.

Various

Process/Prefix

Many functions, among them functions for moving, decoding and saving articles, use what is known as the Process/Prefix convention.

This is a method for figuring out what articles that the user wants the command to be performed on.

It goes like this:

If the numeric prefix is N, perform the operation on the next N articles, starting with the current one. If the numeric prefix is negative, perform the operation on the previous N articles, starting with the current one.

If `transient-mark-mode` in non-`nil` and the region is active, all articles in the region will be worked upon.

If there is no numeric prefix, but some articles are marked with the process mark, perform the operation on the articles that are marked with the process mark.

If there is neither a numeric prefix nor any articles marked with the process mark, just perform the operation on the current article.

Quite simple, really, but it needs to be made clear so that surprises are avoided.

One thing that seems to shock & horrify lots of people is that, for instance, 3 d does exactly the same as d d d. Since each d (which marks the current article as read) by default goes to the next unread article after marking, this means that 3 d will mark the next three unread articles as read, no matter what the summary buffer looks like. Set `gnus-summary-goto-unread` to `nil` for a more straightforward action.

Interactive

`gnus-novice-user`

> If this variable is non-`nil`, you are either a newcomer to the World of Usenet, or you are very cautious, which is a nice thing to be, really. You will be given questions of the type "Are you sure you want to do this?" before doing anything dangerous. This is `t` by default.

`gnus-expert-user`

> If this variable is non-`nil`, you will never ever be asked any questions by Gnus. It will simply assume you know what you're doing, no matter how strange.

`gnus-interactive-catchup`

> Require confirmation before catching up a group if non-`nil`. It is `t` by default.

`gnus-interactive-exit`

> Require confirmation before exiting Gnus. This variable is `t` by default.

Formatting Variables

Throughout this manual you've probably noticed lots of variables that are called things like `gnus-group-line-format` and `gnus-summary-mode-line-format`. These control how Gnus is to output lines in the various buffers. There's quite a lot of them. Fortunately, they all use the same syntax, so there's not that much to be annoyed by.

Here's an example format spec (from the group buffer): `%M%S%5y: %(%g%)\n'. We see that it is indeed extremely ugly, and that there are lots of percentages everywhere.

Each `%' element will be replaced by some string or other when the buffer in question is generated. `%5y' means "insert the `y' spec, and pad with spaces to get a 5-character field". Just like a normal format spec, almost.

You can also say `%6,4y', which means that the field will never be more than 6 characters wide and never less than 4 characters wide.

There are also specs for highlighting, and these are shared by all the format variables. Text inside the `%(' and `%)' specifiers will get the special `mouse-face` property set, which means that it will be highlighted (with `gnus-mouse-face`) when you put the mouse pointer over it.

Text inside the `%[' and `%]' specifiers will have their normal faces set using `gnus-face-0`, which is `bold` by default. If you say `%1[' instead, you'll get `gnus-face-1` instead, and so on. Create as many faces as you wish. The same goes for the `mouse-face` specs--you can say `%3(hello%)' to have `hello' mouse-highlighted with `gnus-mouse-face-3`.

Here's an alternative recipe for the group buffer:

```
;; Create three face types.
(setq gnus-face-1 'bold)
(setq gnus-face-3 'italic)

;; We want the article count to be in
;; a bold and green face.  So we create
;; a new face called `my-green-bold'.
(copy-face 'bold 'my-green-bold)
;; Set the color.
(set-face-foreground 'my-green-bold "ForestGreen")
(setq gnus-face-2 'my-green-bold)

;; Set the new & fancy format.
(setq gnus-group-line-format
      "%M%S%3{%5y%}%2[:%] %(%1{%g%}%)\n")
```

I'm sure you'll be able to use this scheme to create totally unreadable and extremely vulgar displays. Have fun!

Currently Gnus uses the following formatting variables: `gnus-group-line-format`, `gnus-summary-line-format`, `gnus-server-line-format`, `gnus-topic-line-format`,

`gnus-group-mode-line-format`, `gnus-summary-mode-line-format`, `gnus-article-mode-line-format`, `gnus-server-mode-line-format`.

Note that the `%(' specs (and friends) do not make any sense on the mode-line variables.

All these format variables can also be arbitrary elisp forms. In that case, they will be `eval`ed to insert the required lines.

Gnus includes a command to help you while creating your own format specs. M-x gnus-update-format will `eval` the current form, update the spec in question and pop you to a buffer where you can examine the resulting lisp code to be run to generate the line.

Windows Configuration

No, there's nothing here about X, so be quiet.

If `gnus-use-full-window` non-`nil`, Gnus will delete all other windows and occupy the entire Emacs screen by itself. It is `t` by default.

`gnus-buffer-configuration` describes how much space each Gnus buffer should be given. Here's an excerpt of this variable:

```
((group (vertical 1.0 (group 1.0 point)
                      (if gnus-carpal (group-carpal 4))))
 (article (vertical 1.0 (summary 0.25 point)
                        (article 1.0))))
```

This is an alist. The key is a symbol that names some action or other. For instance, when displaying the group buffer, the window configuration function will use `group` as the key. A full list of possible names is listed below.

The value (i. e., the split) says how much space each buffer should occupy. To take the `article` split as an example -

```
(article (vertical 1.0 (summary 0.25 point)
                       (article 1.0)))
```

This split says that the summary buffer should occupy 25% of upper half of the screen, and that it is placed over the article buffer. As you may have noticed, 100% + 25% is actually 125% (yup, I saw y'all reaching for that calculator there). However, the special number `1.0` is used to signal that this buffer should soak up all the rest of the space available after the rest of the buffers have taken whatever they need. There should be only one buffer with the `1.0` size spec per split.

Point will be put in the buffer that has the optional third element `point`.

Here's a more complicated example:

```
(article (vertical 1.0 (group 4)
                       (summary 0.25 point)
```


```
                (if gnus-carpal (summary-carpal 4))
                (article 1.0)))
```

If the size spec is an integer instead of a floating point number, then that number will be used to say how many lines a buffer should occupy, not a percentage.

If the split looks like something that can be `eval`ed (to be precise--if the `car` of the split is a function or a subr), this split will be `eval`ed. If the result is non-`nil`, it will be used as a split. This means that there will be three buffers if `gnus-carpal` is `nil`, and four buffers if `gnus-carpal` is non-`nil`.

Not complicated enough for you? Well, try this on for size:

```
(article (horizontal 1.0
             (vertical 0.5
                 (group 1.0)
                 (gnus-carpal 4))
             (vertical 1.0
                 (summary 0.25 point)
                 (summary-carpal 4)
                 (article 1.0))))
```

Whoops. Two buffers with the mystery 100% tag. And what's that `horizontal` thingie?

If the first element in one of the split is `horizontal`, Gnus will split the window horizontally, giving you two windows side-by-side. Inside each of these strips you may carry on all you like in the normal fashion. The number following `horizontal` says what percentage of the screen is to be given to this strip.

For each split, there *must* be one element that has the 100% tag. The splitting is never accurate, and this buffer will eat any leftover lines from the splits.

To be slightly more formal, here's a definition of what a legal split may look like:

```
split       = frame | horizontal | vertical | buffer | form
frame       = "(frame " size *split ")"
horizontal  = "(horizontal " size *split ")"
vertical    = "(vertical " size *split ")"
buffer      = "(" buffer-name " " size *[ "point" ] ")"
size        = number | frame-params
buffer-name = group | article | summary ...
```

The limitations are that the `frame` split can only appear as the top-level split. form should be an Emacs Lisp form that should return a valid split. We see that each split is fully recursive, and may contain any number of `vertical` and `horizontal` splits.

Finding the right sizes can be a bit complicated. No window may be less than `gnus-window-min-height` (default 2) characters high, and all windows must be at least `gnus-window-min-width` (default 1) characters wide. Gnus will try to enforce this before applying the splits. If you want to use the normal Emacs window width/height limit, you can just set these two variables to `nil`.

If you're not familiar with Emacs terminology, `horizontal` and `vertical` splits may work the opp way of what you'd expect. Windows inside a `horizontal` split are shown side-by-side, and windows within a `vertical` split are shown above each other.

If you want to experiment with window placement, a good tip is to call `gnus-configure-frame` directly with a split. This is the function that does all the real work when splitting buffers. Below is a pretty nonsensical configuration with 5 windows; two for the group buffer and three for the article buffer. (I said it was nonsensical.) If you `eval` the statement below, you can get an idea of how that would look straight away, without going through the normal Gnus channels. Play with it until you're satisfied, and then use `gnus-add-configuration` to add your new creation to the buffer configuration list.

```
(gnus-configure-frame
 '(horizontal 1.0
    (vertical 10
      (group 1.0)
      (article 0.3 point))
    (vertical 1.0
      (article 1.0)
      (horizontal 4
        (group 1.0)
        (article 10)))))
```

You might want to have several frames as well. No prob--just use the `frame` split:

```
(gnus-configure-frame
 '(frame 1.0
         (vertical 1.0
                   (summary 0.25 point)
                   (article 1.0))
         (vertical ((height . 5) (width . 15)
                    (user-position . t)
                    (left . -1) (top . 1))
                   (picon 1.0))))
```

This split will result in the familiar summary/article window configuration in the first (or "main") frame, while a small additional frame will be created where picons will be shown. As you can see, instead of the normal `1.0` top-level spec, each additional split should have a frame parameter alist as the size spec. See section 'Frame Parameters' in The GNU Emacs Lisp Reference Manual.

Here's a list of all possible keys for `gnus-buffer-configuration`:

`group`, `summary`, `article`, `server`, `browse`, `group-mail`, `summary-mail`, `summary-reply`, `info`, `summary-faq`, `edit-group`, `edit-server`, `reply`, `reply-yank`, `followup`, `followup-yank`, `edit-score`.

Since the `gnus-buffer-configuration` variable is so long and complicated, there's a function you can use to ease changing the config of a single setting: `gnus-add-configuration`. If, for instance, you want to change the `article` setting, you could say:


```
(gnus-add-configuration
 '(article (vertical 1.0
             (group 4)
             (summary .25 point)
             (article 1.0))))
```

You'd typically stick these `gnus-add-configuration` calls in your `.gnus' file or in some startup hook -- they should be run after Gnus has been loaded.

Compilation

Remember all those line format specification variables? `gnus-summary-line-format`, `gnus-group-line-format`, and so on. Now, Gnus will of course heed whatever these variables are, but, unfortunately, changing them will mean a quite significant slow-down. (The default values of these variables have byte-compiled functions associated with them, while the user-generated versions do not, of course.)

To help with this, you can run M-x gnus-compile after you've fiddled around with the variables and feel that you're (kind of) satisfied. This will result in the new specs being byte-compiled, and you'll get top speed again.

Mode Lines

`gnus-updated-mode-lines` says what buffers should keep their mode lines updated. It is a list of symbols. Supported symbols include `group`, `article`, `summary`, `server`, `browse`, and `tree`. If the corresponding symbol is present, Gnus will keep that mode line updated with information that may be pertinent. If this variable is `nil`, screen refresh may be quicker.

By default, Gnus displays information on the current article in the mode lines of the summary and article buffers. The information Gnus wishes to display (eg. the subject of the article) is often longer than the mode lines, and therefore have to be cut off at some point. The `gnus-mode-non-string-length` variable says how long the other elements on the line is (i.e., the non-info part). If you put additional elements on the mode line (eg. a clock), you should modify this variable:

```
(add-hook 'display-time-hook
          (lambda () (setq gnus-mode-non-string-length
                           (+ 21
                              (if line-number-mode 5 0)
                              (if column-number-mode 4 0)
                              (length display-time-string)))))
```

If this variable is `nil` (which is the default), the mode line strings won't be chopped off, and they won't be padded either.

Highlighting and Menus

The `gnus-visual` variable controls most of the prettifying Gnus aspects. If `nil`, Gnus won't attempt to create menus or use fancy colors or fonts. This will also inhibit loading the `` `gnus-vis.el' `` file.

This variable can be a list of visual properties that are enabled. The following elements are legal, and are all included by default:

`group-highlight`
: Do highlights in the group buffer.

`summary-highlight`
: Do highlights in the summary buffer.

`article-highlight`
: Do highlights in the article buffer.

`highlight`
: Turn on highlighting in all buffers.

`group-menu`
: Create menus in the group buffer.

`summary-menu`
: Create menus in the summary buffers.

`article-menu`
: Create menus in the article buffer.

`browse-menu`
: Create menus in the browse buffer.

`server-menu`
: Create menus in the server buffer.

`score-menu`
: Create menus in the score buffers.

`menu`
: Create menus in all buffers.

So if you only want highlighting in the article buffer and menus in all buffers, you could say something like:

```
(setq gnus-visual '(article-highlight menu))
```

If you want only highlighting and no menus whatsoever, you'd say:

```
(setq gnus-visual '(highlight))
```

If `gnus-visual` is `t`, highlighting and menus will be used in all Gnus buffers.

Other general variables that influence the look of all buffers include:

`gnus-mouse-face`

This is the face (i.e., font) used for mouse highlighting in Gnus. No mouse highlights will be done if `gnus-visual` is `nil`.

`gnus-display-type`

This variable is symbol indicating the display type Emacs is running under. The symbol should be one of `color`, `grayscale` or `mono`. If Gnus guesses this display attribute wrongly, either set this variable in your `` `~/.emacs' `` or set the resource `Emacs.displayType` in your `` `~/.Xdefaults' ``.

`gnus-background-mode`

This is a symbol indicating the Emacs background brightness. The symbol should be one of `light` or `dark`. If Gnus guesses this frame attribute wrongly, either set this variable in your `` `~/.emacs' `` or set the resource `Emacs.backgroundMode` in your `` `~/.Xdefaults' ``. `gnus-display-type'.

There are hooks associated with the creation of all the different menus:

`gnus-article-menu-hook`

Hook called after creating the article mode menu.

`gnus-group-menu-hook`

Hook called after creating the group mode menu.

`gnus-summary-menu-hook`

Hook called after creating the summary mode menu.

`gnus-server-menu-hook`

Hook called after creating the server mode menu.

`gnus-browse-menu-hook`

Hook called after creating the browse mode menu.

`gnus-score-menu-hook`

Hook called after creating the score mode menu.

Buttons

Those new-fangled mouse contraptions is very popular with the young, hep kids who don't want to learn the proper way to do things these days. Why, I remember way back in the summer of '89, when I was using Emacs on a Tops 20 system. Three hundred users on one single machine, and every user was running Simula compilers. Bah!

Right.

Well, you can make Gnus display bufferfuls of buttons you can click to do anything by setting `gnus-carpal` to `t`. Pretty simple, really. Tell the chiropractor I sent you.

`gnus-carpal-mode-hook`

Hook run in all carpal mode buffers.

`gnus-carpal-button-face`

Face used on buttons.

`gnus-carpal-header-face`

Face used on carpal buffer headers.

`gnus-carpal-group-buffer-buttons`

Buttons in the group buffer.

`gnus-carpal-summary-buffer-buttons`

Buttons in the summary buffer.

`gnus-carpal-server-buffer-buttons`

Buttons in the server buffer.

`gnus-carpal-browse-buffer-buttons`

Buttons in the browse buffer.

All the `buttons` variables are lists. The elements in these list is either a cons cell where the car contains a text to be displayed and the cdr contains a function symbol, or a simple string.

Daemons

Gnus, being larger than any program ever written (allegedly), does lots of strange stuff that you may wish to have done while you're not present. For instance, you may want it to check for new mail once in a while. Or you may want it to close down all connections to all servers when you leave Emacs idle. And stuff like that.

Gnus will let you do stuff like that by defining various handlers. Each handler consists of three elements: A function, a time, and an idle parameter.

Here's an example of a handler that closes connections when Emacs has been idle for thirty minutes:

```
(gnus-demon-close-connections nil 30)
```

Here's a handler that scans for PGP headers every hour when Emacs is idle:

```
(gnus-demon-scan-pgp 60 t)
```

This time parameter and than idle parameter works together in a strange, but wonderful fashion. Basically, if idle is `nil`, then the function will be called every time minutes.

If idle is `t`, then the function will be called after time minutes only if Emacs is idle. So if Emacs is never idle, the function will never be called. But once Emacs goes idle, the function will be called every time minutes.

If idle is a number and time is a number, the function will be called every time minutes only when Emacs has been idle for idle minutes.

If idle is a number and time is `nil`, the function will be called once every time Emacs has been idle for idle minutes.

And if time is a string, it should look like `07:31', and the function will then be called once every day somewhere near that time. Modified by the idle parameter, of course.

(When I say "minute" here, I really mean `gnus-demon-timestep` seconds. This is `60` by default. If you change that variable, all the timings in the handlers will be affected.)

To set the whole thing in motion, though, you have to set `gnus-use-demon` to `t`.

So, if you want to add a handler, you could put something like this in your `.gnus' file:

```
(gnus-demon-add-handler 'gnus-demon-close-connections nil 30)
```

Some ready-made functions to do this has been created: `gnus-demon-add-nocem`, `gnus-demon-add-disconnection`, and `gnus-demon-add-scanmail`. Just put those functions in your `.gnus' if you want those abilities.

If you add handlers to `gnus-demon-handlers` directly, you should run `gnus-demon-init` to make the changes take hold. To cancel all daemons, you can use the `gnus-demon-cancel` function.

Note that adding daemons can be pretty naughty if you overdo it. Adding functions that scan all news and mail from all servers every two seconds is a sure-fire way of getting booted off any respectable system. So behave.

NoCeM

Spamming is posting the same article lots and lots of times. Spamming is bad. Spamming is evil.

Spamming is usually canceled within a day or so by various anti-spamming agencies. These agencies usually also send out NoCeM messages. NoCeM is pronounced "no see-'em", and means what the name implies--these are messages that make the offending articles, like, go away.

What use are these NoCeM messages if the articles are canceled anyway? Some sites do not honor cancel messages and some sites just honor cancels from a select few people. Then you may wish to make use of the NoCeM messages, which are distributed in the `alt.nocem.misc' newsgroup.

Gnus can read and parse the messages in this group automatically, and this will make spam disappear.

There are some variables to customize, of course:

`gnus-use-nocem`

Set this variable to `t` to set the ball rolling. It is `nil` by default.

`gnus-nocem-groups`

Gnus will look for NoCeM messages in the groups in this list. The default is `("alt.nocem.misc" "news.admin.net-abuse.announce")`.

`gnus-nocem-issuers`

There are many people issuing NoCeM messages. This list says what people you want to listen to. The default is `("Automoose-1" "clewis@ferret.ocunix.on.ca;" "jem@xpat.com;" "red@redpoll.mrfs.oh.us (Richard E. Depew)")`; fine, upstanding citizens all of them.

Known despammers that you can put in this list include:

`clewis@ferret.ocunix.on.ca;'

Chris Lewis--Major Canadian despammer who has probably canceled more usenet abuse than anybody else.

`Automoose-1'

The CancelMoose[tm] on autopilot. The CancelMoose[tm] is reputed to be Norwegian, and was the person(s) who invented NoCeM.

`jem@xpat.com;'

Jem--Korean despammer who is getting very busy these days.

`red@redpoll.mrfs.oh.us (Richard E. Depew)'

Richard E. Depew--lone American despammer. He mostly cancels binary postings to non-binary groups and removes spews (regurgitated articles).

You do not have to heed NoCeM messages from all these people--just the ones you want to listen to.

- gnus-nocem-directory This is where Gnus will store its NoCeM cache files. The default is `~/News/NoCeM/'.
- gnus-nocem-expiry-wait The number of days before removing old NoCeM entries from the cache. The default is 15. If you make it shorter Gnus will be faster, but you might then see old spam.

Picons

So... You want to slow down your news reader even more! This is a good way to do so. Its also a great way to impress people staring over your shoulder as you read news.

Picon Basics

What are Picons? To quote directly from the Picons Web site (`http://www.cs.indiana.edu/picons/ftp/index.html'):

> Picons is short for "personal icons". They're small, constrained images used to represent users and domains on the net, organized into databases so that the appropriate image for a given e-mail address can be found. Besides users and domains, there are picon databases for Usenet newsgroups and weather forecasts. The picons are in either monochrome `XBM` format or color `XPM` and `GIF` formats.

Please see the above mentioned web site for instructions on obtaining and installing the picons databases, or the following ftp site: `http://www.cs.indiana.edu/picons/ftp/index.html'.

Gnus expects picons to be installed into a location pointed to by `gnus-picons-database`.

Picon Requirements

To use have Gnus display Picons for you, you must be running XEmacs 19.13 or greater since all other versions of Emacs aren't yet able to display images.

Additionally, you must have `xpm` support compiled into XEmacs.

If you want to display faces from `X-Face` headers, you must have the `netpbm` utilities installed, or munge the `gnus-picons-convert-x-face` variable to use something else.

Easy Picons

To enable displaying picons, simply put the following line in your `~/.gnus' file and start Gnus.

```
(setq gnus-use-picons t)
(add-hook 'gnus-article-display-hook 'gnus-article-display-picons t)
(add-hook 'gnus-summary-prepare-hook 'gnus-group-display-picons t)
(add-hook 'gnus-article-display-hook 'gnus-picons-article-display-x-face)
```

Hard Picons

Gnus can display picons for you as you enter and leave groups and articles. It knows how to interact with three sections of the picons database. Namely, it can display the picons newsgroup pictures, author's face picture(s), and the authors domain. To enable this feature, you need to first decide where to display them.

`gnus-picons-display-where`

> Where the picon images should be displayed. It is `picons` by default (which by default maps to the buffer `*Picons*'). Other valid places could be `article`, `summary`, or `'"*scratch*"' for all I care. Just make sure that you've made the buffer visible using the standard Gnus window configuration routines -- See section Windows Configuration.

Note: If you set `gnus-use-picons` to `t`, it will set up your window configuration for you to include the `picons` buffer.

Now that you've made that decision, you need to add the following functions to the appropriate hooks so these pictures will get displayed at the right time.

`gnus-article-display-picons`

> Looks up and display the picons for the author and the author's domain in the `gnus-picons-display-where` buffer. Should be added to the `gnus-article-display-hook`.

`gnus-group-display-picons`

> Displays picons representing the current group. This function should be added to the `gnus-summary-prepare-hook` or to the `gnus-article-display-hook` if `gnus-picons-display-where` is set to `article`.

`gnus-picons-article-display-x-face`

> Decodes and displays the X-Face header if present. This function should be added to `gnus-article-display-hook`.

Note: You must append them to the hook, so make sure to specify 't' to the append flag of `add-hook`:

```
(add-hook 'gnus-article-display-hook 'gnus-article-display-picons t)
```

Picon Configuration

The following variables offer further control over how things are done, where things are located, and other useless stuff you really don't need to worry about.

`gnus-picons-database`

The location of the picons database. Should point to a directory containing the `` `news' ``, `` `domains' ``, `` `users' `` (and so on) subdirectories. Defaults to `` `/usr/local/faces' ``.

`gnus-picons-news-directory`

Sub-directory of the faces database containing the icons for newsgroups.

`gnus-picons-user-directories`

List of subdirectories to search in `gnus-picons-database` for user faces. Defaults to `("local" "users" "usenix" "misc/MISC")`.

`gnus-picons-domain-directories`

List of subdirectories to search in `gnus-picons-database` for domain name faces. Defaults to `("domains")`. Some people may want to add `unknown' to this list.

`gnus-picons-convert-x-face`

The command to use to convert the `X-Face` header to an X bitmap (`xbm`). Defaults to `(format "{ echo '/* Width=48, Height=48 */'; uncompface; } | icontopbm | pbmtoxbm > %s" gnus-picons-x-face-file-name)`

`gnus-picons-x-face-file-name`

Names a temporary file to store the `X-Face` bitmap in. Defaults to `(format "/tmp/picon-xface.%s.xbm" (user-login-name))`.

`gnus-picons-buffer`

The name of the buffer that `picons` points to. Defaults to `*Icon Buffer*'.

Various Various

`gnus-verbose`

This variable is an integer between zero and ten. The higher the value, the more messages will be displayed. If this variable is zero, Gnus will never flash any messages, if it is seven (which is the default), most important messages will be shown, and if it is ten, Gnus won't ever shut up, but will flash so many messages it will make your head swim.

`gnus-verbose-backends`

This variable works the same way as `gnus-verbose`, but it applies to the Gnus backends instead of Gnus proper.

`nnheader-max-head-length`

When the backends read straight heads of articles, they all try to read as little as possible. This variable (default `4096`) specifies the absolute max length the backends will try to read before giving up on finding a separator line between the head and the body. If this variable is `nil`, there is no upper read bound. If it is `t`, the backends won't try to read the articles piece by piece, but read the entire articles. This makes sense with some versions of `ange-ftp`.

`nnheader-file-name-translation-alist`

This is an alist that says how to translate characters in file names. For instance, if `:' is illegal as a file character in file names on your system (you OS/2 user you), you could say something like:

```
(setq nnheader-file-name-translation-alist
      '((?: . ?_)))
```

In fact, this is the default value for this variable on OS/2 and MS Windows (phooey) systems.

`gnus-hidden-properties`

This is a list of properties to use to hide "invisible" text. It is `(invisible t intangible t)` by default on most systems, which makes invisible text invisible and intangible.

`gnus-parse-headers-hook`

A hook called before parsing headers. It can be used, for instance, to gather statistics on the headers fetched, or perhaps you'd like to prune some headers. I don't see why you'd want that, though.

Go to the previous, next section.

Go to the previous, next section.

The End

Well, that's the manual--you can get on with your life now. Keep in touch. Say hello to your cats from me.

My **ghod**---I just can't stand goodbyes. Sniffle.

Ol' Charles Reznikoff said it pretty well, so I leave the floor to him:

> **Te Deum**
>
> Not because of victories
> I sing,
> having none,
> but for the common sunshine,
> the breeze,
> the largess of the spring.
>
> Not for victory
> but for the day's work done
> as well as I was able;
> not for a seat upon the dais
> but at the common table.

Go to the previous, next section.

Go to the previous, next section.

TOC

Appendices

History

GNUS was written by Masanobu UMEDA. When autumn crept up in '94, Lars Magne Ingebrigtsen grew bored and decided to rewrite Gnus.

If you want to investigate the person responsible for this outrage, you can point your (feh!) web browser to `http://www.ifi.uio.no/~larsi/'. This is also the primary distribution point for the new and spiffy versions of Gnus, and is known as The Site That Destroys Newsrcs And Drives People Mad.

During the first extended alpha period of development, the new Gnus was called "(ding) Gnus". (ding), is, of course, short for ding is not Gnus, which is a total and utter lie, but who cares? (Besides, the "Gnus" in this abbreviation should probably be pronounced "news" as UMEDA intended, which makes it a more appropriate name, don't you think?)

In any case, after spending all that energy on coming up with a new and spunky name, we decided that the name was *too* spunky, so we renamed it back again to "Gnus". But in mixed case. "Gnus" vs. "GNUS". New vs. old.

The first "proper" release of Gnus 5 was done in November 1995 when it was included in the Emacs 19.30 distribution.

In May 1996 the next Gnus generation (aka. "September Gnus") was released under the name "Gnus 5.2".

Why?

What's the point of Gnus?

I want to provide a "rad", "happening", "way cool" and "hep" newsreader, that lets you do anything you can think of. That was my original motivation, but while working on Gnus, it has become clear to me that this generation of newsreaders really belong in the stone age. Newsreaders haven't developed much since the infancy of the net. If the volume continues to rise with the current rate of increase, all current newsreaders will be pretty much useless. How do you deal with newsgroups that have thousands of new articles each day? How do you keep track of millions of people who post?

Gnus offers no real solutions to these questions, but I would very much like to see Gnus being used as a testing ground for new methods of reading and fetching news. Expanding on UMEDA-san's wise decision to separate the newsreader from the backends, Gnus now offers a simple interface for anybody who wants to write new backends for fetching mail and news from different sources. I have added hooks for customizations everywhere I could imagine useful. By doing so, I'm inviting every one of you to explore and invent.

May Gnus never be complete. C-u 100 M-x hail-emacs.

Compatibility

Gnus was designed to be fully compatible with GNUS. Almost all key bindings have been kept. More key bindings have been added, of course, but only in one or two obscure cases have old bindings been changed.

Our motto is:

> In a cloud bones of steel.

All commands have kept their names. Some internal functions have changed their names.

The `gnus-uu` package has changed drastically. see section Decoding Articles.

One major compatibility question is the presence of several summary buffers. All variables that are relevant while reading a group are buffer-local to the summary buffer they belong in. Although many important variables have their values copied into their global counterparts whenever a command is executed in the summary buffer, this change might lead to incorrect values being used unless you are careful.

All code that relies on knowledge of GNUS internals will probably fail. To take two examples: Sorting `gnus-newsrc-alist` (or changing it in any way, as a matter of fact) is strictly verboten. Gnus maintains a hash table that points to the entries in this alist (which speeds up many functions), and changing the alist directly will lead to peculiar results.

Old hilit19 code does not work at all. In fact, you should probably remove all hilit code from all Gnus hooks (`gnus-group-prepare-hook` and `gnus-summary-prepare-hook`). Gnus provides various integrated functions for highlighting. These are faster and more accurate. To make life easier for everybody, Gnus will by default remove all hilit calls from all hilit hooks. Uncleanliness! Away!

Packages like `expire-kill` will no longer work. As a matter of fact, you should probably remove all old GNUS packages (and other code) when you start using Gnus. More likely than not, Gnus already does what you have written code to make GNUS do. (Snicker.)

Even though old methods of doing things are still supported, only the new methods are documented in this manual. If you detect a new method of doing something while reading this manual, that does not mean you have to stop doing it the old way.

Gnus understands all GNUS startup files.

Overall, a casual user who hasn't written much code that depends on GNUS internals should suffer no problems. If problems occur, please let me know by issuing that magic command M-x gnus-bug.

Conformity

No rebels without a clue here, ma'am. We conform to all standards known to (wo)man. Except for those standards and/or conventions we disagree with, of course.

RFC 822

There are no known breaches of this standard.

RFC 1036

There are no known breaches of this standard, either.

Usenet Seal of Approval

Gnus hasn't been formally through the Seal process, but I have read through the Seal text and I think Gnus would pass.

Son-of-RFC 1036

We do have some breaches to this one.

MIME

Gnus does no MIME handling, and this standard-to-be seems to think that MIME is the bees' knees, so we have major breakage here.

X-Newsreader

This is considered to be a "vanity header", while I consider it to be consumer information. After seeing so many badly formatted articles coming from `tin` and `Netscape` I know not to use either of those for posting articles. I would not have known that if it wasn't for the `X-Newsreader` header.

References

Gnus does line breaking on this header. I infer from RFC1036 that being conservative in what you output is not creating 5000-character lines, so it seems like a good idea to me. However, this standard-to-be says that whitespace in the `References` header is to be preserved, so... It doesn't matter one way or the other to Gnus, so if somebody tells me what The Way is, I'll change it. Or not.

If you ever notice Gnus acting non-compliantly with regards to the texts mentioned above, don't hesitate to drop a note to Gnus Towers and let us know.

Emacsen

Gnus should work on :

- Emacs 19.30 and up.
- XEmacs 19.13 and up.
- Mule versions based on Emacs 19.30 and up.

Gnus will absolutely not work on any Emacsen older than that. Not reliably, at least.

There are some vague differences between Gnus on the various platforms:

- The mouse-face on Gnus lines under Emacs and Mule is delimited to certain parts of the lines while they cover the entire line under XEmacs.
- The same with current-article marking--XEmacs puts an underline under the entire summary line while Emacs and Mule are nicer

and kinder.

- XEmacs features more graphics--a logo and a toolbar.
- Citation highlighting us better under Emacs and Mule than under XEmacs.
- Emacs 19.26-19.28 have tangible hidden headers, which can be a bit confusing.

Contributors

The new Gnus version couldn't have been done without the help of all the people on the (ding) mailing list. Every day for over a year I have gotten billions of nice bug reports from them, filling me with joy, every single one of them. Smooches. The people on the list have been tried beyond endurance, what with my "oh, that's a neat idea <type type>, yup, I'll release it right away <ship off> no wait, that doesn't work at all <type type>, yup, I'll ship that one off right away <ship off> no, wait, that absolutely does not work" policy for releases. Micro$oft--bah. Amateurs. I'm *much* worse. (Or is that "worser"? "much worser"? "worsest"?)

I would like to take this opportunity to thank the Academy for... oops, wrong show.

- Masanobu UMEDA The writer of the original GNUS.
- Per Abrahamsen Custom, scoring, highlighting and SOUP code (as well as numerous other things).
- Luis Fernandes Design and graphics.
- Wes Hardaker `gnus-picon.el' and the manual section on picons (see section Picons).
- Brad Miller `gnus-gl.el' and the GroupLens manual section (see section GroupLens).
- Sudish Joseph Innumerable bug fixes.
- Ilja Weis `gnus-topic.el'.
- Steven L. Baur Lots and lots of bugs detections and fixes.
- Vladimir Alexiev The refcard and reference booklets.
- Felix Lee & JWZ I stole some pieces from the XGnus distribution by Felix Lee and JWZ.
- Scott Byer `nnfolder.el' enhancements & rewrite.
- Peter Mutsaers Orphan article scoring code.
- Ken Raeburn POP mail support.
- Hallvard B Furuseth Various bits and pieces, especially dealing with .newsrc files.
- Brian Edmonds `gnus-bbdb.el'.
- Ricardo Nassif and Mark Borges Proof-reading.
- Kevin Davidson Came up with the name ding, so blame him.

Peter Arius, Stainless Steel Rat, Ulrik Dickow, Jack Vinson, Daniel Quinlan, Frank D. Cringle, Geoffrey T. Dairiki, Fabrice Popineau and Andrew Eskilsson have all contributed code and suggestions.

New Features

- The look of all buffers can be changed by setting format-like variables (see section Group Buffer Format and see section Summary Buffer Format).
- Local spool and several NNTP servers can be used at once (see section Select Methods).
- You can combine groups into virtual groups (see section Virtual Groups).
- You can read a number of different mail formats (see section Getting Mail). All the mail backends implement a convenient mail expiry scheme (see section Expiring Mail).
- Gnus can use various strategies for gathering threads that have lost their roots (thereby gathering loose sub-threads into one thread) or it can go back and retrieve enough headers to build a complete thread (see section Customizing Threading).
- Killed groups can be displayed in the group buffer, and you can read them as well (see section Listing Groups).
- Gnus can do partial group updates--you do not have to retrieve the entire active file just to check for new articles in a few groups (see section The Active File).
- Gnus implements a sliding scale of subscribedness to groups (see section Group Levels).
- You can score articles according to any number of criteria (see section Scoring). You can even get Gnus to find out how to score articles for you (see section Adaptive Scoring).

- Gnus maintains a dribble buffer that is auto-saved the normal Emacs manner, so it should be difficult to lose much data on v you have read if your machine should go down (see section Auto Save).
- Gnus now has its own startup file (`.gnus') to avoid cluttering up the `.emacs' file.
- You can set the process mark on both groups and articles and perform operations on all the marked items (see section Process/Prefix).
- You can grep through a subset of groups and create a group from the results (see section Kibozed Groups).
- You can list subsets of groups according to, well, anything (see section Listing Groups).
- You can browse foreign servers and subscribe to groups from those servers (see section Browse Foreign Server).
- Gnus can fetch articles asynchronously on a second connection to the server (see section Asynchronous Article Fetching).
- You can cache articles locally (see section Article Caching).
- The uudecode functions have been expanded and generalized (see section Decoding Articles).
- You can still post uuencoded articles, which was a little-known feature of GNUS' past (see section Uuencoding and Posting).
- Fetching parents (and other articles) now actually works without glitches (see section Finding the Parent).
- Gnus can fetch FAQs and group descriptions (see section Group Information).
- Digests (and other files) can be used as the basis for groups (see section Document Groups).
- Articles can be highlighted and customized (see section Customizing Articles).
- URLs and other external references can be buttonized (see section Article Buttons).
- You can do lots of strange stuff with the Gnus window & frame configuration (see section Windows Configuration).
- You can click on buttons instead of using the keyboard (see section Buttons).
- Gnus can use NoCeM files to weed out spam (see section NoCeM).

This is, of course, just a *short* overview of the *most* important new features. No, really. There are tons more. Yes, we have feeping creaturism in full effect, but nothing too gratuitous, I would hope.

Newest Features

Also known as the todo list. Sure to be implemented before the next millennium.

Be afraid. Be very afraid.

- Native MIME support is something that should be done.
- A better and simpler method for specifying mail composing methods.
- Allow posting through mail-to-news gateways.
- Really do unbinhexing.

And much, much, much more. There is more to come than has already been implemented. (But that's always true, isn't it?)

<URL:http://www.ifi.uio.no/~larsi/sgnus/todo> is where the actual up-to-the-second todo list is located, so if you're really curious, you could point your Web browser over that-a-way.

Terminology

news

This is what you are supposed to use this thing for--reading news. News is generally fetched from a nearby NNTP server, and is generally publicly available to everybody. If you post news, the entire world is likely to read just what you have written, and they'll all snigger mischievously. Behind your back.

mail

Everything that's delivered to you personally is mail. Some news/mail readers (like Gnus) blur the distinction between mail and news, but there is a difference. Mail is private. News is public. Mailing is not posting, and replying is not following up.

reply

Send a mail to the person who has written what you are reading.

follow up

Post an article to the current newsgroup responding to the article you are reading.

backend

Gnus gets fed articles from a number of backends, both news and mail backends. Gnus does not handle the underlying media, so to speak--this is all done by the backends.

native

Gnus will always use one method (and backend) as the native, or default, way of getting news.

foreign

You can also have any number of foreign groups active at the same time. These are groups that use different backends for getting news.

secondary

Secondary backends are somewhere half-way between being native and being foreign, but they mostly act like they are native.

article

A nessage that has been posted as news.

mail message

A message that has been mailed.

message

A mail message or news article

head

The top part of a message, where administrative information (etc.) is put.

body

The rest of an article. Everything that is not in the head is in the body.

header

A line from the head of an article.

headers

A collection of such lines, or a collection of heads. Or even a collection of NOV lines.

NOV

When Gnus enters a group, it asks the backend for the headers of all unread articles in the group. Most servers support the News OverView format, which is more compact and much faster to read and parse than the normal HEAD format.

level

Each group is subscribed at some level or other (1-9). The ones that have a lower level are "more" subscribed than the groups with a higher level. In fact, groups on levels 1-5 are considered subscribed; 6-7 are unsubscribed; 8 are zombies; and 9 are killed. Commands for listing groups and scanning for new articles will all use the numeric prefix as working level.

killed groups

No information on killed groups is stored or updated, which makes killed groups much easier to handle than subscribed groups.

zombie groups

Just like killed groups, only slightly less dead.

active file

The news server has to keep track of what articles it carries, and what groups exist. All this information in stored in the active file, which is rather large, as you might surmise.

bogus groups

A group that exists in the `.newsrc' file, but isn't known to the server (i. e., it isn't in the active file), is a *bogus group*. This means that the group probably doesn't exist (any more).

server

A machine than one can connect to and get news (or mail) from.

select method

A structure that specifies the backend, the server and the virtual server parameters.

virtual server

A named select method. Since a select methods defines all there is to know about connecting to a (physical) server, taking the who things as a whole is a virtual server.

Customization

All variables are properly documented elsewhere in this manual. This section is designed to give general pointers on how to customize Gnus for some quite common situations.

Slow/Expensive NNTP Connection

If you run Emacs on a machine locally, and get your news from a machine over some very thin strings, you want to cut down on the amount of data Gnus has to get from the NNTP server.

`gnus-read-active-file`

Set this to `nil`, which will inhibit Gnus from requesting the entire active file from the server. This file is often v. large. You also have to set `gnus-check-new-news` and `gnus-check-bogus-newsgroups` to `nil` to make sure that Gnus doesn't suddenly decide to fetch the active file anyway.

`gnus-nov-is-evil`

This one has to be `nil`. If not, grabbing article headers from the NNTP server will not be very fast. Not all NNTP servers support XOVER; Gnus will detect this by itself.

Slow Terminal Connection

Let's say you use your home computer for dialing up the system that runs Emacs and Gnus. If your modem is slow, you want to reduce the amount of data that is sent over the wires as much as possible.

`gnus-auto-center-summary`

Set this to `nil` to inhibit Gnus from re-centering the summary buffer all the time. If it is `vertical`, do only vertical re-centering. If it is neither `nil` nor `vertical`, do both horizontal and vertical recentering.

`gnus-visible-headers`

Cut down on the headers that are included in the articles to the minimum. You can, in fact, make do without them altogether--most of the useful data is in the summary buffer, anyway. Set this variable to `^NEVVVVER' or `From:', or whatever you feel you need.

`gnus-article-display-hook`

Set this hook to all the available hiding commands:

```
(setq gnus-article-display-hook
      '(gnus-article-hide-headers gnus-article-hide-signature
        gnus-article-hide-citation))
```

`gnus-use-full-window`

By setting this to `nil`, you can make all the windows smaller. While this doesn't really cut down much generally, it means that you have to see smaller portions of articles before deciding that you didn't want to read them anyway.

`gnus-thread-hide-subtree`

If this is non-`nil`, all threads in the summary buffer will be hidden initially.

`gnus-updated-mode-lines`

If this is `nil`, Gnus will not put information in the buffer mode lines, which might save some time.

Little Disk Space

The startup files can get rather large, so you may want to cut their sizes a bit if you are running out of space.

`gnus-save-newsrc-file`

If this is `nil`, Gnus will never save `.newsrc'---it will only save `.newsrc.eld'. This means that you will not be able to use any other newsreaders than Gnus. This variable is `t` by default.

`gnus-save-killed-list`

If this is `nil`, Gnus will not save the list of dead groups. You should also set `gnus-check-new-newsgroups` to `ask-server` and `gnus-check-bogus-newsgroups` to `nil` if you set this variable to `nil`. This variable is `t` by default.

Slow Machine

If you have a slow machine, or are just really impatient, there are a few things you can do to make Gnus run faster.

Set`gnus-check-new-newsgroups` and `gnus-check-bogus-newsgroups` to `nil` to make startup faster.

Set `gnus-show-threads`, `gnus-use-cross-reference` and `gnus-nov-is-evil` to `nil` to make entering and exiting the summary buffer faster.

Set `gnus-article-display-hook` to `nil` to make article processing a bit faster.

Troubleshooting

Gnus works *so* well straight out of the box--I can't imagine any problems, really.

Ahem.

1. Make sure your computer is switched on.
2. Make sure that you really load the current Gnus version. If you have been running GNUS, you need to exit Emacs and start it up again before Gnus will work.
3. Try doing an M-x gnus-version. If you get something that looks like `Gnus v5.46; nntp 4.0' you have the right files loaded. If, on the other hand, you get something like `NNTP 3.x' or `nntp flee', you have some old `` `.el' `` files lying around. Delete these.
4. Read the help group (G h in the group buffer) for a FAQ and a how-to.

If all else fails, report the problem as a bug.

If you find a bug in Gnus, you can report it with the M-x gnus-bug command. M-x set-variable RET debug-on-error RET t RET, and send me the backtrace. I will fix bugs, but I can only fix them if you send me a precise description as to how to reproduce the bug.

You really can never be too detailed in a bug report. Always use the M-x gnus-bug command when you make bug reports, even if it creates a 10Kb mail each time you use it, and even if you have sent me your environment 500 times before. I don't care. I want the full info each time.

It is also important to remember that I have no memory whatsoever. If you send a bug report, and I send you a reply, and then you send back just "No, it's not! Moron!", I will have no idea what you are insulting me about. Always over-explain everything. It's much easier for all of us--if I don't have all the information I need, I will just mail you and ask for more info, and everything takes more time.

If the problem you're seeing is very visual, and you can't quite explain it, copy the Emacs window to a file (with `xwd`, for instance), put it somewhere it can be reached, and include the URL of the picture in the bug report.a

If you just need help, you are better off asking on `gnu.emacs.gnus'. I'm not very helpful.

You can also ask on the ding mailing list---`ding@ifi.uio.no'. Write to `ding-request@ifi.uio.no' to subscribe.

A Programmer's Guide to Gnus

It is my hope that other people will figure out smart stuff that Gnus can do, and that other people will write those smart things as well. To facilitate that I thought it would be a good idea to describe the inner workings of Gnus. And some of the not-so-inner workings, while I'm at it.

You can never expect the internals of a program not to change, but I will be defining (in some details) the interface between Gnus and its backends (this is written in stone), the format of the score files (ditto), data structures (some are less likely to change than others) and general method of operations.

Backend Interface

Gnus doesn't know anything about NNTP, spools, mail or virtual groups. It only knows how to talk to virtual servers. A virtual server is a backend and some backend variables. As examples of the first, we have `nntp`, `nnspool` and `nnmbox`. As examples of the latter we have `nntp-port-number` and `nnmbox-directory`.

When Gnus asks for information from a backend--say `nntp`---on something, it will normally include a virtual server name in the function parameters. (If not, the backend should use the "current" virtual server.) For instance, `nntp-request-list` takes a virtual

server as its only (optional) parameter. If this virtual server hasn't been opened, the function should fail.

Note that a virtual server name has no relation to some physical server name. Take this example:

```
(nntp "odd-one"
      (nntp-address "ifi.uio.no")
      (nntp-port-number 4324))
```

Here the virtual server name is `odd-one' while the name of the physical server is `ifi.uio.no'.

The backends should be able to switch between several virtual servers. The standard backends implement this by keeping an alist of virtual server environments that it pulls down/pushes up when needed.

There are two groups of interface functions: required functions, which must be present, and optional functions, which Gnus will always check whether are present before attempting to call.

All these functions are expected to return data in the buffer `nntp-server-buffer` (` *nntpd*'), which is somewhat unfortunately named, but we'll have to live with it. When I talk about "resulting data", I always refer to the data in that buffer. When I talk about "return value", I talk about the function value returned by the function call.

Some backends could be said to be server-forming backends, and some might be said to not be. The latter are backends that generally only operate on one group at a time, and have no concept of "server" -- they have a group, and they deliver info on that group and nothing more.

In the examples and definitions I will refer to the imaginary backend `nnchoke`.

Required Backend Functions

`(nnchoke-retrieve-headers ARTICLES &optional GROUP SERVER FETCH-OLD)`

articles is either a range of article numbers or a list of `Message-ID`s. Current backends do not fully support either--only sequences (lists) of article numbers, and most backends do not support retrieval of `Message-ID`s. But they should try for both.

The result data should either be HEADs or NOV lines, and the result value should either be `headers` or `nov` to reflect this. This might later be expanded to `various`, which will be a mixture of HEADs and NOV lines, but this is currently not supported by Gnus.

If fetch-old is non-`nil` it says to try to fetch "extra headers, in some meaning of the word. This is generally done by fetching (at most) fetch-old extra headers less than the smallest article number in `articles`, and fill in the gaps as well. The presence of this parameter can be ignored if the backend finds it cumbersome to follow the request. If this is non-`nil` and not a number, do maximum fetches.

Here's an example HEAD:

```
221 1056 Article retrieved.
Path: ifi.uio.no!sturles
From: sturles@ifi.uio.no (Sturle Sunde)
Newsgroups: ifi.discussion
Subject: Re: Something very droll
Date: 27 Oct 1994 14:02:57 +0100
Organization: Dept. of Informatics, University of Oslo, Norway
Lines: 26
Message-ID: <38o8e1$a0o@holmenkollen.ifi.uio.no>
References: <38jdmq$4qu@visbur.ifi.uio.no>
NNTP-Posting-Host: holmenkollen.ifi.uio.no
.
```

So a `headers` return value would imply that there's a number of these in the data buffer.

Here's a BNF definition of such a buffer:

```
headers        = *head
head           = error / valid-head
```


```
error-message  = [ "4" / "5" ] 2number " " <error message> eol
valid-head     = valid-message *header "." eol
valid-message  = "221 " <number> " Article retrieved." eol
header         = <text> eol
```

If the return value is `nov`, the data buffer should contain network overview database lines. These are basically fields separated by tabs.

```
nov-buffer = *nov-line
nov-line   = 8*9 [ field <TAB> ] eol
field      = <text except TAB>
```

For a closer explanation what should be in those fields, see section Headers.

`(nnchoke-open-server SERVER &optional DEFINITIONS)`

server is here the virtual server name. definitions is a list of `(VARIABLE VALUE)` pairs that defines this virtual server.

If the server can't be opened, no error should be signaled. The backend may then choose to refuse further attempts at connecting to this server. In fact, it should do so.

If the server is opened already, this function should return a non-`nil` value. There should be no data returned.

`(nnchoke-close-server &optional SERVER)`

Close connection to server and free all resources connected to it. Return `nil` if the server couldn't be closed for some reason.

There should be no data returned.

`(nnchoke-request-close)`

Close connection to all servers and free all resources that the backend have reserved. All buffers that have been created by that backend should be killed. (Not the `nntp-server-buffer`, though.) This function is generally only called when Gnus is shutting down.

There should be no data returned.

`(nnchoke-server-opened &optional SERVER)`

If server is the current virtual server, and the connection to the physical server is alive, then this function should return a non-`nil` vlue. This function should under no circumstances attempt to reconnect to a server that is has lost connection to.

There should be no data returned.

`(nnchoke-status-message &optional SERVER)`

This function should return the last error message from server.

There should be no data returned.

`(nnchoke-request-article ARTICLE &optional GROUP SERVER TO-BUFFER)`

The result data from this function should be the article specified by article. This might either be a `Message-ID` or a number. It is optional whether to implement retrieval by `Message-ID`, but it would be nice if that were possible.

If to-buffer is non-`nil`, the result data should be returned in this buffer instead of the normal data buffer. This is to make it possible to avoid copying large amounts of data from one buffer to another, and Gnus mainly request articles to be inserted directly into its article buffer.

If it is at all possible, this function should return a cons cell where the car is the group name the article was fetched from, and the cdr is the article number. This will enable Gnus to find out what the real group and article numbers are when fetching articles by `Message-ID`. If this isn't possible, `t` should be returned on successful article retrievement.

`(nnchoke-open-group GROUP &optional SERVER)`

Make group the current group.

There should be no data returned by this function.

`(nnchoke-request-group GROUP &optional SERVER)`

Get data on group. This function also has the side effect of making group the current group.

Here's an example of some result data and a definition of the same:

```
211 56 1000 1059 ifi.discussion
```

The first number is the status, which should be `211`. Next is the total number of articles in the group, the lowest article number, the highest article number, and finally the group name. Note that the total number of articles may be less than one might think while just considering the highest and lowest article numbers, but some articles may have been canceled. Gnus just discards the total-number, so whether one should take the bother to generate it properly (if that is a problem) is left as an exercise to the reader.

```
group-status = [ error / info ] eol
error        = [ "4" / "5" ] 2<number> " " <Error message>
info         = "211 " 3* [ <number> " " ] <string>
```

`(nnchoke-close-group GROUP &optional SERVER)`

Close group and free any resources connected to it. This will be a no-op on most backends.

There should be no data returned.

`(nnchoke-request-list &optional SERVER)`

Return a list of all groups available on server. And that means *all*.

Here's an example from a server that only carries two groups:

```
ifi.test 0000002200 0000002000 y
ifi.discussion 3324 3300 n
```

On each line we have a group name, then the highest article number in that group, the lowest article number, and finally a flag.

```
active-file = *active-line
active-line = name " " <number> " " <number> " " flags eol
name        = <string>
flags       = "n" / "y" / "m" / "x" / "j" / "=" name
```

The flag says whether the group is read-only (`n'), is moderated (`m'), is dead (`x'), is aliased to some other group (`=other-group' or none of the above (`y').

`(nnchoke-request-post &optional SERVER)`

This function should post the current buffer. It might return whether the posting was successful or not, but that's not required. If, for instance, the posting is done asynchronously, it has generally not been completed by the time this function concludes. In that case, this function should set up some kind of sentinel to beep the user loud and clear if the posting could not be completed.

There should be no result data from this function.

Optional Backend Functions

`(nnchoke-retrieve-groups GROUPS &optional SERVER)`

groups is a list of groups, and this function should request data on all those groups. How it does it is of no concern to Gnus, but it should attempt to do this in a speedy fashion.

The return value of this function can be either `active` or `group`, which says what the format of the result data is. The former is in the same format as the data from `nnchoke-request-list`, while the latter is a buffer full of lines in the same format as `nnchoke-request-group` gives.

```
group-buffer = *active-line / *group-status
```

`(nnchoke-request-update-info GROUP INFO &optional SERVER)`

A Gnus group info (see section Group Info) is handed to the backend for alterations. This comes in handy if the backend really carries all the information (as is the case with virtual an imap groups). This function may alter the info in any manner it sees fit, and should return the (altered) group info. This function may alter the group info destructively, so no copying is needed before boogeying.

There should be no result data from this function.

`(nnchoke-request-type GROUP &optional ARTICLE)`

When the user issues commands for "sending news" (F in the summary buffer, for instance), Gnus has to know whether the article the user is following up is news or mail. This function should return `news` if article in group is news, `mail` if it is mail and `unknown` if the type can't be decided. (The article parameter is necessary in `nnvirtual` groups which might very well combine mail groups and news groups.)

There should be no result data from this function.

`(nnchoke-request-update-mark GROUP ARTICLE MARK)`

If the user tries to set a mark that the backend doesn't like, this function may change the mark. Gnus will use whatever this function returns as the mark for article instead of the original mark. If the backend doesn't care, it must return the original mark, and not `nil` or any other type of garbage.

The only use for this that I can see is what `nnvirtual` does with it--if a component group is auto-expirable, marking an article as read in the virtual group should result in the article being marked as expirable.

There should be no result data from this function.

`(nnchoke-request-scan &optional GROUP SERVER)`

This function may be called at any time (by Gnus or anything else) to request that the backend check for incoming articles, in one way or another. A mail backend will typically read the spool file or query the POP server when this function is invoked. The group doesn't have to be heeded--if the backend decides that it is too much work just scanning for a single group, it may do a total scan of all groups. It would be nice, however, to keep things local if that's practical.

There should be no result data from this function.

`(nnchoke-request-asynchronous GROUP &optional SERVER ARTICLES)`

This is a request to fetch articles asynchronously later. articles is an alist of (article-number line-number). One would generally expect that if one later fetches article number 4, for instance, some sort of asynchronous fetching of the articles after 4 (which might be 5, 6, 7 or 11, 3, 909 depending on the order in that alist) would be fetched asynchronously, but that is left up to the backend. Gnus doesn't care.

There should be no result data from this function.

`(nnchoke-request-group-description GROUP &optional SERVER)`

The result data from this function should be a description of group.

```
description-line = name <TAB> description eol
name             = <string>
description      = <text>
```

`(nnchoke-request-list-newsgroups &optional SERVER)`

The result data from this function should be the description of all groups available on the server.

```
description-buffer = *description-line
```

`(nnchoke-request-newgroups DATE &optional SERVER)`

The result data from this function should be all groups that were created after `date', which is in normal human-readable date format. The data should be in the active buffer format.

`(nnchoke-request-create-group GROUP &optional SERVER)`

This function should create an empty group with name group.

There should be no return data.

`(nnchoke-request-expire-articles ARTICLES &optional GROUP SERVER FORCE)`

This function should run the expiry process on all articles in the articles range (which is currently a simple list of article numbers.) It is left up to the backend to decide how old articles should be before they are removed by this function. If force is non-`nil`, all articles should be deleted, no matter how new they are.

This function should return a list of articles that it did not/was not able to delete.

There should be no result data returned.

`(nnchoke-request-move-article ARTICLE GROUP SERVER ACCEPT-FORM &optional LAST)`

This function should move article (which is a number) from group by calling accept-form.

This function should ready the article in question for moving by removing any header lines it has added to the article, and generally should "tidy up" the article. Then it should `eval` accept-form in the buffer where the "tidy" article is. This will do the actual copying. If this `eval` returns a non-`nil` value, the article should be removed.

If last is `nil`, that means that there is a high likelihood that there will be more requests issued shortly, so that allows some optimizations.

The function should return a cons where the car is the group name and the cdr is the article number that the article was entered as.

There should be no data returned.

`(nnchoke-request-accept-article GROUP &optional SERVER LAST)`

This function takes the current buffer and inserts it into group. If last in `nil`, that means that there will be more calls to this function in short order.

The function should return a cons where the car is the group name and the cdr is the article number that the article was entered as.

There should be no data returned.

`(nnchoke-request-replace-article ARTICLE GROUP BUFFER)`

This function should remove article (which is a number) from group and insert buffer there instead.

There should be no data returned.

`(nnchoke-request-delete-group GROUP FORCE &optional SERVER)`

This function should delete group. If force, it should really delete all the articles in the group, and then delete the group itself. (If there is such a thing as "the group itself".)

There should be no data returned.

`(nnchoke-request-rename-group GROUP NEW-NAME &optional SERVER)`

This function should rename group into new-name. All articles that are in group should move to new-name.

There should be no data returned.

Writing New Backends

The various backends share many similarities. `nnml` is just like `nnspool`, but it allows you to edit the articles on the server. `nnmh` is just like `nnml`, but it doesn't use an active file, and it doesn't maintain overview databases. `nndir` is just like `nnml`, but it has no concept of "groups", and it doesn't allow editing articles.

It would make sense if it were possible to "inherit" functions from backends when writing new backends. And, indeed, you can do that if you want to. (You don't have to if you don't want to, of course.)

All the backends declare their public variables and functions by using a package called `nnoo`.

To inherit functions from other backends (and allow other backends to inherit functions from the current backend), you should use the following macros: following.

`nnoo-declare`

This macro declares the first parameter to be a child of the subsequent parameters. For instance:

```
(nnoo-declare nndir
  nnml nnmh)
```

`nndir` has here declared that it intends to inherit functions from both `nnml` and `nnmh`.

`defvoo`

This macro is equivalent to `defvar`, but registers the variable as a public server variable. Most state-oriented variables sho declared with `defvoo` instead of `defvar`.

In addition to the normal `defvar` parameters, it takes a list of variables in the parent backends to map the variable to when executing a function in those backends.

```
(defvoo nndir-directory nil
  "Where nndir will look for groups."
  nnml-current-directory nnmh-current-directory)
```

This means that `nnml-current-directory` will be set to `nndir-directory` when an `nnml` function is called on behalf of `nndir`. (The same with `nnmh`.)

`nnoo-define-basics`

This macro defines some common functions that almost all backends should have.

```
(nnoo-define-basics nndir)
```

`deffoo`

This macro is just like `defun` and takes the same parameters. In addition to doing the normal `defun` things, it registers the function as being public so that other backends can inherit it.

`nnoo-map-functions`

This macro allows mapping of functions from the current backend to functions from the parent backends.

```
(nnoo-map-functions nndir
  (nnml-retrieve-headers 0 nndir-current-group 0 0)
  (nnmh-request-article 0 nndir-current-group 0 0))
```

This means that when `nndir-retrieve-headers` is called, the first, third, and fourth parameters will be passed on to `nnml-retrieve-headers`, while the second parameter is set to the value of `nndir-current-group`.

`nnoo-import`

This macro allows importing functions from backends. It should be the last thing in the source file, since it will only define functions that haven't already been defined.

```
(nnoo-import nndir
  (nnmh
   nnmh-request-list
   nnmh-request-newgroups)
  (nnml))
```

This means that calls to `nndir-request-list` should just be passed on to `nnmh-request-list`, while all public functions from `nnml` that haven't been defined in `nndir` yet should be defined now.

Below is a slightly shortened version of the `nndir` backend.

```
;;; nndir.el -- single directory newsgroup access for Gnus
;; Copyright (C) 1995,96 Free Software Foundation, Inc.

;;; Code:

(require 'nnheader)
(require 'nnmh)
(require 'nnml)
(require 'nnoo)
(eval-when-compile (require 'cl))

(nnoo-declare nndir
  nnml nnmh)

(defvoo nndir-directory nil
```


```
  "Where nndir will look for groups."
  nnml-current-directory nnmh-current-directory)

(defvoo nndir-nov-is-evil nil
  "*Non-nil means that nndir will never retrieve NOV headers."
  nnml-nov-is-evil)

(defvoo nndir-current-group "" nil nnml-current-group nnmh-current-group)
(defvoo nndir-top-directory nil nil nnml-directory nnmh-directory)
(defvoo nndir-get-new-mail nil nil nnml-get-new-mail nnmh-get-new-mail)

(defvoo nndir-status-string "" nil nnmh-status-string)
(defconst nndir-version "nndir 1.0")

;;; Interface functions.

(nnoo-define-basics nndir)

(deffoo nndir-open-server (server &optional defs)
  (setq nndir-directory
        (or (cadr (assq 'nndir-directory defs))
            server))
  (unless (assq 'nndir-directory defs)
    (push `(nndir-directory ,server) defs))
  (push `(nndir-current-group
          ,(file-name-nondirectory (directory-file-name nndir-directory)))
        defs)
  (push `(nndir-top-directory
          ,(file-name-directory (directory-file-name nndir-directory)))
        defs)
  (nnoo-change-server 'nndir server defs))

(nnoo-map-functions nndir
  (nnml-retrieve-headers 0 nndir-current-group 0 0)
  (nnmh-request-article 0 nndir-current-group 0 0)
  (nnmh-request-group nndir-current-group 0 0)
  (nnmh-close-group nndir-current-group 0))

(nnoo-import nndir
  (nnmh
   nnmh-status-message
   nnmh-request-list
   nnmh-request-newgroups))

(provide 'nndir)
```

Score File Syntax

Score files are meant to be easily parsable, but yet extremely mallable. It was decided that something that had the same read syntax as an Emacs Lisp list would fit that spec.

Here's a typical score file:

```
(("summary"
  ("win95" -10000 nil s)
  ("Gnus"))
 ("from"
  ("Lars" -1000))
 (mark -100))
```

BNF definition of a score file:

```
score-file       = "" / "(" *element ")"
element          = rule / atom
rule             = string-rule / number-rule / date-rule
string-rule      = "(" quote string-header quote space *string-match ")"
number-rule      = "(" quote number-header quote space *number-match ")"
date-rule        = "(" quote date-header quote space *date-match ")"
quote            = <ascii 34>
string-header    = "subject" / "from" / "references" / "message-id" /
                   "xref" / "body" / "head" / "all" / "followup"
number-header    = "lines" / "chars"
date-header      = "date"
string-match     = "(" quote <string> quote [ "" / [ space score [ "" /
                   space date [ "" / [ space string-match-t ] ] ] ] ] ")"
score            = "nil" / <integer>
date             = "nil" / <natural number>
string-match-t   = "nil" / "s" / "substring" / "S" / "Substring" /
                   "r" / "regex" / "R" / "Regex" /
                   "e" / "exact" / "E" / "Exact" /
                   "f" / "fuzzy" / "F" / "Fuzzy"
number-match     = "(" <integer> [ "" / [ space score [ "" /
                   space date [ "" / [ space number-match-t ] ] ] ] ] ")"
number-match-t   = "nil" / "=" / "<" / ">" / ">=" / "<="
date-match       = "(" quote <string> quote [ "" / [ space score [ "" /
                   space date [ "" / [ space date-match-t ] ] ] ] ")"
date-match-t     = "nil" / "at" / "before" / "after"
atom             = "(" [ required-atom / optional-atom ] ")"
required-atom    = mark / expunge / mark-and-expunge / files /
                   exclude-files / read-only / touched
optional-atom    = adapt / local / eval
mark             = "mark" space nil-or-number
nil-or-number    = "nil" / <integer>
expunge          = "expunge" space nil-or-number
mark-and-expunge = "mark-and-expunge" space nil-or-number
files            = "files" *[ space <string> ]
exclude-files    = "exclude-files" *[ space <string> ]
read-only        = "read-only" [ space "nil" / space "t" ]
adapt            = "adapt" [ space "nil" / space "t" / space adapt-rule ]
adapt-rule       = "(" *[ <string> *[ "(" <string> <integer> ")" ] ")"
local            = "local" *[ space "(" <string> space <form> ")" ]
eval             = "eval" space <form>
space            = *[ " " / <TAB> / <NEWLINE> ]
```

Any unrecognized elements in a score file should be ignored, but not discarded.

As you can see, white space is needed, but the type and amount of white space is irrelevant. This means that formatting of the score file is left up to the programmer--if it's simpler to just spew it all out on one looong line, then that's ok.

The meaning of the various atoms are explained elsewhere in this manual.

Headers

Gnus uses internally a format for storing article headers that corresponds to the NOV format in a mysterious fashion. One could almost suspect that the author looked at the NOV specification and just shamelessly *stole* the entire thing, and one would be right.

Header is a severely overloaded term. "Header" is used in RFC1036 to talk about lines in the head of an article (eg., `From`). It is used by many people as a synonym for "head"---"the header and the body". (That should be avoided, in my opinion.) And Gnus uses a format internally that it calls "header", which is what I'm talking about here. This is a 9-element vector, basically, with each header (ouch) having one slot.

These slots are, in order: `number`, `subject`, `from`, `date`, `id`, `references`, `chars`, `lines`, `xref`. There are macros for accessing and setting these slots -- they all have predictable names beginning with `mail-header-` and `mail-header-set-`, respectively.

The `xref` slot is really a `misc` slot. Any extra info will be put in there.

Ranges

GNUS introduced a concept that I found so useful that I've started using it a lot and have elaborated on it greatly.

The question is simple: If you have a large amount of objects that are identified by numbers (say, articles, to take a *wild* example) that you want to callify as being "included", a normal sequence isn't very useful. (A 200,000 length sequence is a bit long-winded.)

The solution is as simple as the question: You just collapse the sequence.

```
(1 2 3 4 5 6 10 11 12)
```

is transformed into

```
((1 . 6) (10 . 12))
```

To avoid having those nasty `(13 . 13)' elements to denote a lonesome object, a `13' is a valid element:

```
((1 . 6) 7 (10 . 12))
```

This means that comparing two ranges to find out whether they are equal is slightly tricky:

```
((1 . 5) 7 8 (10 . 12))
```

and

```
((1 . 5) (7 . 8) (10 . 12))
```

are equal. In fact, any non-descending list is a range:

```
(1 2 3 4 5)
```

is a perfectly valid range, although a pretty long-winded one. This is also legal:

```
(1 . 5)
```

and is equal to the previous range.

Here's a BNF definition of ranges. Of course, one must remember the semantic requirement that the numbers are non-descending. (Any number of repetition of the same number is allowed, but apt to disappear in range handling.)

```
range           = simple-range / normal-range
simple-range    = "(" number " . " number ")"
normal-range    = "(" start-contents ")"
contents        = "" / simple-range *[ " " contents ] /
                  number *[ " " contents ]
```

Gnus currently uses ranges to keep track of read articles and article marks. I plan on implementing a number of range operators in C if The Powers That Be are willing to let me. (I haven't asked yet, because I need to do some more thinking on what operators I need to make life totally range-based without ever having to convert back to normal sequences.)

Group Info

Gnus stores all permanent info on groups in a group info list. This list is from three to six elements (or more) long and exhaustively describes the group.

Here are two example group infos; one is a very simple group while the second is a more complex one:


```
("no.group" 5 (1 . 54324))

("nnml:my.mail" 3 ((1 . 5) 9 (20 . 55))
                ((tick (15 . 19)) (replied 3 6 (19 . 3)))
                (nnml "")
                (auto-expire (to-address "ding@ifi.uio.no")))
```

The first element is the group name as Gnus knows the group; the second is the group level; the third is the read articles in range format; the fourth is a list of article marks lists; the fifth is the select method; and the sixth contains the group parameters.

Here's a BNF definition of the group info format:

```
info          = "(" group space level space read
                [ "" / [ space marks-list [ "" / [ space method [ "" /
                space parameters ] ] ] ] ] ")"
group         = quote <string> quote
level         = <integer in the range of 1 to inf>
read          = range
marks-lists   = nil / "(" *marks ")"
marks         = "(" <string> range ")"
method        = "(" <string> *elisp-forms ")"
parameters    = "(" *elisp-forms ")"
```

Actually that `marks' rule is a fib. A `marks' is a `<string>' consed on to a `range', but that's a bitch to say in pseudo-BNF.

Emacs/XEmacs Code

While Gnus runs under Emacs, XEmacs and Mule, I decided that one of the platforms must be the primary one. I chose Emacs. Not because I don't like XEmacs or Mule, but because it comes first alphabetically.

This means that Gnus will byte-compile under Emacs with nary a warning, while XEmacs will pump out gigabytes of warnings while byte-compiling. As I use byte-compilation warnings to help me root out trivial errors in Gnus, that's very useful.

I've also consistently used Emacs function interfaces, but have used Gnusey aliases for the functions. To take an example: Emacs defines a `run-at-time` function while XEmacs defines a `start-itimer` function. I then define a function called `gnus-run-at-time` that takes the same parameters as the Emacs `run-at-time`. When running Gnus under Emacs, the former function is just an alias for the latter. However, when running under XEmacs, the former is an alias for the following function:

```
(defun gnus-xmas-run-at-time (time repeat function &rest args)
  (start-itimer
   "gnus-run-at-time"
   `(lambda ()
      (,function ,@args))
   time repeat))
```

This sort of thing has been done for bunches of functions. Gnus does not redefine any native Emacs functions while running under XEmacs -- it does this `defalias` thing with Gnus equivalents instead. Cleaner all over.

Of course, I could have chosen XEmacs as my native platform and done mapping functions the other way around. But I didn't. The performance hit these indirections impose on Gnus under XEmacs should be slight.

Various File Formats

Active File Format

The active file lists all groups that are available on the server in question. It also lists the highest and lowest current article numbers in each group.

Here's an excerpt from a typical active file:


```
soc.motss 296030 293865 y
alt.binaries.pictures.fractals 3922 3913 n
comp.sources.unix 1605 1593 m
comp.binaries.ibm.pc 5097 5089 y
no.general 1000 900 y
```

Here's a pseudo-BNF definition of this file:

```
active      = *group-line
group-line  = group space high-number space low-number space flag <NEWLINE>
group       = <non-white-space string>
space       = " "
high-number = <non-negative integer>
low-number  = <positive integer>
flag        = "y" / "n" / "m" / "j" / "x" / "=" group
```

Newsgroups File Format

The newsgroups file lists groups along with their descriptions. Not all groups on the server have to be listed, and not all groups in the file have to exist on the server. The file is meant purely as information to the user.

The format is quite simple; a group name, a tab, and the description. Here's the definition:

```
newsgroups    = *line
line          = group tab description <NEWLINE>
group         = <non-white-space string>
tab           = <TAB>
description   = <string>
```

Emacs for Heathens

Believe it or not, but some people who use Gnus haven't really used Emacs much before they embarked on their journey on the Gnus Love Boat. If you are one of those unfortunates whom "M-C-a", "kill the region", and "set `gnus-flargblossen` to an alist where the key is a regexp that is used for matching on the group name" are magical phrases with little or no meaning, then this appendix is for you. If you are already familiar with Emacs, just ignore this and go fondle your cat instead.

Keystrokes

- Q: What is an experienced Emacs user?
- A: A person who wishes that the terminal had pedals.

Yes, when you use Emacs, you are apt to use the control key, the shift key and the meta key a lot. This is very annoying to some people (notably `vi`le users), and the rest of us just love the hell out of it. Just give up and submit. Emacs really does stand for "Escape-Meta-Alt-Control-Shift", and not "Editing Macros", as you may have heard from other disreputable sources (like the Emacs author).

The shift key is normally located near your pinky fingers, and are normally used to get capital letters and stuff. You probably use it all the time. The control key is normally marked "CTRL" or something like that. The meta key is, funnily enough, never marked as such on any keyboards. The one I'm currently at has a key that's marked "Alt", which is the meta key on this keyboard. It's usually located somewhere to the left hand side of the keyboard, usually on the bottom row.

Now, us Emacs people doesn't say "press the meta-control-m key", because that's just too inconvenient. We say "press the M-C-m key". M- is the prefix that means "meta" and "C-" is the prefix that means "control". So "press C-k" means "press down the control key, and hold it down while you press k". "Press M-C-k" means "press down and hold down the meta key and the control key and then press k". Simple, ay?

This is somewhat complicated by the fact that not all keyboards have a meta key. In that case you can use the "escape" key. Then M-k means "press escape, release escape, press k". That's much more work than if you have a meta key, so if that's the case, I respectfully suggest you get a real keyboard with a meta key. You can't live without it.

Emacs Lisp

Emacs is the King of Editors because it's really a Lisp interpreter. Each and every key you tap runs some Emacs Lisp code snippet, and since Emacs Lisp is an interpreted language, that means that you can configure any key to run any arbitrary code. You just, like, do it.

Gnus is written in Emacs Lisp, and is run as a bunch of interpreted functions. (These are byte-compiled for speed, but it's still interpreted.) If you decide that you don't like the way Gnus does certain things, it's trivial to have it do something a different way. (Well, at least if you know how to write Lisp code.) However, that's beyond the scope of this manual, so we are simply going to talk about some common constructs that you normally use in your `.emacs' file to customize Gnus.

If you want to set the variable `gnus-florgbnize` to four (4), you write the following:

```
(setq gnus-florgbnize 4)
```

This function (really "special form") `setq` is the one that can set a variable to some value. This is really all you need to know. Now you can go and fill your `.emacs` file with lots of these to change how Gnus works.

If you have put that thing in your `.emacs` file, it will be read and `eval`ed (which is lisp-ese for "run") the next time you start Emacs. If you want to change the variable right away, simply say C-x C-e after the closing parenthesis. That will `eval` the previous "form", which here is a simple `setq` statement.

Go ahead--just try it, if you're located at your Emacs. After you C-x C-e, you will see `4' appear in the echo area, which is the return value of the form you `eval`ed.

Some pitfalls:

If the manual says "set `gnus-read-active-file` to `some`", that means:

```
(setq gnus-read-active-file 'some)
```

On the other hand, if the manual says "set `gnus-nntp-server` to `nntp.ifi.uio.no'", that means:

```
(setq gnus-nntp-server "nntp.ifi.uio.no")
```

So be careful not to mix up strings (the latter) with symbols (the former). The manual is unambiguous, but it can be confusing.

\input texinfo

Frequently Asked Questions

This is the Gnus Frequently Asked Questions list. If you have a Web browser, the official hypertext version is at `http://www.miranova.com/~steve/gnus-faq.html>', and has probably been updated since you got this manual.

Installation

- Q1.1 What is the latest version of Gnus?

  The latest (and greatest) version is 5.0.10. You might also run across something called *September Gnus*. September Gnus is the alpha version of the next major release of Gnus. It is currently not stable enough to run unless you are prepared to debug lisp.

- Q1.2 Where do I get Gnus?

  Any of the following locations:
  - `ftp://ftp.ifi.uio.no/pub/emacs/gnus/gnus.tar.gz'
  - `ftp://ftp.pilgrim.umass.edu/pub/misc/ding/'
  - `gopher://gopher.pilgrim.umass.edu/11/pub/misc/ding/'
  - `ftp://aphrodite.nectar.cs.cmu.edu/pub/ding-gnus/'
  - `ftp://ftp.solace.mh.se:/pub/gnu/elisp/'

- Q1.3 Which version of Emacs do I need?

  At least GNU Emacs 19.28, or XEmacs 19.12 is recommended. GNU Emacs 19.25 has been reported to work under certain

circumstances, but it doesn't *officially* work on it. 19.27 has also been reported to work. Gnus has been reported to work under OS/2 as well as Unix.

- Q1.4 Where is timezone.el?

  Upgrade to XEmacs 19.13. In earlier versions of XEmacs this file was placed with Gnus 4.1.3, but that has been corrected.

- Q1.5 When I run Gnus on XEmacs 19.13 I get weird error messages.

  You're running an old version of Gnus. Upgrade to at least version 5.0.4.

- Q1.6 How do I unsubscribe from the Mailing List?

  Send an e-mail message to `` `ding-request@ifi.uio.no' `` with the magic word *unsubscribe* somewhere in it, and you will be removed.

  If you are reading the digest version of the list, send an e-mail message to `` `ding-rn-digests-d-request@moe.shore.net' `` with *unsubscribe* as the subject and you will be removed.

- Q1.7 How do I run Gnus on both Emacs and XEmacs?

  The basic answer is to byte-compile under XEmacs, and then you can run under either Emacsen. There is, however, a potential version problem with easymenu.el with Gnu Emacs prior to 19.29.

  Per Abrahamsen <abraham@dina.kvl.dk> writes :
  The internal easymenu.el interface changed between 19.28 and 19.29 in order to make it possible to create byte compiled files that can be shared between Gnu Emacs and XEmacs. The change is upward compatible, but not downward compatible. This gives the following compatibility table:

```
Compiled with:  | Can be used with:
----------------+-----------------------------------
19.28           | 19.28         19.29
19.29           |               19.29          XEmacs
XEmacs          |               19.29          XEmacs
```

  If you have Gnu Emacs 19.28 or earlier, or XEmacs 19.12 or earlier, get a recent version of auc-menu.el from `` `ftp://ftp.iesd.auc.dk/pub/emacs-lisp/auc-menu.el' ``, and install it under the name easymenu.el somewhere early in your load path.

- Q1.8 What resources are available?

  There is the newsgroup Gnu.emacs.gnus. Discussion of Gnus 5.x is now taking place there. There is also a mailing list, send mail to `` `ding-request@ifi.uio.no' `` with the magic word *subscribe* somewhere in it.

  NOTE: the traffic on this list is heavy so you may not want to be on it (unless you use Gnus as your mailer reader, that is). The mailing list is mainly for developers and testers.

  Gnus has a home World Wide Web page at `` `http://www.ifi.uio.no/~larsi/ding.html' ``. Gnus has a write up in the X Windows Applications FAQ at `` `http://www.ee.ryerson.ca:8080/~elf/xapps/Q-III.html' ``. The Gnus manual is also available on the World Wide Web. The canonical source is in Norway at `` `http://www.ifi.uio.no/~larsi/ding-manual/gnus_toc.html' ``.

  There are three mirrors in the United States:

  1. `` `http://www.miranova.com/gnus-man/' ``
  2. `` `http://www.pilgrim.umass.edu/pub/misc/ding/manual/gnus_toc.html' ``
  3. `` `http://www.rtd.com/~woo/gnus/' ``

  PostScript copies of the Gnus Reference card are available from `` `ftp://ftp.cs.ualberta.ca/pub/oolog/gnus/' ``. They are mirrored at `` `ftp://ftp.pilgrim.umass.edu/pub/misc/ding/refcard/' `` in the United States. And `` `ftp://marvin.fkphy.uni-duesseldorf.de/pub/gnus/' `` in Germany. An online version of the Gnus FAQ is available at `` `http://www.miranova.com/~steve/gnus-faq.html' ``. Off-line formats are also available:
  ASCII: `` `ftp://ftp.miranova.com/pub/gnus/gnus-faq' ``
  PostScript: `` `ftp://ftp.miranova.com/pub/gnus/gnus-faq.ps' ``.

- Q1.9 Gnus hangs on connecting to NNTP server

  I am running XEmacs on SunOS and Gnus prints a message about Connecting to NNTP server and then just hangs.

  Ben Wing <wing@netcom.com> writes :
  I wonder if you're hitting the infamous *libresolv* problem. The basic problem is that under SunOS you can compile either with DNS or NIS name lookup libraries but not both. Try substituting the IP address and see if that works; if so, you need to download the sources and recompile.

- Q1.10 Mailcrypt 3.4 doesn't work

  This problem is verified to still exist in Gnus 5.0.9 and MailCrypt 3.4. The answer comes from Peter Arius <arius@immd2.informatik.uni-erlangen.de>.

  I found out that mailcrypt uses `gnus-eval-in-buffer-window`, which is a macro. It seems as if you have compiled mailcrypt with plain old GNUS in load path, and the XEmacs byte compiler has inserted that macro definition into `` `mc-toplev.elc' ``. The solution is to recompile `` `mc-toplev.el' `` with Gnus 5 in load-path, and it works fine.

  Steve Baur <steve@miranova.com> adds :
  The problem also manifests itself if neither GNUS 4 nor Gnus 5 is in the load-path.

- Q1.11 What other packages work with Gnus?
  - Mailcrypt.

    Mailcrypt is an Emacs interface to PGP. It works, it installs without hassle, and integrates very easily. Mailcrypt can be obtained from
    `` `ftp://cag.lcs.mit.edu/pub/patl/mailcrypt-3.4.tar.gz' ``.
  - Tiny Mime.

    Tiny Mime is an Emacs MUA interface to MIME. Installation is a two-step process unlike most other packages, so you should be prepared to move the byte-compiled code somewhere. There are currently two versions of this package available. It can be obtained from
    `` `ftp://ftp.jaist.ac.jp/pub/GNU/elisp/' ``. Be sure to apply the supplied patch. It works with Gnus through version 5.0.9. In order for all dependencies to work correctly the load sequence is as follows:

    ```
    (load "tm-setup")
    (load "gnus")
    (load "mime-compose")
    ```

    NOTE: Loading the package disables citation highlighting by default. To get the old behavior back, use the M-t command.

Customization

- Q2.1 Custom Edit does not work under XEmacs

  The custom package has not been ported to XEmacs.

- Q2.2 How do I quote messages?

  I see lots of messages with quoted material in them. I am wondering how to have Gnus do it for me.

  This is Gnus, so there are a number of ways of doing this. You can use the built-in commands to do this. There are the F and R keys from the summary buffer which automatically include the article being responded to. These commands are also selectable as *Followup and Yank* and *Reply and Yank* in the Post menu.

  C-c C-y grabs the previous message and prefixes each line with `ail-indentation-spaces` spaces or `mail-yank-prefix` if that is non-nil, unless you have set your own `mail-citation-hook`, which will be called to to do the job.

  You might also consider the Supercite package, which allows for pretty arbitrarily complex quoting styles. Some people love it, some people hate it.

- Q2.3 How can I keep my nnvirtual:* groups sorted?

  How can I most efficiently arrange matters so as to keep my nnvirtual:* (etc) groups at the top of my group selection buffer, whilst keeping everything sorted in alphabetical order. If you don't subscribe often to new groups then the easiest way is to first sort the groups and then manually kill and yank the virtuals wherever you want them.

- Q2.4 Any good suggestions on stuff for an all.SCORE file?

  Here is a collection of suggestions from the Gnus mailing list.

  1. From "Dave Disser" <disser@sdd.hp.com>
     I like blasting anything without lowercase letters. Weeds out most of the make $$ fast, as well as the lame titles like "IBM" and "HP-UX" with no further description.

     ```
     (("Subject"
      ("^\\(Re: \\)?[^a-z]*$" -200 nil R)))
     ```

  2. From "Peter Arius" <arius@immd2.informatik.uni-erlangen.de>
     The most vital entries in my (still young) all.SCORE:

     ```
     (("xref"
       ("alt.fan.oj-simpson" -1000 nil s))
      ("subject"
       ("\\<\\(make\\|fast\\|big\\)\\s-*\\(money\\|cash\\|bucks?\\)\\>" -1000 nil r)
       ("$$$$" -1000 nil s)))
     ```

  3. From "Per Abrahamsen" <abraham@dina.kvl.dk>

     ```
     (("subject"
       ;; CAPS OF THE WORLD, UNITE
       ("^..[^a-z]+$" -1 nil R)
       ;; $$$ Make Money $$$ (Try work)
       ("$" -1 nil s)
       ;; I'm important! And I have exclamation marks to prove it!
       ("!" -1 nil s)))
     ```

  4. From "heddy boubaker" <boubaker@cenatls.cena.dgac.fr>
     I would like to contribute with mine.

     ```
     (
      (read-only t)
      ("subject"
       ;; ALL CAPS SUBJECTS
       ("^\\([Rr][Ee]: +\\)?[^a-z]+$" -1 nil R)
       ;; $$$ Make Money $$$
       ("$$" -10 nil s)
       ;; Empty subjects are worthless!
       ("^ *\\([(<]none[>)]\\|(no subject\\( given\\)?)\\)? *$" -10 nil r)
       ;; Sometimes interesting announces occur!
       ("ANN?OU?NC\\(E\\|ING\\)" +10 nil r)
       ;; Some people think they're on mailing lists
       ("\\(un\\)?sub?scribe" -100 nil r)
       ;; Stop Micro$oft NOW!!
       ("\\(m\\(icro\\)?[s$]\\(oft\\|lot\\)?-?\\)?wind?\\(ows\\|aube\\|oze\\)?[-
     ]*\\('?95\\|NT\\|3[.]1\\|32\\)" -1001 nil r)
       ;; I've nothing to buy
       ("\\(for\\|4\\)[- ]*sale" -100 nil r)
       ;; SELF-DISCIPLINED people
       ("\\[[^a-z0-9 \t\n][^a-z0-9 \t\n]\\]" +100 nil r)
       )
      ("from"
       ;; To keep track of posters from my site
       (".dgac.fr" +1000 nil s))
      ("followup"
       ;; Keep track of answers to my posts
       ("boubaker" +1000 nil s))
      ("lines"
     ```


```
 ;; Some people have really nothing to say!!
 (1 -10 nil <=))
(mark -100)
(expunge -1000)
)
```

5. From "Christopher Jones" <cjones@au.oracle.com>
   The sample `all.SCORE' files from Per and boubaker could be augmented with:

```
(("subject"
  ;; No junk mail please!
  ("please ignore" -500 nil s)
  ("test" -500 nil e))
)
```

6. From "Brian Edmonds" <edmonds@cs.ubc.ca>
   Augment any of the above with a fast method of scoring down excessively cross posted articles.

```
 ("xref"
  ;; the more cross posting, the exponentially worse the article
  ("^xref: \\S-+ \\S-+ \\S-+ \\S-+" -1 nil r)
  ("^xref: \\S-+ \\S-+ \\S-+ \\S-+ \\S-+" -2 nil r)
  ("^xref: \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+" -4 nil r)
  ("^xref: \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+" -8 nil r)
  ("^xref: \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+" -16 nil r)
  ("^xref: \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+" -32 nil r)
  ("^xref: \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+" -64 nil r)
  ("^xref: \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+" -128
nil r)
  ("^xref: \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+"
-256 nil r)
  ("^xref: \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+ \\S-+
\\S-+" -512 nil r))
```

- Q2.5 What do I use to yank-through when replying?

  You should probably reply and followup with R and F, instead of r and f, which solves your problem. But you could try something like:

```
(defconst mail-yank-ignored-headers
  "^.*:"
  "Delete these headers from old message when it's inserted in a reply.")
```

- Q2.6 I don't like the default WWW browser

  Now when choosing an URL Gnus starts up a W3 buffer, I would like it to always use Netscape (I don't browse in text-mode ;-).

  1. Activate `Customize...' from the `Help' menu.
  2. Scroll down to the `WWW Browser' field.
  3. Click `mouse-2' on `WWW Browser'.
  4. Select `Netscape' from the pop up menu.
  5. Press `C-c C-c'

  If you are using XEmacs then to specify Netscape do

```
(setq gnus-button-url 'gnus-netscape-open-url)
```

- Q2.7 What, if any, relation is between "ask-server" and "(setq gnus-read-active-file 'some)"?

  In order for Gnus to show you the complete list of newsgroups, it will either have to either store the list locally, or ask the server to transmit the list. You enable the first with

```
(setq gnus-save-killed-list t)
```

and the second with

```
(setq gnus-read-active-file t)
```

If both are disabled, Gnus will not know what newsgroups exists. There is no option to get the list by casting a spell.

- Q2.8 Moving between groups is slow.

  Per Abrahamsen <abraham@dina.kvl.dk> writes:

  Do you call `define-key` or something like that in one of the summary mode hooks? This would force Emacs to recalculate the keyboard shortcuts. Removing the call should speed up M-x gnus-summary-mode RET by a couple of orders of magnitude. You can use

  ```
  (define-key gnus-summary-mode-map KEY COMMAND)
  ```

  in your `` `.gnus' `` instead.

Reading News

- Q3.1 How do I convert my kill files to score files?

  A kill-to-score translator was written by Ethan Bradford <ethanb@ptolemy.astro.washington.edu>. It is available from `` `http://baugi.ifi.uio.no/~larsi/ding-various/gnus-kill-to-score.el' ``.

- Q3.2 My news server has a lot of groups, and killing groups is painfully slow.

  Don't do that then. The best way to get rid of groups that should be dead is to edit your newsrc directly. This problem will be addressed in the near future.

- Q3.3 How do I use an NNTP server with authentication?

  Put the following into your .gnus:

  ```
  (add-hook 'nntp-server-opened-hook 'nntp-send-authinfo)
  ```

- Q3.4 Not reading the first article.

  How do I avoid reading the first article when a group is selected?

  1. Use RET to select the group instead of SPC.
  2. `(setq gnus-auto-select first nil)`
  3. Luis Fernandes <elf@mailhost.ee.ryerson.ca>writes:
     This is what I use...customize as necessary...

     ```
     ;;; Don't auto-select first article if reading sources, or archives or
     ;;; jobs postings, etc. and just display the summary buffer
     (add-hook 'gnus-select-group-hook
               (function
                (lambda ()
                  (cond ((string-match "sources" gnus-newsgroup-name)
                         (setq gnus-auto-select-first nil))
                               ((string-match "jobs" gnus-newsgroup-name)
                                    (setq gnus-auto-select-first nil))
                        ((string-match "comp\\.archives" gnus-newsgroup-name)
                         (setq gnus-auto-select-first nil))
                        ((string-match "reviews" gnus-newsgroup-name)
                         (setq gnus-auto-select-first nil))
                        ((string-match "announce" gnus-newsgroup-name)
                         (setq gnus-auto-select-first nil))
                        ((string-match "binaries" gnus-newsgroup-name)
                         (setq gnus-auto-select-first nil))
                        (t
                         (setq gnus-auto-select-first t))))))
     ```

4. Per Abrahamsen <abraham@dina.kvl.dk> writes:
   Another possibility is to create an `all.binaries.all.SCORE' file like this:

   ```
   ((local
     (gnus-auto-select-first nil)))
   ```

   and insert

   ```
   (setq gnus-auto-select-first t)
   ```

   in your `.gnus'.

- Q3.5 Why aren't BBDB known posters marked in the summary buffer?

  Brian Edmonds <edmonds@cs.ubc.ca> writes:
  Due to changes in Gnus 5.0, `bbdb-gnus.el' no longer marks known posters in the summary buffer. An updated version, `gnus-bbdb.el' is available at the locations listed below. This package also supports autofiling of incoming mail to folders specified in the BBDB. Extensive instructions are included as comments in the file.

  Send mail to `majordomo@edmonds.home.cs.ubc.ca' with the following line in the body of the message: *get misc gnus-bbdb.el*.

  Or get it from the World Wide Web:
  `http://www.cs.ubc.ca/spider/edmonds/gnus-bbdb.el'.

Reading Mail

- Q4.1 What does the message "Buffer has changed on disk" mean in a mail group?

  Your filter program should not deliver mail directly to your folders, instead it should put the mail into spool files. Gnus will then move the mail safely from the spool files into the folders. This will eliminate the problem. Look it up in the manual, in the section entitled "Mail & Procmail".

- Q4.2 How do you make articles un-expirable?

  I am using nnml to read news and have used `gnus-auto-expirable-newsgroups` to automagically expire articles in some groups (Gnus being one of them). Sometimes there are interesting articles in these groups that I want to keep. Is there any way of explicitly marking an article as un-expirable - that is mark it as read but not expirable?

  Use u, !, d or M-u in the summary buffer. You just remove the E mark by setting some other mark. It's not necessary to tick the articles.

- Q4.3 How do I delete bogus nnml: groups?

  My problem is that I have various mail (nnml) groups generated while experimenting with Gnus. How do I remove them now? Setting the level to 9 does not help. Also `gnus-group-check-bogus-groups` does not recognize them.

  Removing mail groups is tricky at the moment. (It's on the to-do list, though.) You basically have to kill the groups in Gnus, shut down Gnus, edit the active file to exclude these groups, and probably remove the nnml directories that contained these groups as well. Then start Gnus back up again.

- Q4.4 What happened to my new mail groups?

  I got new mail, but I have never seen the groups they should have been placed in.

  They are probably there, but as zombies. Press A z to list zombie groups, and then subscribe to the groups you want with u. This is all documented quite nicely in the user's manual.

- Q4.5 Not scoring mail groups

  How do you *totally* turn off scoring in mail groups?

  Use an nnbabyl:all.SCORE (or nnmh, or nnml, or whatever) file containing:

  ```
  ((adapt ignore)
   (local (gnus-use-scoring nil))
   (exclude-files "all.SCORE"))
  ```

Go to the previous, next section.

TOC

Go to the previous, next section.

Index

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

r

s

t

u

V

W

X

Z

Go to the previous, next section.

Go to the previous section.

Key Index

!

#

&

*

,

.

/

<

=

>

?

^

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

V

W

x

y

Z

Go to the previous section.

mh-e

Go to the next section.

mh-e

The Emacs Interface to MH

by Bill Wohler

Edition 1.2 for mh-e Version 5.0.2

August 1995

Preface

These chapters introduce another interface to MH that is accessible through the GNU Emacs editor, namely, *mh-e*. mh-e is easy to use. I don't assume that you know GNU Emacs or even MH at this point, since I didn't know either of them when I discovered mh-e. However, mh-e was the tip of the iceberg, and I discovered more and more niceties about GNU Emacs and MH. Now I'm fully hooked on both of them.

The mh-e package is distributed with GNU Emacs, (1) so you shouldn't have to do anything special to use it. But it's important to note a brief history of mh-e. Version 3 was prevalent through the Emacs 18 and early Emacs 19 years. Then Version 4 came out (Emacs 19.23), which introduced several new and changed commands. Finally, Version 5.0 was released, which fixed some bugs and incompatibilities. This is the version covered by this manual. section Getting Started will help you decide which version you have.

If you don't already use GNU Emacs but want to learn more, you can read an online tutorial by starting GNU Emacs and typing C-h t (`help-with-tutorial`). (This notation is described in section GNU Emacs Terms and Conventions.) If you want to take the plunge, consult the GNU Emacs Manual, from the Free Software Foundation.

If more information is needed, you can go to the Unix manual pages of the individual MH commands. When the name is not obvious, I'll guide you to a relevant MH manual page that describes the action

more fully.

I hope you enjoy these chapters! If you have any comments, or suggestions for this document, please let me know.

Bill Wohler <*wohler@newt.com*>
8 February 1995

Go to the next section.

Go to the previous, next section.

Tour Through mh-e

This chapter introduces some of the terms you'll need to know and then takes you on a tour of mh-e. (2) When you're done, you'll be able to send, read, and file mail, which is all that a lot of people ever do. But if you're the curious type, you'll read section Using mh-e to be able to use all the features of mh-e. If you're the adventurous type, you'll read section Customizing mh-e to make mh-e do what you want. I suggest you read this chapter first to get the big picture, and then you can read the other two as you wish.

GNU Emacs Terms and Conventions

If you're an experienced Emacs user, you can skip the following conventions and definition of terms and go directly to section Getting Started below. The conventions are as follows:

C-x
: Hold down the CTRL (Control) key and press the x key.

M-x
: Hold down the META or ALT key and press the x key.

  Since some keyboards don't have a META key, you can generate M-x, for example, by pressing ESC (Escape), *releasing it*, (3) and then pressing the x key.

RET
: Press the RETURN or ENTER key. This is normally used to complete a command.

SPC
: Press the space bar.

TAB
: Press the TAB key.

DEL
: Press the DELETE key. This may also be a Backspace key, depending on your keyboard or Emacs configuration.

A prefix argument allows you to pass an argument to any Emacs function. To pass an argument, type C-u before the Emacs command or keystroke. Numeric arguments can be passed as well. For example, to insert five f's, use C-u 5 f. There is a default of four when using C-u, and you can use multiple prefix arguments to provide arguments of powers of four. To continue our example, you could insert four f's with C-u f, 16 f's with C-u C-u f, 64 f's with C-u C-u C-u f, and so on. Numeric and valueless negative arguments can also be inserted with the META key. Examples include M-5 to specify an argument of 5, or M-- which specifies a negative argument with no particular value.

NOTE

The prefix C-u or M- is not necessary in mh-e's MH-Folder modes (see section Receiving Mail). In these modes, simply enter the numerical argument before entering the command.

There are several other terms that are used in Emacs that you should know. The point is where the cursor currently is. You can save your current place in the file by setting a mark. This operation is useful in several ways. The mark can be later used when defining a region, which is the text between the point and mark. Many commands operate on regions, such as those for deleting text or filling paragraphs. A mark can be set with C-@ (or C-SPC).

The minibuffer is the bottom line of the Emacs window, where all prompting and multiple-character input is directed. If you are prompted for information in the minibuffer, such as a filename, Emacs can help you complete your answer if you type SPC or TAB. A second SPC or TAB will list all possibilities at that point. The minibuffer is also where you enter Emacs function names after typing M-x. For example, in the first paragraph, I mentioned that you could obtain help with C-h t (`help-with-tutorial`). What this means is that you can get a tutorial by typing either C-h t or M-x help-with-tutorial. In the latter case, you are prompted for `help-with-tutorial' in the minibuffer after typing M-x.

In case of trouble: Emacs can be interrupted at any time with C-g. For example, if you've started a command that requests that you enter something in the minibuffer, but then you change your mind, type C-g and you'll be back where you started. If you want to exit Emacs entirely, use C-x C-c.

Getting Started

Because there are many old versions of mh-e out there, it is important to know which version you have. I'll be talking about Version 5 which is similar to Version 4 and vastly different from Version 3.

First, enter M-x load-library RET mh-e RET. (4) The message, `Loading mh-e...done', should be displayed in the minibuffer. If you get `Cannot open load file: mh-e', then your Emacs is very badly configured, or mh-e is missing. You may wish to have your system administrator install a new Emacs or at least the latest mh-e files.

Having loaded mh-e successfully, enter M-x mh-version RET. The version of mh-e should be displayed. Hopefully it says that you're running Version 5.0.2 which is the latest version as of this printing. If instead Emacs beeps and says `[No match]', then you're running an old version of mh-e.

If these tests reveal a non-existent or old version of mh-e, please consider obtaining a new version. You can have your system administrator upgrade the system-wide version, or you can install your own personal version. It's really quite easy; instructions for getting and installing mh-e are in section Getting mh-e. In the meantime, see section Changes to mh-e, which compares the old and new names of commands, functions, variables, and buffers.

Also, older versions of mh-e assumed that you had already set up your MH environment. Newer versions set up a new MH environment for you by running `install-mh` and notifying you of this fact with the message in a temporary buffer:

```
I'm going to create the standard MH path for you.
```

Therefore, if you've never run MH before and you're using an old version of mh-e, you need to run `install-mh` from the shell before you continue the tour. If you don't, you'll be greeted with the error message: `Can't find MH profile'.

If, during the tour described in this chapter, you see a message like: `Searching for program: no such file or directory, /usr/local/bin/mhpath', it means that the MH programs and files are kept in a nonstandard directory.

In this case, simply add the following to `~/.emacs' and restart emacs.

```
(setq mh-progs "/path/to/MH/binary/directory/")
(setq mh-lib "/path/to/MH/library/directory/")
```

The `~' notation used by `~/.emacs' above represents your home directory. This is used by the bash and csh shells. If your shell does not support this feature, you could use the environment variable `$HOME' (such as `$HOME/.emacs') or the absolute path (as in `/home/wohler/.emacs') instead.

At this point, you should see something like the screen in the figure in section Receiving Mail. We're now ready to move on.

Sending Mail

Let's start our tour by sending ourselves a message which we can later read and process. Enter M-x mh-smail to invoke the mh-e program to send messages. You will be prompted in the minibuffer by `To:'. Enter your login name. The next prompt is `cc:'. Hit RET to indicate that no carbon copies are to be sent. At the `Subject:' prompt, enter Test or anything else that comes to mind.

Once you've specified the recipients and subject, your message appears in an Emacs buffer whose mode (5) is MH-Letter. Enter some text in the body of the message, using normal Emacs commands. You should now have something like this: (6)

```
-----Emacs: *scratch*         (Lisp Interaction)--All--------------------
To: wohler
cc:
Subject: Test
--------
  This is a test message to get the wheels churning...#


--**-{draft}      (MH-Letter)--All-----------------------------------------
```

mh-e message composition window

Note the line of dashes that separates the header and the body of the message. It is essential that these dashes (or a blank line) are present or the body of your message will be considered to be part of the header.

There are several commands specific to MH-Letter mode, but at this time we'll only use C-c C-c to send your message. Type C-c C-c now. That's all there is to it!

Receiving Mail

To read the mail you've just sent yourself, enter M-x mh-rmail. This incorporates the new mail and put the output from `inc` (called scan lines after the MH program `scan` which prints a one-line summary of each message) into a buffer called `+inbox' whose major mode is MH-Folder.

NOTE

> The M-x mh-rmail command will show you only new mail, not old mail. If you were to run this tour again, you would use M-r to pull all your messages into mh-e.

You should see the scan line for your message, and perhaps others. Use n or p to move the cursor to your test message and type RET to read your message. You should see something like:

```
   3  24Aug  root       received fax files on Wed Aug 24 11:00:13 PDT 1994
#  4+ 24Aug  To:wohler  Test<<This is a test message to get the wheels chu

--%%-{+inbox} 4 msgs (1-4)      (MH-Folder Show)--Bot--------------------
To: wohler
Subject: Test
Date: Wed, 24 Aug 1994 13:01:13 -0700
From: Bill Wohler <wohler@newt.com>

  This is a test message to get the wheels churning...




-----{show-+inbox} 4      (MH-Show)--Bot---------------------------------
```

After incorporating new messages

If you typed a long message, you can view subsequent pages with SPC and previous pages with DEL.

Processing Mail

The first thing we want to do is reply to the message that we sent ourselves. Ensure that the cursor is still on the same line as your test message and type r. You are prompted in the minibuffer with `Reply to whom:'. Here mh-e is asking whether you'd like to reply to the original sender only, to the sender and primary recipients, or to the sender and all recipients. If you simply hit RET, you'll reply only to the sender. Hit RET now.

You'll find yourself in an Emacs buffer similar to that when you were sending the original message, like this:

```
To: wohler
Subject: Re: Test
In-reply-to: Bill Wohler's message of Wed, 24 Aug 1994 13:01:13 -0700
```


```
                <199408242001.NAA00505@newt.com>
--------
#

--**-{draft}      (MH-Letter)--All-----------------------------------------
To: wohler
Subject: Test
Date: Wed, 24 Aug 1994 13:01:13 -0700
From: Bill Wohler <wohler@newt.com>

  This is a test message to get the wheels churning...

-----{show-+inbox} 4      (MH-Show)--Bot-------------------------------------
Composing a reply...done
```
Composition window during reply

By default, MH will not add you to the address list of your replies, so if you find that the `To:' header field is missing, don't worry. In this case, type C-c C-f C-t to create and go to the `To:' field, where you can type your login name again. You can move around with the arrow keys or with C-p (`previous-line`), C-n (`next-line`), C-b (`backward-char`), and C-f (`forward-char`) and can delete the previous character with DEL. When you're finished editing your message, send it with C-c C-c as before.

You'll often want to save messages that were sent to you in an organized fashion. This is done with folders. You can use folders to keep messages from your friends, or messages related to a particular topic. With your cursor in the MH-Folder buffer and positioned on the message you sent to yourself, type o to output (`refile` in MH parlance) that message to a folder. Enter test at the `Destination:' prompt and type y (or SPC) when mh-e asks to create the folder `+test'. Note that a `^' (caret) appears next to the message number, which means that the message has been marked for refiling but has not yet been refiled. We'll talk about how the refile is actually carried out in a moment.

Your previous reply is now waiting in the system mailbox. You incorporate this mail into your MH-Folder buffer named `+inbox' with the i command. Do this now. After the mail is incorporated, use n or p to move the cursor to the new message, and read it with RET. Let's delete this message by typing d. Note that a `D' appears next to the message number. This means that the message is marked for deletion but is not yet deleted. To perform the deletion (and the refile we did previously), use the x command.

If you want to send another message you can use m instead of M-x mh-smail. So go ahead, send some mail to your friends!

Leaving mh-e

You may now wish to exit `emacs` entirely. Use C-x C-c to exit `emacs`. If you exited without running x in the `+inbox' buffer, Emacs will offer to save it for you. Type y or SPC to save `+inbox' changes, which means to perform any refiles and deletes that you did there.

If you don't want to leave Emacs, you can type q to bury (hide) the mh-e folder or delete them entirely with C-x k. You can then later recall them with C-x b or M-x mh-rmail.

More About mh-e

These are the basic commands to get you going, but there are plenty more. If you think that mh-e is for you, read section Using mh-e and section Customizing mh-e to find out how you can:

- Print your messages. (section Printing Your Mail and section Printing Your Mail.)
- Edit messages and include your signature. (section Editing a Draft and section Editing a Draft.)
- Forward messages. (section Forwarding Mail and section Forwarding Mail.)
- Read digests. (section Viewing Your Mail.)
- Edit bounced messages. (section Editing Old Drafts and Bounced Messages and section Editing Old Drafts and Bounced Messages.)
- Send multimedia messages. (section Editing Multimedia Messages and section Editing Multimedia Messages.)
- Process mail that was sent with `shar` or `uuencode`. (section Files and Pipes.)
- Use sequences conveniently. (section Using Sequences.)
- Show header fields in different fonts. (section Viewing Your Mail.)
- Find previously refiled messages. (section Searching Through Messages.)
- Place messages in a file. (section Files and Pipes.)

Remember that you can also use MH commands when you're not running mh-e (and when you are!).

Go to the previous, next section.

Go to the previous, next section.

Using mh-e

This chapter leaves the tutorial style and goes into more detail about every mh-e command. The default, or "out of the box," behavior is documented. If this is not to your liking (for instance, you print with something other than `lpr`), see the associated section in section Customizing mh-e which is organized exactly like this chapter.

There are many commands, but don't get intimidated. There are command summaries at the beginning of each section. In case you have or would like to rebind the keys, the command summaries also list the associated Emacs Lisp function. Furthermore, even if you're stranded on a desert island with a laptop and are without your manuals, you can get a summary of all these commands with GNU Emacs online help: use C-h m (`describe-mode`) for a brief summary of commands or C-h i to read this manual via Info. The online help is quite good; try running C-h C-h C-h. This brings up a list of available help topics, one of which displays the documentation for a given key (like C-h k C-n). In addition, review section GNU Emacs Terms and Conventions, if any of the GNU Emacs conventions are strange to you.

Let's get started!

Reading Your Mail

The mh-e entry point for reading mail is M-x mh-rmail. This command incorporates your mail and creates a buffer called `+inbox' in MH-Folder mode. The M-x mh-rmail command shows you only new mail, not old mail. (7) The `+inbox' buffer contains scan lines, which are one-line summaries of each incorporated message. You can perform most MH commands on these messages via one-letter commands discussed in this chapter. See `scan`(1) for a description of the contents of the scan lines, and see the Figure in section Receiving Mail, for an example.

RET
: Display a message (`mh-show`).

SPC
: Go to next page in message (`mh-page-msg`).

DEL
: Go to previous page in message (`mh-previous-page`).

, (comma)
: Display a message with all header fields (`mh-header-display`).

M-SPC
: Go to next message in digest (`mh-page-digest`).

M-DEL
: Go to previous message in digest (`mh-page-digest-backwards`).

M-b
: Break up digest into separate messages (`mh-burst-digest`).

n
: Display next message (`mh-next-undeleted-msg`).

p
: Display previous message (`mh-previous-undeleted-msg`).

g
: Go to a message (`mh-goto-msg`).

M-<
: Go to first message (`mh-first-msg`).

M->
: Go to last message (`mh-last-msg`).

t
: Toggle between MH-Folder and MH-Folder Show modes (`mh-toggle-showing`).

Viewing Your Mail

The RET (`mh-show`) command displays the message that the cursor is on. If the message is already displayed, it scrolls to the beginning of the message. Use SPC (`mh-page-msg`) and DEL (`mh-previous-page`) to move forwards and backwards one page at a time through the message. You can give either of these commands a prefix argument that specifies the number of lines to scroll (such as 10 SPC). mh-e normally hides a lot of the superfluous header fields that mailers add to a message, but if you wish to see all of them, use the , (comma; `mh-header-display`) command.

Reading Digests

A digest is a message that contains other messages. Special mh-e commands let you read digests conveniently. You can use SPC and DEL to page through the digest as if it were a normal message, but if you wish to skip to the next message in the digest, use M-SPC (`mh-page-digest`). To return to a previous message, use M-DEL (`mh-page-digest-backwards`).

Another handy command is M-b (`mh-burst-digest`). This command uses the MH command `burst` to break out each message in the digest into its own message. Using this command, you can quickly delete unwanted messages, like this: Once the digest is split up, toggle out of MH-Folder Show mode with t (see section Moving Around) so that the scan lines fill the screen and messages aren't displayed. Then use d (see section Deleting Your Mail) to quickly delete messages that you don't want to read (based on the `Subject:' header field). You can also burst the digest to reply directly to the people who posted the messages in the digest. One problem you may encounter is that the `From:' header fields are preceded with a `>' so that your reply can't create the `To:' field correctly. In this case, you must correct the `To:' field yourself. This is described later in section Editing Textual Messages.

Reading Multimedia Mail

MH has the ability to read MIME (Multipurpose Internet Mail Extensions) messages. Unfortunately, mh-e does not yet have this ability, so you have to use the MH commands `show` or `mhn` from the shell to read MIME messages. (8)

Moving Around

To move on to the next message, use the n (`mh-next-undeleted-msg`) command; use the p (`mh-previous-undeleted-msg`) command to read the previous message. Both of these commands can be given a prefix argument to specify how many messages to skip (for example, 5 n). You can also move to a specific message with g (`mh-goto-msg`). You can enter the message number either before or after typing g. In the latter case, Emacs prompts you. Finally, you can go to the first or last message with M-< (`mh-first-msg`) and M-> (`mh-last-msg`) respectively.

You can also use the Emacs commands C-p (`previous-line`) and C-n (`next-line`) to move up and down the scan lines in the MH-Folder window. These commands can be used in conjunction with RET to look at deleted or refiled messages.

The command t (`mh-toggle-showing`) switches between MH-Folder mode and MH-Folder Show mode. (9) MH-Folder mode turns off the associated show buffer so that you can perform operations on the messages quickly without reading them. This is an excellent way to prune out your junk mail or to refile a group of messages to another folder for later examination.

Sending Mail

You can send a mail message in several ways. You can call M-x mh-smail directly, or from the command line like this:

```
% emacs -f mh-smail
```

From within mh-e's MH-Folder mode, other methods of sending mail are available as well:

m
: Compose a message (`mh-send`).

r
: Reply to a message (`mh-reply`).

f
: Forward message(s) (`mh-forward`).

M-d
: Redistribute a message (`mh-redistribute`).

M-e
: Edit a message that was bounced by mailer (`mh-extract-rejected-mail`).

M-a
: Edit a message to send it again (`mh-edit-again`).

From within a MH-Folder buffer, you can simply use the command m (`mh-send`). However you invoke `mh-send`, you are prompted for the `To:', `cc:', and `Subject:' header fields. Once you've specified the recipients and subject, your message appears in an Emacs buffer whose mode is MH-Letter (see the Figure in section Sending Mail to see what the buffer looks like). MH-Letter mode allows you to edit your message, to check the validity of the recipients, to insert other messages into your message, and to send the message. We'll go more into depth about editing a draft (10) (a message you're composing) in just a moment.

TOC

`mh-smail` always creates a two-window layout with the current buffer on top and the draft on the bottom. If you would rather preserve the window layout, use M-x mh-smail-other-window.

Replying to Mail

To compose a reply to a message, use the r (`mh-reply`) command. If you supply a prefix argument (as in C-u r), the message you are replying to is inserted in your reply after having first been run through `mhl` with the format file `` `mhl.reply' ``. See `mhl`(1) to see how you can modify the default `` `mhl.reply' `` file.

When you reply to a message, you are first prompted with `Reply to whom?'. You have several choices here.

Response	Reply Goes To
from	The person who sent the message. This is the default, so RET is sufficient.
to	Replies to the sender, plus all recipients in the `To:' header field.
all cc	Forms a reply to the sender, plus all recipients.

Depending on your answer, `repl` is given a different argument to form your reply. Specifically, a choice of from or none at all runs `repl -nocc all`, and a choice of to runs `repl -cc to`. Finally, either cc or all runs `repl -cc all -nocc me`.

Two windows are then created. One window contains the message to which you are replying. Your draft, in MH-Letter mode (described in section Editing a Draft), is in the other window.

If you wish to customize the header or other parts of the reply draft, please see `repl`(1) and `mh-format`(5).

Forwarding Mail

To forward a message, use the f (`mh-forward`) command. You are given a draft to edit that looks like it would if you had run the MH command `forw`. You are given a chance to add some text (see section Editing a Draft).

You can forward several messages by using a prefix argument; in this case, you are prompted for the name of a sequence, a symbolic name that represents a list or range of message numbers (for example, C-u f forbob RET). All of the messages in the sequence are inserted into your draft. By the way, although sequences are often mentioned in this chapter, you don't have to worry about them for now; the full description of sequences in mh-e is at the end in section Using Sequences. To learn more about sequences in general, please see `mh-sequence`(5).

Redistributing Your Mail

The command M-d (`mh-redistribute`) is similar in function to forwarding mail, but it does not allow you to edit the message, nor does it add your name to the `From:' header field. It appears to the recipient as if the message had come from the original sender. For more information on redistributing messages, see `dist`(1). Also investigate the M-a (`mh-edit-again`) command in section Editing Old Drafts and Bounced Messages, for another way to redistribute messages.

Editing Old Drafts and Bounced Messages

If you don't complete a draft for one reason or another, and if the draft buffer is no longer available, you can pick your draft up again with M-a (`mh-edit-again`). If you don't use a draft folder, your last `` `draft' `` file will be used. If you use draft folders, you'll need to visit the draft folder with M-f drafts RET, use n to move to the appropriate message, and then use M-a to prepare the message for editing.

The M-a command can also be used to take messages that were sent to you and to send them to more people.

Don't use M-a to re-edit a message from a *Mailer-Daemon* who complained that your mail wasn't posted for some reason or another. In this case, use M-e (`mh-extract-rejected-mail`) to prepare the message for editing by removing the *Mailer-Daemon* envelope and unneeded header fields. Fix whatever addressing problem you had, and send the message again with C-c C-c.

Editing a Draft

When you edit a message that you want to send (called a draft in this case), the mode used is MH-Letter. This mode provides several commands in addition to the normal Emacs editing commands to help you edit your draft.

C-c C-y
: Insert contents of message to which you're replying (`mh-yank-cur-msg`).

C-c C-i
: Insert a message from a folder (`mh-insert-letter`).

C-c C-f C-t
: Move to `To:' header field (`mh-to-field`).

C-c C-f C-c
: Move to `cc:' header field (`mh-to-field`).

C-c C-f C-s
: Move to `Subject:' header field (`mh-to-field`).

C-c C-f C-f
: Move to `From:' header field (`mh-to-field`).

C-c C-f C-b
: Move to `Bcc:' header field (`mh-to-field`).

C-c C-f C-f
: Move to `Fcc:' header field (`mh-to-fcc`).

C-c C-f C-d
: Move to `Dcc:' header field (`mh-to-field`).

C-c C-w
: Display expanded recipient list (`mh-check-whom`).

C-c C-s
: Insert signature in message (`mh-insert-signature`).

C-c C-m C-f
: Include forwarded message (MIME) (`mh-mhn-compose-forw`).

C-c C-m C-e
: Include anonymous ftp reference (MIME) (`mh-mhn-compose-anon-ftp`).

C-c C-m C-t
: Include anonymous ftp reference to compressed tar file (MIME) (`mh-mhn-compose-external-compressed-tar`).

C-c C-m C-i
: Include binary, image, sound, etc. (MIME) (`mh-mhn-compose-insertion`).

C-c C-e
: Run through `mhn` before sending (`mh-edit-mhn`).

C-c C-m C-u
: Undo effects of `mhn` (`mh-revert-mhn-edit`).

C-c C-c
: Save draft and send message (`mh-send-letter`).

C-c C-q
: Quit editing and delete draft message (`mh-fully-kill-draft`).

Editing Textual Messages

The following sections show you how to edit a draft. The commands described here are also applicable to messages that have multimedia components.

Inserting letter to which you're replying

It is often useful to insert a snippet of text from a letter that someone mailed to provide some context for your reply. The command C-c C-y (`mh-yank-cur-msg`) does this by yanking a portion of text from the message to which you're replying and inserting `> ' before each line.

You can control how much text is included when you run this command. If you run this command right away, without entering the buffer containing the message to you, this command will yank the entire message, as is, into your reply. (11) If you enter the buffer containing the message sent to you and move the cursor to a certain point and return to your reply and run C-c C-y, then the text yanked will range from that point to the end of the message. Finally, the most common action you'll perform is to enter the message sent to you, move the cursor to the beginning of a paragraph or phrase, set the mark with C-SPC or C-@, and move the cursor to the end of the paragraph or phrase. The cursor position is called the point, and the space between the mark and point is called the region. Having done that, C-c C-y will insert the region you selected.

Inserting messages

Messages can be inserted with C-c C-i (`mh-insert-letter`). This command prompts you for the folder and message number and inserts the message, indented by `> '. Certain undesirable header fields are removed before insertion. If given a prefix argument (like C-u C-c C-i), the header is left intact, the message is not indented, and `> ' is not inserted before each line.

Editing the header

Because the header is part of the message, you can edit the header fields as you wish. However, several convenience functions exist to help you create and edit them. For example, the command C-c C-f C-t (`mh-to-field`; alternatively, C-c C-f t) moves the cursor to the `To:' header field, creating it if necessary. The functions to move to the `cc:', `Subject:', `From:', `Bcc:', and `Dcc:' header fields are similar.

One function behaves differently from the others, namely, C-c C-f C-f (`mh-to-fcc`; alternatively, C-c C-f f). This function will prompt you for the folder name in which to file a copy of the draft.

Be sure to leave a row of dashes or a blank line between the header and the body of the message.

Checking recipients

The C-c C-w (`mh-check-whom`) command expands aliases so you can check the actual address(es) in the alias. A new buffer is created with the output of `whom`.

Inserting your signature

You can insert your signature at the current cursor location with the C-c C-s (`mh-insert-signature`) command. The text of your signature is taken from the file `` `~/.signature' ``.

Editing Multimedia Messages

mh-e has the capability to create multimedia messages. It uses the MIME (Multipurpose Internet Mail Extensions) protocol. The MIME protocol allows you to incorporate images, sound, video, binary files, and even commands that fetch a file with `ftp' when your recipient reads the message! If you were to create a multimedia message with plain MH commands, you would use `mhn`. Indeed, the mh-e MIME commands merely insert `mhn` directives which are later expanded by `mhn`.

Each of the mh-e commands for editing multimedia messages or for incorporating multimedia objects is prefixed with C-c C-m .

Several MIME objects are defined. They are called content types. The table in section Editing a Draft contains a list of the content types that mh-e currently knows about. Several of the mh-e commands fill in the content type for you, whereas others require you to enter one. Most of the time, it should be obvious which one to use (e.g., use image/jpeg to include a JPEG image). If not, you can refer to RFC 1521, (12) which defines the MIME protocol, for a list of valid content types.

You are also sometimes asked for a content description. This is simply an optional brief phrase, in your own words, that describes the object. If you don't care to enter a content description, just press return and none will be included; however, a reader may skip over multimedia fields unless the content description is compelling.

Remember: you can always add `mhn` directives by hand.

Forwarding multimedia messages

Mail may be forwarded with MIME using the command C-c C-m C-f (`mh-mhn-compose-forw`). You are prompted for a content description, the name of the folder in which the messages to forward are located, and the messages' numbers.

Including an ftp reference

You can even have your message initiate an `ftp` transfer when the recipient reads the message. To do this, use the C-c C-m C-e (`mh-mhn-compose-anon-ftp`) command. You are prompted for the remote host and pathname, the content type, and the content description.

Including tar files

If the remote file (see section Including an ftp reference) is a compressed tar file, you can use C-c C-m C-t (`mh-mhn-compose-external-compressed-tar`). Then, in addition to retrieving the file via anonymous *ftp*, the file will also be uncompressed and untarred. You are prompted for the remote host and pathname and the content description. The pathname should contain at least one `/' (slash), because the pathname is broken up into directory and name components.

Including other multimedia objects

Images, sound, and video can be inserted in your message with the C-c C-m C-i (`mh-mhn-compose-insertion`) command. You are prompted for the filename containing the object, the content type, and a content description of the object.

Readying multimedia messages for sending

When you are finished editing a MIME message, it might look like this:

```
   3  24Aug  root                 received fax files on Wed Aug 24 11:00:13
   4+ 24Aug  To:wohler            Test<<This is a test message to get the wh




--%%-{+inbox} 4 msgs (1-4)      (MH-Folder Show)--Bot------------------
To: wohler
cc:
Subject: Test of MIME
--------
#@application/octet-stream [Nonexistent ftp test file] \
access-type=anon-ftp; site=berzerk.com; name=panacea.tar.gz; \
directory="/pub/"
#audio/basic [Test sound bite] /tmp/noise.au
```

```
--**-{draft}      (MH-Letter)--All-----------------------------------
```

TOC

mh-e MIME draft

The lines added by the previous commands are `mhn` directives and need to be converted to MIME directives before sending. This is accomplished by the command C-c C-e (`mh-edit-mhn`), which runs `mhn` on the message. The following screen shows what those commands look like in full MIME format. You can see why mail user agents are usually built to hide these details from the user.

```
To: wohler
cc:
Subject: Test of MIME
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary="----- =_aaaaaaaaaa0"
Content-ID: <1623.777796162.0@newt.com>

------- =_aaaaaaaaaa0
Content-Type: message/external-body; access-type="anon-ftp";
        site="berzerk.com"; name="panacea.tar.gz"; directory="/pub/"

Content-Type: application/octet-stream
Content-ID: <1623.777796162.1@newt.com>
Content-Description: Nonexistent ftp test file

------- =_aaaaaaaaaa0
Content-Type: audio/basic
Content-ID: <1623.777796162.2@newt.com>
Content-Description: Test sound bite
Content-Transfer-Encoding: base64

Q3JlYXRpdmUgVm9pY2UgRmlsZRoaAAoBKREBQh8AgwCAgH9/f35+fn59fX5+fn5+f3/f3
f4B/f39/f39/f39/f39+f39+f39/f39/f4B/f39/fn5/f39/f3+Af39/f39/gH9/f39/fn
-----{draft}      (MH-Letter)--Top-----------------------------------
```

mh-e MIME draft ready to send

This action can be undone by running C-c C-m C-u (`mh-revert-mhn-edit`). It does this by reverting to a backup file. You are prompted to confirm this action, but you can avoid the confirmation by adding an argument (for example, C-u C-c C-m C-u).

Sending a Message

When you are all through editing a message, you send it with the C-c C-c (`mh-send-letter`) command. You can give an argument (as in C-u C-c C-c) to monitor the first stage of the delivery.

Killing the Draft

If for some reason you are not happy with the draft, you can kill it instead with C-c C-q (`mh-fully-kill-draft`). Emacs then kills the draft buffer and deletes the draft message.

Moving Your Mail Around

This section covers how messages and folders can be moved about or manipulated. Messages may be incorporated into your `` `+inbox' ``, deleted, and refiled. Messages containing `shar` or `uuencode` output can be stored. Folders can be visited, sorted, packed, or deleted. Here's a list of the available commands to do these things:

i
: Incorporate new mail into folder (`mh-inc-folder`).

d
: Delete message (`mh-delete-msg`).

C-d
: Delete message, don't move to next message (`mh-delete-msg-no-motion`).

M-s
: Find messages that meet search criteria (`mh-search-folder`).

o
: Output (refile) message to folder (`mh-refile-msg`).

c
: Copy message to folder (`mh-copy-msg`).

C-o
: Output (write) message to file (`mh-write-msg-to-file`).

!
: Repeat last output command (`mh-refile-or-write-again`).

l
: Print message with `lpr` (`mh-print-msg`).

|
: Pipe message through shell command (`mh-pipe-msg`).

M-n
: Unpack message created with `uudecode` or `shar` (`mh-store-msg`).

M-l
: List all folders (`mh-list-folders`).

M-f
: Visit folder (`mh-visit-folder`).

M-r
: Regenerate scan lines (`mh-rescan-folder`).

M-x mh-sort-folder

Sort folder.

M-p

Pack folder (`mh-pack-folder`).

M-k

Remove folder (`mh-kill-folder`).

x

Execute pending refiles and deletes (`mh-execute-commands`).

u

Undo pending refile or delete (`mh-undo`).

M-u

Undo all pending refiles and deletes (`mh-undo-folder`).

q

Quit (`mh-quit`).

Incorporating Your Mail

If at any time you receive new mail, incorporate the new mail into your `+inbox' buffer with i (`mh-inc-folder`). Note that i will display the `+inbox' buffer, even if there isn't any new mail. You can incorporate mail from any file into the current folder by specifying a prefix argument; you'll be prompted for the name of the file to use (for example, C-u i ~/mbox RET).

Emacs can notify you when you have new mail by displaying `Mail' in the mode line. To enable this behavior, and to have a clock in the mode line besides, add the following to `~/.emacs':

```
(display-time)
```

Deleting Your Mail

To mark a message for deletion, use the d (`mh-delete-msg`) command. A `D' is placed by the message in the scan window, and the next message is displayed. If the previous command had been p, then the next message displayed is the message previous to the message just deleted. If you specify a prefix argument, you will be prompted for a sequence (see section Using Sequences) to delete (for example, C-u d frombob RET). The x command actually carries out the deletion (see section Finishing Up). C-d (`mh-delete-msg-no-motion`) marks the message for deletion but leaves the cursor at the current message in case you wish to perform other operations on the message.

Organizing Your Mail with Folders

mh-e has analogies for each of the MH `folder` and `refile` commands. To refile a message in another folder, use the o (`mh-refile-msg`) (mnemonic: "output") command. You are prompted for the folder name.

If you are refiling several messages into the same folder, you can use the ! (`mh-refile-or-write-again`) command to repeat the last refile or write (see the description of C-o in

section Files and Pipes). Or, place the messages into a sequence (section Using Sequences) and specify a prefix argument to o, in which case you'll be prompted for the name of the sequence (for example, C-u o search RET).

If you wish to copy a message to another folder, you can use the c (`mh-copy-msg`) command (see the `-link` argument to `refile`(1)). You are prompted for a folder, and you can specify a prefix argument if you want to copy a sequence into another folder. In this case, you are then prompted for the sequence. Note that unlike the o command, the copy takes place immediately. The original copy remains in the current folder.

When you want to read the messages that you have refiled into folders, use the M-f (`mh-visit-folder`) command to visit the folder. You are prompted for the folder name.

Other commands you can perform on folders include: M-l (`mh-list-folders`), to list all the folders in your mail directory; M-k (`mh-kill-folder`), to remove a folder; M-x mh-sort-folder, to sort the messages by date (see `sortm`(1) to see how to sort by other criteria); M-p (`mh-pack-folder`), to pack a folder, removing gaps from the numbering sequence; and M-r (`mh-rescan-folder`), to rescan the folder, which is useful to grab all messages in your `` `+inbox' `` after processing your new mail for the first time. If you don't want to rescan the entire folder, give M-r or M-p a prefix argument and you'll be prompted for a range of messages to display (for instance, C-u M-r last:50 RET).

Printing Your Mail

Printing mail is simple. Enter l (`mh-print-msg`) (for *l*ine printer or *l*pr). The message is formatted with `mhl` and printed with the `lpr` command. You can print all the messages in a sequence by specifying a prefix argument, in which case you are prompted for the name of the sequence (as in C-u l frombob RET).

Files and Pipes

mh-e does offer a couple of commands that are not a part of MH. The first one, C-o (`mh-write-msg-to-file`), writes a message to a file (think of the o as in "output"). You are prompted for the filename. If the file already exists, the message is appended to it. You can also write the message to the file without the header by specifying a prefix argument (such as C-u C-o /tmp/foobar RET). Subsequent writes to the same file can be made with the ! command.

You can also pipe the message through a Unix shell command with the | (`mh-pipe-msg`) command. You are prompted for the Unix command through which you wish to run your message. If you give an argument to this command, the message header is included in the text passed to the command (the contrived example C-u | lpr would be done with the l command instead).

If the message is a shell archive `shar` or has been run through `uuencode` use M-n (`mh-store-msg`) to extract the body of the message. The default directory for extraction is the current directory, and you have a chance to specify a different extraction directory. The next time you use this command, the default directory is the last directory you used.

Finishing Up

If you've deleted a message or refiled it, but changed your mind, you can cancel the action before you've executed it. Use u (`mh-undo`) to undo a refile on or deletion of a single message. You can also undo refiles and deletes for messages that belong to a given sequence by specifying a prefix argument. You'll be prompted

for the name of the sequence (as in C-u u frombob RET). Alternatively, you can use M-u (`mh-undo-folder`) to undo all refiles or deletes in the current folder.

If you've marked messages to be deleted or refiled and you want to go ahead and delete or refile the messages, use x (`mh-execute-commands`). Many mh-e commands that may affect the numbering of the messages (such as M-r or M-p) will ask if you want to process refiles or deletes first and then either run x for you or undo the pending refiles and deletes, which are lost.

When you want to quit using mh-e and go back to editing, you can use the q (`mh-quit`) command. This buries the buffers of the current mh-e folder and restores the buffers that were present when you first ran M-x mh-rmail. You can later restore your mh-e session by selecting the `+inbox' buffer or by running M-x mh-rmail again.

Searching Through Messages

You can search a folder for messages to or from a particular person or about a particular subject. In fact, you can also search for messages containing selected strings in any arbitrary header field or any string found within the messages. Use the M-s (`mh-search-folder`) command. You are first prompted for the name of the folder to search and then placed in the following buffer in MH-Pick mode:

```
From: #
To:
Cc:
Date:
Subject:
--------




--**-Emacs: pick-pattern    (MH-Pick)------All----------------------------
```

Pick window

Edit this template by entering your search criteria in an appropriate header field that is already there, or create a new field yourself. If the string you're looking for could be anywhere in a message, then place the string underneath the row of dashes. The M-s command uses the MH command `pick` to do the real work, so read `pick`(1) to find out more about how to enter the criteria.

There are no semantics associated with the search criteria--they are simply treated as strings. Case is ignored when all lowercase is used, and regular expressions (a la `ed`) are available. It is all right to specify several search criteria. What happens then is that a logical *and* of the various fields is performed. If you prefer a logical *or* operation, run M-s multiple times.

As an example, let's say that we want to find messages from Ginnean about horseback riding in the Kosci National Park (Australia) during January, 1994. Normally we would start with a broad search and narrow down if necessary to produce a manageable amount of data, but we'll cut to the chase and create a fairly restrictive set of criteria as follows:

```
From: ginnean
To:
Cc:
Date: Jan 1994
Subject: horse.*kosciusko
--------
```

As with MH-Letter mode, MH-Pick provides commands like C-c C-f C-t to help you fill in the blanks.

C-c C-f C-t
: Move to `To:' header field (`mh-to-field`).

C-c C-f C-c
: Move to `cc:' header field (`mh-to-field`).

C-c C-f C-s
: Move to `Subject:' header field (`mh-to-field`).

C-c C-f C-f
: Move to `From:' header field (`mh-to-field`).

C-c C-f C-b
: Move to `Bcc:' header field (`mh-to-field`).

C-c C-f C-f
: Move to `Fcc:' header field (`mh-to-field`).

C-c C-f C-d
: Move to `Dcc:' header field (`mh-to-field`).

C-c C-c
: Execute the search (`mh-do-pick-search`).

To perform the search, type C-c C-c (`mh-do-pick-search`). The selected messages are placed in the *search* sequence, which you can use later in forwarding (see section Forwarding Mail), printing (see section Printing Your Mail), or narrowing your field of view (see section Using Sequences). Subsequent searches are appended to the *search* sequence. If, however, you wish to start with a clean slate, first delete the *search* sequence (how to do this is discussed in section Using Sequences).

If you're searching in a folder that is already displayed in a MH-Folder buffer, only those messages contained in the buffer are used for the search. Therefore, if you want to search in all messages, first kill the folder's buffer with C-x k or scan the entire folder with M-r.

Using Sequences

For the whole scoop on MH sequences, refer to `mh-sequence`(5). As you've read, several of the mh-e commands can operate on a sequence, which is a shorthand for a range or group of messages. For example, you might want to forward several messages to a friend or colleague. Here's how to manipulate sequences.

%
: Put message in a sequence (`mh-put-msg-in-seq`).

?
: Display sequences that message belongs to (`mh-msg-is-in-seq`).

M-q
: List all sequences in folder (`mh-list-sequences`).

M-%
: Remove message from sequence (`mh-delete-msg-from-seq`).

M-#
: Delete sequence (`mh-delete-seq`).

C-x n
: Restrict display to messages in sequence (`mh-narrow-to-seq`).

C-x w
: Remove restriction; display all messages (`mh-widen`).

M-x mh-update-sequences
: Push mh-e's state out to MH.

To place a message in a sequence, use % (`mh-put-msg-in-seq`) to do it manually, or use the MH command `pick` or the mh-e version of `pick` (section Searching Through Messages) which create a sequence automatically. Give % a prefix argument and you can add all the messages in one sequence to another sequence (for example, C-u % SourceSequence RET).

Once you've placed some messages in a sequence, you may wish to narrow the field of view to just those messages in the sequence you've created. To do this, use C-x n (`mh-narrow-to-seq`). You are prompted for the name of the sequence. What this does is show only those messages that are in the selected sequence in the MH-Folder buffer. In addition, it limits further mh-e searches to just those messages. When you want to widen the view to all your messages again, use C-x w (`mh-widen`).

You can see which sequences a message is in with the ? (`mh-msg-is-in-seq`) command. Or, you can list all sequences in a selected folder (default is current folder) with M-q (`mh-list-sequences`).

If you want to remove a message from a sequence, use M-% (`mh-delete-msg-from-seq`), and if you want to delete an entire sequence, use M-# (`mh-delete-seq`). In the latter case you are prompted for the sequence to delete. Note that this deletes only the sequence, not the messages in the sequence. If you want to delete the messages, use C-u d (see section Deleting Your Mail above).

Two sequences are maintained internally by mh-e and pushed out to MH when you type either the x or q command. They are the sequence specified by your `Unseen-Sequence:' profile entry and *cur*. However, you can also just update MH's state with the command M-x mh-update-sequences. See section Viewing Your Mail

for an example of how this command might be used.

With the exceptions of C-x n and C-x w, the underlying MH command dealing with sequences is `mark`.

Miscellaneous Commands

One other command worth noting is M-x mh-version. Since there were a few changes in command letters between Versions 3 and 4, use this command to see which version you are running. This command didn't exist before Version 4, so the message `[No match]' indicates that it's time to upgrade (see section Getting mh-e). In the meantime, use the older commands that are listed in section Changes to mh-e. The output of M-x mh-version should also be included with any bug report you send (see section Bug Reports).

Go to the previous, next section.

Go to the previous, next section.

Customizing mh-e

Until now, we've talked about the mh-e commands as they work "out of the box." Of course, it is also possible to reconfigure mh-e beyond recognition. The following sections describe all of the customization variables, show the defaults, and make recommendations for customization. The outline of this chapter is identical to that of section Using mh-e, to make it easier to find the variables you'd need to modify to affect a particular command.

However, when customizing your mail environment, first try to change what you want in MH, and only change mh-e if changing MH is not possible. That way you will get the same behavior inside and outside GNU Emacs. Note that mh-e does not provide hooks for customizations that can be done in MH; this omission is intentional.

Many string or integer variables are easy enough to modify using Emacs Lisp. Any such modifications should be placed in a file called `.emacs' in your home directory (that is, `~/.emacs'). For example, to modify the variable that controls printing, you could add:

```
(setq mh-lpr-command-format "nenscript -G -r -2 -i'%s'")
```

section Printing Your Mail talks more about this variable.

Variables can also hold Boolean values. In Emacs Lisp, the Boolean values are `nil`, which means false, and `t`, which means true. Usually, variables are turned off by setting their value to `nil`, as in

```
(setq mh-bury-show-buffer nil)
```

which keeps the MH-Show buffer at the top of the buffer stack. To turn a variable on, you use

```
(setq mh-bury-show-buffer t)
```

which places the MH-Show buffer at the bottom of the buffer stack. However, the text says to turn on a variable by setting it to a *non-`nil`* value, because sometimes values other than `t` are meaningful (for example, see `mhl-formfile`, described in section Viewing Your Mail). Other variables, such as hooks, involve a little more Emacs Lisp programming expertise.

You can also "preview" the effects of changing variables before committing the changes to `~/.emacs'. Variables can be changed in the current Emacs session by using M-x set-variable.

In general, commands in this text refer to Emacs Lisp functions. Programs outside of Emacs are specifically called MH commands, shell commands, or Unix commands.

I hope I've included enough examples here to get you well on your way. If you want to explore Emacs Lisp further, a programming manual does exist, (13) and you can look at the code itself for examples. Look in the Emacs Lisp directory on your system (such as `/usr/local/lib/emacs/lisp') and find all the `mh-*.el' files there. When calling mh-e and other Emacs Lisp functions directly from Emacs Lisp code, you'll need to know the correct arguments. Use the online help for this. For example, try C-h f mh-execute-commands RET. If you write your own functions, please do not prefix your symbols (variables and functions) with `mh-`. This prefix is reserved for the mh-e package. To avoid conflicts with existing mh-e symbols, use a prefix like `my-` or your initials.

Reading Your Mail

I'll start out by including a function that I use as a front end to mh-e. (14) It toggles between your working window configuration, which may be quite involved--windows filled with source, compilation output, man pages, and other documentation--and your mh-e window configuration. Like the rest of the customization described in this chapter, simply add the following code to `~/.emacs'. Don't be intimidated by the size of this example; most customizations are only one line.

@filbreak

Starting mh-e

```
(defvar my-mh-screen-saved nil
  "Set to non-nil when mh-e window configuration shown.")
(defvar my-normal-screen nil "Normal window configuration.")
(defvar my-mh-screen nil "mh-e window configuration.")

(defun my-mh-rmail (&optional arg)
  "Toggle between mh-e and normal screen configurations.
With non-nil or prefix argument, inc mailbox as well
when going into mail."
  (interactive "P")                ; user callable function, P=prefix arg
  (setq my-mh-screen-saved         ; save state
        (cond
         ;; Bring up mh-e screen if arg or normal window configuration.
         ;; If arg or +inbox buffer doesn't exist, run mh-rmail.
         ((or arg (null my-mh-screen-saved))
          (setq my-normal-screen (current-window-configuration))
          (if (or arg (null (get-buffer "+inbox")))
              (mh-rmail)
            (set-window-configuration my-mh-screen))
          t)                       ; set my-mh-screen-saved to t
         ;; Otherwise, save mh-e screen and restore normal screen.
         (t
          (setq my-mh-screen (current-window-configuration))
          (set-window-configuration my-normal-screen)
          nil))))                  ; set my-mh-screen-saved to nil

(global-set-key "\C-x\r" 'my-mh-rmail)  ; call with C-x RET
```

If you type an argument (C-u) or if `my-mh-screen-saved` is `nil` (meaning a non-mh-e window configuration), the current window configuration is saved, either +inbox is displayed or `mh-rmail` is run, and the mh-e window configuration is shown. Otherwise, the mh-e window configuration is saved and the original configuration is displayed.

Now to configure mh-e. The following table lists general mh-e variables and variables that are used while reading mail.

`mh-progs`

Directory containing MH programs (default: dynamic).

`mh-lib`
: Directory containing MH support files and programs (default: dynamic).

`mh-do-not-confirm`
: Don't confirm on non-reversible commands (default: `nil`).

`mh-summary-height`
: Number of scan lines to show (includes mode line) (default: 4).

`mh-folder-mode-hook`
: Functions to run in MH-Folder mode (default: `nil`).

`mh-clean-message-header`
: Remove extraneous headers (default: `nil`).

`mh-invisible-headers`
: Headers to hide (default: `'"^Received: \\| ^Message-Id: \\| ^Remailed-\\| ^Via: \\| ^Mail-from: \\| ^Return-Path: \\| ^In-Reply-To: \\| ^Resent-'").

`mh-visible-headers`
: Headers to display (default: `nil`).

`mhl-formfile`
: Format file for `mhl` (default: `nil`).

`mh-show-hook`
: Functions to run when showing message (default: `nil`).

`mh-show-mode-hook`
: Functions to run when showing message (default: `nil`).

`mh-bury-show-buffer`
: Leave show buffer at bottom of stack (default: `t`).

`mh-show-buffer-mode-line-buffer-id`
: Name of show buffer in mode line (default: `'"{show-%s} %d"').

The two variables `mh-progs` and `mh-lib` are used to tell mh-e where the MH programs and supporting files are kept, respectively. mh-e does try to figure out where they are kept for itself by looking in common places and in the user's `PATH' environment variable, but if it cannot find the directories, or finds the wrong ones, you should set these variables. The name of the directory should be placed in double quotes, and there should be a trailing slash (`/'). See the example in section Getting Started.

If you never make mistakes, and you do not like confirmations for your actions, you can set `mh-do-not-confirm` to a non-`nil` value to disable confirmation for unrecoverable commands such as M-k (`mh-kill-folder`) and M-u (`mh-undo-folder`). Here's how you set boolean values:

```
(setq mh-do-not-confirm t)
```

The variable `mh-summary-height` controls the number of scan lines displayed in the MH-Folder window, including the mode line. The default value of 4 means that 3 scan lines are displayed. Here's how you set numerical values:

```
(setq mh-summary-height 2)              ; only show the current scan line
```

Normally the buffer for displaying messages is buried at the bottom at the buffer stack. You may wish to disable

this feature by setting `mh-bury-show-buffer` to `nil`. One advantage of not burying the show buffer is that one can delete the show buffer more easily in an electric buffer list because of its proximity to its associated MH-Folder buffer. Try running M-x electric-buffer-list to see what I mean.

The hook `mh-folder-mode-hook` is called when a new folder is created with MH-Folder mode. This could be used to set your own key bindings, for example:

```
Create additional key bindings via mh-folder-mode-hook

(defvar my-mh-init-done nil "Non-nil when one-time mh-e settings made.")

(defun my-mh-folder-mode-hook ()
  "Hook to set key bindings in MH-Folder mode."
  (if (not my-mh-init-done)             ; only need to bind the keys once
      (progn
        (local-set-key "/" 'search-msg)
        (local-set-key "b" 'mh-burst-digest)    ; better use of b
        (setq my-mh-init-done t))))

;;; Emacs 19
(add-hook 'mh-folder-mode-hook 'my-mh-folder-mode-hook)
;;; Emacs 18
;;;   (setq mh-folder-mode-hook (cons 'my-mh-folder-mode-hook
;;;                                   mh-folder-mode-hook))

(defun search-msg ()
  "Search for a regexp in the current message."
  (interactive)                         ; user function
  (save-window-excursion
    (other-window 1)                    ; go to next window
    (isearch-forward-regexp)))          ; string search; hit return (ESC
                                        ;   in Emacs 18) when done
```

Viewing Your Mail

Several variables control what displayed messages look like. Normally messages are delivered with a handful of uninteresting header fields. You can make them go away by setting `mh-clean-message-header` to a non-`nil` value. The header can then be cleaned up in two ways. By default, the header fields in `mh-invisible-headers` are removed. On the other hand, you could set `mh-visible-headers` to the fields that you would like to see. If this variable is set, `mh-invisible-headers` is ignored. I suggest that you not set `mh-visible-headers` since if you use this variable, you might miss a lot of header fields that you'd rather not miss. As an example of how to set a string variable, `mh-visible-headers` can be set to show a minimum set of header fields (see (section 'Syntax of Regular Expressions' in The GNU Emacs Manual, for a description of the special characters in this string):

```
(setq mh-visible-headers "^From: \\|^Subject: \\|^Date: ")
```

Normally mh-e takes care of displaying messages itself (rather than calling an MH program to do the work). If you'd rather have `mhl` display the message (within mh-e), set the variable `mhl-formfile` to a non-`nil` value. You can set this variable either to `t` to use the default format file or to a filename if you have your own format file

(`mhl`(1) tells you how to write one). When writing your own format file, use a nonzero value for `overflowoffset` to ensure the header is RFC 822 compliant and parsable by mh-e. `mhl` is always used for printing and forwarding; in this case, the value of `mhl-formfile` is consulted if it is a filename.

Two hooks can be used to control how messages are displayed. The first hook, `mh-show-mode-hook`, is called early on in the process of displaying of messages. It is used to perform some actions on the contents of messages, such as highlighting the header fields. If you're running Emacs 19 under the X Window System, the following example will highlight the `From:' and `Subject:' header fields. This is a very nice feature indeed.

Emphasize header fields in different fonts via mh-show-mode-hook

```
(defvar my-mh-keywords
   '(("^From: \\(.*\\)" 1 'bold t)
     ("^Subject: \\(.*\\)" 1 'highlight t))
  "mh-e additions for font-lock-keywords.")

(defun my-mh-show-mode-hook ()
  "Hook to turn on and customize fonts."
  (require 'font-lock)                  ; for font-lock-keywords below
  (make-local-variable 'font-lock-mode-hook) ; don't affect other buffers
  (add-hook 'font-lock-mode-hook        ; set a hook with inline function
            (function                   ; modifies font-lock-keywords when
             (lambda ()                 ; font-lock-mode run
               (setq font-lock-keywords
                     (append my-mh-keywords font-lock-keywords)))))
  (font-lock-mode 1))                   ; change the typefaces

(if window-system                       ; can't do this on ASCII terminal
    (add-hook 'mh-show-mode-hook 'my-mh-show-mode-hook))
```

The second hook, `mh-show-hook`, is the last thing called after messages are displayed. It's used to affect the behavior of mh-e in general or when `mh-show-mode-hook` is too early. For example, if you wanted to keep mh-e in sync with MH, you could use `mh-show-hook` as follows:

```
(add-hook 'mh-show-hook 'mh-update-sequences)
```

The function `mh-update-sequences` is documented in section Finishing Up. For those who like to modify their mode lines, use `mh-show-buffer-mode-line-buffer-id` to modify the mode line in the MH-Show buffers. Place the two escape strings `%s' and `%d', which will display the folder name and the message number, respectively, somewhere in the string in that order. The default value of `"{show-%s} %d"' yields a mode line of

```
-----{show-+inbox} 4      (MH-Show)--Bot---------------------------------
```

Moving Around

When you use t (`mh-toggle-showing`) to toggle between show mode and scan mode, the MH-Show buffer is hidden and the MH-Folder buffer is left alone. Setting `mh-recenter-summary-p` to a non-`nil` value causes the toggle to display as many scan lines as possible, with the cursor at the middle. The effect of `mh-recenter-summary-p` is rather useful, but it can be annoying on a slow network connection.

Sending Mail

You may wish to start off by adding the following useful key bindings to your `.emacs' file:

```
(global-set-key "\C-xm" 'mh-smail)
(global-set-key "\C-x4m" 'mh-smail-other-window)
```

In addition, several variables are useful when sending mail or replying to mail. They are summarized in the following table.

`mh-comp-formfile`
: Format file for drafts (default: `"components"').

`mh-repl-formfile`
: Format file for replies (default: `"replcomps"').

`mh-letter-mode-hook`
: Functions to run in MH-Letter mode (default: `nil`).

`mh-compose-letter-function`
: Functions to run when starting a new draft (default: `nil`).

`mh-reply-default-reply-to`
: Whom reply goes to (default: `nil`).

`mh-forward-subject-format`
: Format string for forwarded message subject (default: `"%s: %s"').

`mh-redist-full-contents`
: `send` requires entire message (default: `nil`).

`mh-new-draft-cleaned-headers`
: Remove these header fields from re-edited draft (default: `"^Date:\\| ^Received:\\| ^Message-Id:\\| ^From:\\| ^Sender:\\| ^Delivery-Date:\\| ^Return-Path:"').

Since mh-e does not use `comp` to create the initial draft, you need to set `mh-comp-formfile` to the name of your components file if it isn't `components'. This is the name of the file that contains the form for composing messages. If it does not contain an absolute pathname, mh-e searches for the file first in your MH directory and then in the system MH library directory (such as `/usr/local/lib/mh'). Replies, on the other hand, are built using `repl`. You can change the location of the field file from the default of `replcomps' by modifying `mh-repl-formfile`.

Two hooks are provided to run commands on your freshly created draft. The first hook, `mh-letter-mode-hook`, allows you to do some processing before editing a letter. For example, you may wish to modify the header after `repl` has done its work, or you may have a complicated `components' file and need to tell mh-e where the cursor should go. Here's an example of how you would use this hook--all of the other hooks are set in this fashion as well.

```
Prepare draft for editing via mh-letter-mode-hook

(defvar letter-mode-init-done nil
  "Non-nil when one-time mh-e settings have made.")

(defun my-mh-letter-mode-hook ()
```


```
  "Hook to prepare letter for editing."
  (if (not letter-mode-init-done)    ; only need to bind the keys once
      (progn
        (local-set-key "\C-ctb" 'add-enriched-text)
        (local-set-key "\C-cti" 'add-enriched-text)
        (local-set-key "\C-ctf" 'add-enriched-text)
        (local-set-key "\C-cts" 'add-enriched-text)
        (local-set-key "\C-ctB" 'add-enriched-text)
        (local-set-key "\C-ctu" 'add-enriched-text)
        (local-set-key "\C-ctc" 'add-enriched-text)
        (setq letter-mode-init-done t)))
  (setq fill-prefix "  ")            ; I find indented text easier to read
  (save-excursion
    (goto-char (point-max))          ; go to end of message to
    (mh-insert-signature)))          ;   insert signature

(add-hook 'mh-letter-mode-hook 'my-mh-letter-mode-hook)
```

The function, `add-enriched-text` is defined in the example in section Editing Multimedia Messages.

The second hook, a function really, is `mh-compose-letter-function`. Like `mh-letter-mode-hook`, it is called just before editing a new message; however, it is the last function called before you edit your message. The consequence of this is that you can write a function to write and send the message for you. This function is passed three arguments: the contents of the `To:', `Subject:', and `cc:' header fields.

Replying to Mail

If you find that most of the time that you specify cc when you reply to a message, set `mh-reply-default-reply-to` to `cc'. This variable is normally set to `nil` so that you are prompted for the recipient of a reply. It can be set to one of `from', `to', or `cc'; you are then no longer prompted for the recipient(s) of your reply.

Forwarding Mail

When forwarding a message, the format of the `Subject:' header field can be modified by the variable `mh-forward-subject-format`. This variable is a string which includes two escapes (`%s'). The first `%s' is replaced with the sender of the original message, and the second one is replaced with the original `Subject:'. The default value of `"%s: %s"' takes a message with the header:

```
To: Bill Wohler <wohler@newt.com>
Subject: Re: 49er football
From: Greg DesBrisay <gd@cellnet.com>
```

and creates a subject header field of:

```
Subject: Greg DesBrisay: Re: 49er football
```

TOC

Redistributing Your Mail

The variable `mh-redist-full-contents` must be set to non-`nil` if `dist` requires the whole letter for redistribution, which is the case if `send` is compiled with the BERK (15) option (which many people abhor). If you find that MH will not allow you to redistribute a message that has been redistributed before, this variable should be set to `nil`.

Editing Old Drafts and Bounced Messages

The header fields specified by `mh-new-draft-cleaned-headers` are removed from an old draft that has been recreated with M-e (`mh-extract-rejected-mail`) or M-a (`mh-edit-again`). If when you edit an old draft with these commands you find that there are header fields that you don't want included, you can append them to this variable. For example,

```
(setq mh-new-draft-cleaned-headers
      (concat mh-new-draft-cleaned-headers "\\|^Some-Field:"))
```

This appends the regular expression `\\|^Some-Field:' to the variable (see section 'Syntax of Regular Expressions' in The GNU Emacs Manual). The `\\|' means *or*, and the `^' (caret) matches the beginning of the line. This is done to be very specific about which fields match. The literal `:' is appended for the same reason.

Editing a Draft

There are several variables used during the draft editing phase. Examples include changing the name of the file that holds your signature or telling mh-e about new multimedia types. They are:

`mh-yank-from-start-of-msg`
: How to yank when region not set (default: `t`).

`mh-ins-buf-prefix`
: Indent for yanked messages (default: `"> "').

`mail-citation-hook`
: Functions to run on yanked messages (default: `nil`).

`mh-delete-yanked-msg-window`
: Delete message window on yank (default: `nil`).

`mh-mime-content-types`
: List of valid content types (default: `'(("text/plain")
("text/richtext") ("multipart/mixed") ("multipart/alternative")
("multipart/digest") ("multipart/parallel") ("message/rfc822")
("message/partial") ("message/external-body")
("application/octet-stream") ("application/postscript")
("image/jpeg") ("image/gif") ("audio/basic") ("video/mpeg"))').

`mh-mhn-args`
: Additional arguments for `mhn` (default: `nil`).

`mh-signature-file-name`
: File containing signature (default: `"~/.signature"').

`mh-before-send-letter-hook`

Functions to run before sending draft (default: `nil`).

`mh-send-prog`

MH program used to send messages (default: `"send"').

Editing Textual Messages

The following two sections include variables that customize the way you edit a draft. The discussion here applies to editing multimedia messages as well.

Inserting letter to which you're replying

To control how much of the message to which you are replying is yanked by C-c C-y (`mh-yank-cur-msg`) into your reply, modify `mh-yank-from-start-of-msg`. The default value of `t` means that the entire message is copied. If it is set to `'body` (don't forget the apostrophe), then only the message body is copied. If it is set to `nil`, only the part of the message following point (the current cursor position in the message's buffer) is copied. In any case, this variable is ignored if a region is set in the message you are replying to. The string contained in `mh-ins-buf-prefix` is inserted before each line of a message that is inserted into a draft with C-c C-y (`mh-yank-cur-msg`). I suggest that you not modify this variable. The default value of `"> '" is the default string for many mailers and news readers: messages are far easier to read if several included messages have all been indented by the same string. The variable `mail-citation-hook` is `nil` by default, which means that when a message is inserted into the letter, each line is prefixed by `mh-ins-buf-prefix`. Otherwise, it can be set to a function that modifies an included citation. (16) If you like to yank all the text from the message you're replying to in one go, set `mh-delete-yanked-msg-window` to non-`nil` to delete the window containing the original message after yanking it to make more room on your screen for your reply.

Inserting your signature

You can change the name of the file inserted with C-c C-s (`mh-insert-signature`) by changing `mh-signature-file-name` (default: `"~/.signature"').

Editing Multimedia Messages

The variable `mh-mime-content-types` contains a list of the currently valid content types. They are listed in the table in section Editing a Draft. If you encounter a new content type, you can add it like this:

```
(setq mh-mime-content-types (append mh-mime-content-types
                                    '(("new/type"))))
```

Emacs macros can be used to insert enriched text directives like `<bold>'. The following code will make, for example, C-c t b insert the `<bold>' directive.

```
Emacs macros for entering enriched text

(defvar enriched-text-types '(("b" . "bold") ("i" . "italic") ("f" . "fixed")
                              ("s" . "smaller") ("B" . "bigger")
                              ("u" . "underline") ("c" . "center"))
  "Alist of (final-character . directive) choices for add-enriched-text.
Additional types can be found in RFC 1563.")
```


```
(defun add-enriched-text (begin end)
  "Add enriched text directives around region.
The directive used comes from the list enriched-text-types and is
specified by the last keystroke of the command.  When called from Lisp,
arguments are BEGIN and END."
  (interactive "r")
  ;; Set type to the directive indicated by the last keystroke.
  (let ((type (cdr (assoc (char-to-string (logior last-input-char ?`))
                          enriched-text-types))))
    (save-restriction              ; restores state from narrow-to-region
      (narrow-to-region begin end) ; narrow view to region
      (goto-char (point-min))      ; move to beginning of text
      (insert "<" type ">")        ; insert beginning directive
      (goto-char (point-max))      ; move to end of text
      (insert "</" type ">"))))    ; insert terminating directive
```

To use the function `add-enriched-text`, first create keybindings for it (see section Sending Mail). Then, set the mark with C-@ or C-SPC, type in the text to be highlighted, and type C-c t b. This adds `<bold>' where you set the mark and adds `</bold>' at the location of your cursor, giving you something like: `You should be <bold>very</bold>'. You may also be interested in investigating `sgml-mode`.

Readying multimedia messages for sending

If you wish to pass additional arguments to `mhn` to affect how it builds your message, use the variable `mh-mhn-args`. For example, you can build a consistency check into the message by setting `mh-mhn-args` to `-check`. The recipient of your message can then run `mhn -check` on the message---`mhn` will complain if the message has been corrupted on the way. The C-c C-e (`mh-mhn-edit`) command only consults this variable when given a prefix argument.

Sending a Message

If you want to check your spelling in your message before sending, use `mh-before-send-letter-hook` like this:

Spell-check message via mh-before-send-letter-hook

```
(add-hook 'mh-before-send-letter-hook 'ispell-message)
```

In case the MH `send` program is installed under a different name, use `mh-send-prog` to tell mh-e the name.

Moving Your Mail Around

If you change the name of some of the MH programs or have your own printing programs, the following variables can help you. They are described in detail in the subsequent sections.

`mh-inc-prog`

Program to incorporate mail (default: `'"inc"').

`mh-inc-folder-hook`

Functions to run when incorporating mail (default: `nil`).

`mh-delete-msg-hook`
: Functions to run when deleting messages (default: `nil`).

`mh-print-background`
: Print in foreground or background (default: `nil`).

`mh-lpr-command-format`
: Command used to print (default: `'"lpr -J '%s'"').

`mh-default-folder-for-message-function`
: Function to generate a default folder (default: `nil`).

`mh-auto-folder-collect`
: Collect folder names in background at startup (default: `t`).

`mh-recursive-folders`
: Collect nested folders (default: `nil`).

`mh-refile-msg-hook`
: Functions to run when refiling message (default: `nil`).

`mh-store-default-directory`
: Default directory for storing files created by `uuencode` or `shar` (default: `nil`).

`mh-sortm-args`
: Additional arguments for `sortm` (default: `nil`).

`mh-scan-prog`
: Program to scan messages (default: `'"scan"').

`mh-before-quit-hook`
: Functions to run before quitting (default: `nil`). See also `mh-quit-hook`.

`mh-quit-hook`
: Functions to run after quitting (default: `nil`). See also `mh-before-quit-hook`.

Incorporating Your Mail

The name of the program that incorporates new mail is stored in `mh-inc-prog`; it is `'"inc"' by default. This program generates a one-line summary for each of the new messages. Unless it is an absolute pathname, the file is assumed to be in the `mh-progs` directory. You may also link a file to `inc` that uses a different format (see `mh-profile`(5)). You'll then need to modify several variables appropriately; see `mh-scan-prog` below. You can set the hook `mh-inc-folder-hook`, which is called after new mail is incorporated by the i (`mh-inc-folder`) command. A good use of this hook is to rescan the whole folder either after running M-x mh-rmail the first time or when you've changed the message numbers from outside of mh-e.

Rescan folder after incorporating new mail via mh-inc-folder-hook

```
(defun my-mh-inc-folder-hook ()
  "Hook to rescan folder after incorporating mail."
  (if (buffer-modified-p)            ; if outstanding refiles and deletes,
      (mh-execute-commands))         ;   carry them out
  (mh-rescan-folder)                 ; synchronize with +inbox
  (mh-show))                         ; show the current message
```


```
(add-hook 'mh-inc-folder-hook 'my-mh-inc-folder-hook)
```

Deleting Your Mail

The hook `mh-delete-msg-hook` is called after you mark a message for deletion. For example, the current maintainer of mh-e used this once when he kept statistics on his mail usage.

Organizing Your Mail with Folders

By default, operations on folders work only one level at a time. Set `mh-recursive-folders` to non-`nil` to operate on all folders. This mostly means that you'll be able to see all your folders when you press TAB when prompted for a folder name. The variable `mh-auto-folder-collect` is normally turned on to generate a list of folder names in the background as soon as mh-e is loaded. Otherwise, the list is generated when you need a folder name the first time (as with o (`mh-refile-msg`)). If you have a lot of folders and you have `mh-recursive-folders` set, this could take a while, which is why it's nice to do the folder collection in the background.

The function `mh-default-folder-for-message-function` is used by o (`mh-refile-msg`) and C-c C-f C-f (`mh-to-fcc`) to generate a default folder. The generated folder name should be a string with a `+' before it. For each of my correspondents, I use the same name for both an alias and a folder. So, I wrote a function that takes the address in the `From:' header field, finds it in my alias file, and returns the alias, which is used as a default folder name. This is the most complicated example given here, and it demonstrates several features of Emacs Lisp programming. You should be able to drop this into `~/.emacs', however. If you use this to store messages in a subfolder of your Mail directory, you can modify the line that starts `(format +%s...' and insert your subfolder after the folder symbol `+'.

@filbreak

Creating useful default folder for refiling via mh-default-folder-for-message-function

```
(defun my-mh-folder-from-address ()
  "Determine folder name from address.
Takes the address in the From: header field, and returns its corresponding
alias from the user's personal aliases file. Returns nil if the address
was not found."
  (require 'rfc822)                        ; for the rfc822 functions
  (search-forward-regexp "^From: \\(.*\\)") ; grab header field contents
  (save-excursion                          ; save state
    (let ((addr (car (rfc822-addresses  ; get address
                      (buffer-substring (match-beginning 1)
                                        (match-end 1)))))
          (buffer (get-buffer-create " *temp*")) ; set local variables
          folder)
      (set-buffer buffer)                  ; jump to temporary buffer
      (unwind-protect                      ; run kill-buffer when done
          (progn                           ; function grouping construct
            (insert-file-contents (expand-file-name "aliases"
                                                    mh-user-path))
            (goto-char (point-min))   ; grab aliases file and go to start
```


```
            (setq folder
                  ;; Search for the given address, even commented-out
                  ;; addresses are found!
                  ;; The function search-forward-regexp sets values that are
                  ;; later used by match-beginning and match-end.
                  (if (search-forward-regexp (format "^;*\\(.*\\):.*%s"
                                                     addr) nil t)
                      ;; NOTE WELL: this is what the return value looks like.
                      ;; You can modify the format string to match your own
                      ;; Mail hierarchy.
                      (format "+%s" (buffer-substring (match-beginning 1)
                                                      (match-end 1))))))
              (kill-buffer buffer))          ; get rid of our temporary buffer
            folder)))                        ; function's return value

(setq mh-default-folder-for-message-function 'my-mh-folder-from-address)
```

The hook `mh-refile-msg-hook` is called after a message is marked to be refiled.

The variable `mh-sortm-args` holds extra arguments to pass on to the `sortm` command. Note: this variable is only consulted when a prefix argument is given to M-x mh-sort-folder. It is used to override any arguments given in a `sortm:` entry in your MH profile (`` `~/.mh_profile' ``).

Scan line formatting

The name of the program that generates a listing of one line per message is held in `mh-scan-prog` (default: `"scan"'). Unless this variable contains an absolute pathname, it is assumed to be in the `mh-progs` directory. You may link another program to `scan` (see `mh-profile`(5)) to produce a different type of listing.

If you change the format of the scan lines you'll need to tell mh-e how to parse the new format. As you see, quite a lot of variables are involved to do that. The first variable has to do with pruning out garbage.

`mh-valid-scan-line`

> This regular expression describes a valid scan line. This is used to eliminate error messages that are occasionally produced by `inc` or `scan` (default: `"^ *[0-9]"').

Next, two variables control how the message numbers are parsed.

`mh-msg-number-regexp`

> This regular expression is used to extract the message number from a scan line. Note that the message number must be placed in quoted parentheses, (\\(...\\)), as in the default of `"^ *\\([0-9]+\\)"'.

`mh-msg-search-regexp`

> Given a message number (which is inserted in `%d'), this regular expression will match the scan line that it represents (default: `"^[^0-9]*%d[^0-9]"').

Finally, there are a slew of variables that control how mh-e marks up the scan lines.

`mh-cmd-note`

> Number of characters to skip over before inserting notation (default: 4). Note how it relates to the following regular expressions.

`mh-deleted-msg-regexp`

> This regular expression describes deleted messages (default: `"^....D"'). See also `mh-note-deleted`.

`mh-refiled-msg-regexp`

This regular expression describes refiled messages (default: `'"^....\\^"'). See also `mh-note-refiled`.

`mh-cur-scan-msg-regexp`

This regular expression matches the current message (default: `'"^....\\+"'). See also `mh-note-cur`.

`mh-good-msg-regexp`

This regular expression describes which messages should be shown when mh-e goes to the next or previous message. Normally, deleted or refiled messages are skipped over (default: `'"^....[^D^]"').

`mh-note-deleted`

Messages that have been deleted to are marked by this string (default: `'"D"'). See also `mh-deleted-msg-regexp`.

`mh-note-refiled`

Messages that have been refiled are marked by this string (default: `'"^"'). See also `mh-refiled-msg-regexp`.

`mh-note-copied`

Messages that have been copied are marked by this string (default: `'"C"').

`mh-note-cur`

The current message (in MH, not in mh-e) is marked by this string (default: `'"+"'). See also `mh-cur-scan-msg-regexp`.

`mh-note-repl`

Messages that have been replied to are marked by this string (default: `'"-"').

`mh-note-forw`

Messages that have been forwarded are marked by this string (default: `'"F"').

`mh-note-dist`

Messages that have been redistributed are marked by this string (default: `'"R"').

`mh-note-printed`

Messages that have been printed are marked by this string (default: `'"P"').

`mh-note-seq`

Messages in a sequence are marked by this string (default: `'"%"').

Printing Your Mail

Normally messages are printed in the foreground. If this is slow on your system, you may elect to set `mh-print-background` to non-`nil` to print in the background. If you do this, do not delete the message until it is printed or else the output may be truncated. The variable `mh-lpr-command-format` controls how the printing is actually done. The string can contain one escape, `%s', which is filled with the name of the folder and the message number and is useful for print job names. As an example, the default is `'"lpr -J '%s'"'.

Files and Pipes

The initial directory for the `mh-store-msg` command is held in `mh-store-default-directory`. Since I almost always run `mh-store-msg` on sources, I set it to my personal source directory like this:

```
(setq mh-store-default-directory (expand-file-name "~/src/"))
```

Subsequent incarnations of `mh-store-msg` offer the last directory used as the default. By the way, `mh-store-msg` calls the Emacs Lisp function `mh-store-buffer`. I mention this because you can use it directly if you're editing a buffer that contains a file that has been run through `uuencode` or `shar`. For example, you can extract the contents of the current buffer in your home directory by typing M-x mh-store-buffer RET ~ RET.

Finishing Up

The two variables `mh-before-quit-hook` and `mh-quit-hook` are called by q (`mh-quit`). The former one is called before the quit occurs, so you might use it to perform any mh-e operations; you could perform some query and abort the quit or call `mh-execute-commands`, for example. The latter is not run in an mh-e context, so you might use it to modify the window setup.

Searching Through Messages

If you find that you do the same thing over and over when editing the search template, you may wish to bind some shortcuts to keys. This can be done with the variable `mh-pick-mode-hook`, which is called when M-s (`mh-search-folder`) is run on a new pattern.

The string `mh-partial-folder-mode-line-annotation` is used to annotate the mode line when only a portion of the folder is shown. For example, this will be displayed after running M-s (`mh-search-folder`) to list messages based on some search criteria (see section Searching Through Messages). The default annotation of `"select"' yields a mode line that looks like:

```
--%%-{+inbox/select} 2 msgs (2-3)      (MH-Folder)--All----------------
```

Go to the previous, next section.

Go to the previous, next section.

Odds and Ends

This appendix covers a few topics that don't fit elsewhere. Here I tell you how to report bugs and how to get on the mh-e mailing list. I also point out some additional sources of information.

Bug Reports

The current maintainer of mh-e is Stephen Gildea <*gildea@lcs.mit.edu*>. Please mail bug reports directly to him, as well as any praise or suggestions. Please include the output of M-x mh-version (see section Miscellaneous Commands) in any bug report you send.

mh-e Mailing List

There is a mailing list, *mh-e@x.org*, for discussion of mh-e and announcements of new versions. Send a "subscribe" message to *mh-e-request@x.org* to be added. Do not report bugs on this list; mail them directly to the maintainer (see section Bug Reports).

MH FAQ

An FAQ appears monthly in the newsgroup `comp.mail.mh'. While very little is there that deals with mh-e specifically, there is an incredible wealth of material about MH itself which you will find useful. The subject of the FAQ is MH Frequently Asked Questions (FAQ) with Answers.

The FAQ can be also obtained by anonymous `ftp` or via the World Wide Web (WWW). It is located at:

`ftp://rtfm.mit.edu/pub/usenet/news.answers/mail/mh-faq/part1`
`http://www.cis.ohio-state.edu/hypertext/faq/usenet/mail/mh-faq/part1/faq.html`

Otherwise, you can use mail. Send mail to *mail-server@rtfm.mit.edu* containing the following:

```
send usenet/news.answers/mail/mh-faq/part1
```

Getting mh-e

If you're running a pre-4.0 version of mh-e, please consider upgrading. You can either have your system administrator upgrade your Emacs, or just the files for mh-e.

The MH distribution contains a copy of mh-e in `` `miscellany/mh-e' ``. Make sure it is at least Version 4.0.

The latest version of mh-e can be obtained via anonymous `ftp` from `ftp.x.org'. The file containing mh-e is currently `` `/misc/mh-e/mh-e-5.0.2.tar.Z' `` I suggest that you extract the files from `` `mh-e-5.0.2.tar.Z' `` in the following fashion:


```
% cd                                  # Start in your home directory
% mkdir lib lib/emacs                 # Create directory for mh-e
% cd lib/emacs
% zcat path/to/mh-e-5.0.2.tar.Z | tar xvf -    # Extract files
```

To use these new files, add the following to `~/.emacs':

```
(setq load-path (cons (expand-file-name "~/lib/emacs") load-path))
```

That's it! If you're already running Emacs, please quit that session and start again to load in the new mh-e. Check that you're running the new version with the command M-x mh-version after running any mh-e command. The distribution comes with a file called `MH-E-NEWS' so you can see what's new.

Go to the previous, next section.

Go to the previous, next section.

History of mh-e

mh-e was originally written by Brian Reid in 1983 and has changed hands twice since then. Jim Larus wanted to do something similar for GNU Emacs, and ended up completely rewriting it that same year. In 1989, Stephen Gildea picked it up and is now currently improving and maintaining it.

From Brian Reid

One day in 1983 I got the flu and had to stay home from work for three days with nothing to do. I used that time to write MHE. The fundamental idea behind MHE was that it was a "puppeteer" driving the MH programs underneath it. MH had a model that the editor was supposed to run as a subprocess of the mailer, which seemed to me at the time to be the tail wagging the dog. So I turned it around and made the editor drive the MH programs. I made sure that the UCI people (who were maintaining MH at the time) took in my changes and made them stick.

Today, I still use my own version of MHE because I don't at all like the way that GNU mh-e works and I've never gotten to be good enough at hacking Emacs Lisp to make GNU mh-e do what I want. The Gosling-emacs version of MHE and the GNU Emacs version of mh-e have almost nothing in common except similar names. They work differently, have different conceptual models, and have different key bindings. (17)

Brian Reid, June 1994

From Jim Larus

Brian Reid, while at CMU or shortly after going to Stanford wrote a mail reading program called MHE for Gosling Emacs. It had much the same structure as mh-e (i.e., invoked MH programs), though it was simpler and the commands were slightly different. Unfortunately, I no longer have a copy so the differences are lost in the mists of time.

In '82-83, I was working at BBN and wrote a lot of mlisp code in Gosling Emacs to make it look more like Tennex Emacs. One of the packages that I picked up and improved was Reid's mail system. In '83, I went back to Berkeley. About that time, Stallman's first version of GNU Emacs came out and people started to move to it from Gosling Emacs (as I recall, the transition took a year or two). I decided to port Reid's MHE and used the mlisp to Emacs Lisp translator that came with GNU Emacs. It did a lousy job and the resulting code didn't work, so I bit the bullet and rewrote the code by hand (it was a lot smaller and simpler then, so it took only a day or two).

Soon after that, mh-e became part of the standard Emacs distribution and suggestions kept dribbling in for improvements. mh-e soon reached sufficient functionality to keep me happy, but I kept on improving it because I was a graduate student with plenty of time on my hands and it was more fun than my dissertation. In retrospect, the one thing that I regret is not writing any documentation, which seriously

limited the use and appeal of the package.

In '89, I came to Wisconsin as a professor and decided not to work on mh-e. It was stable, except for minor bugs, and had enough functionality, so I let it be for a few years. Stephen Gildea of BBN began to pester me about the bugs, but I ignored them. In 1990, he went off to the X Consortium, said good bye, and said that he would now be using `xmh`. A few months later, he came back and said that he couldn't stand `xmh` and could I put a few more bug fixes into mh-e. At that point, I had no interest in fixing mh-e, so I gave the responsibility of maintenance to him and he has done a fine job since then.

Jim Larus, June 1994

From Stephen Gildea

In 1987 I went to work for Bolt Beranek and Newman, as Jim had before me. In my previous job, I had been using RMAIL, but as my folders tend to run large, I was frustrated with the speed of RMAIL. However, I stuck with it because I wanted the GNU Emacs interface. I am very familiar and comfortable with the Emacs interface (with just a few modifications of my own) and dislike having to use applications with embedded editors; they never live up to Emacs.

MH is the mail reader of choice at BBN, so I converted to it. Since I didn't want to give up using an Emacs interface, I started using mh-e. As is my wont, I started hacking on it almost immediately. I first used version 3.4m. One of the first features I added was to treat the folder buffer as a file-visiting buffer: you could lock it, save it, and be warned of unsaved changes when killing it. I also worked to bring its functionality a little closer to RMAIL. Jim Larus was very cooperative about merging in my changes, and my efforts first appeared in version 3.6, distributed with Emacs 18.52 in 1988. Next I decided mh-e was too slow and optimized it a lot. Version, 3.7, distributed with Emacs 18.56 in 1990, was noticeably faster.

When I moved to the X Consortium I became the first person there to not use xmh. (There is now one other engineer there using mh-e.) About this point I took over maintenance of mh-e from Jim and was finally able to add some features Jim hadn't accepted, such as the backward searching undo. My first release was 3.8 (Emacs 18.58) in 1992.

Now, in 1994, we see a flurry of releases, with both 4.0 and 5.0. Version 4.0 added many new features, including background folder collection and support for composing MIME messages. (Reading MIME messages remains to be done, alas.) While writing this book, Bill Wohler gave mh-e its closest examination ever, uncovering bugs and inconsistencies that required a new major version to fix, and so version 5 was released.

Stephen Gildea, June 1994

Go to the previous, next section.

Go to the previous, next section.

Changes to mh-e

mh-e had a fairly major facelift between Versions 3 and 4. The differences between Versions 4 and 5 from the user's viewpoint are relatively minor. The prompting order for the folder and message number in a couple of functions had been switched inadvertently in Version 4. Version 5 switches the order back. The `+inbox' folder is no longer hard-coded, but rather uses the `Inbox' MH Profile entry. See the file `etc/MH-E-NEWS' in the Emacs distribution for more details on the changes.

This section documents the changes between Version 3 and newer versions so that you'll know which commands to use (or which commands you won't have) in case you're stuck with an old version.

The following tables summarize the changes to buffer names, commands and variables.

Buffer Mode Names

Version 3	Version 4
mh-e folder	MH-Folder
mh-e scan	MH-Folder
mh-e show	MH-Folder Show
Fundamental	MH-Show
mh-e letter	MH-Letter
mh-e letter	MH-Pick

Commands

Version 3		Version 4	
Function	**Command**	**Command**	**Function**
mh-first-msg	<	M-<	mh-first-msg
-	-	M->	mh-last-msg
mh-show	.	RET	mh-show
-	-	,	mh-header-display
mh-reply	a	r	mh-reply
mh-redistribute	r	M-d	mh-redistribute
mh-unshar-msg	-	M-n	mh-store-msg
mh-write-msg-to-file	M-o	C-o	mh-write-msg-to-file
mh-delete-msg-from-seq	C-u M-%	M-#	mh-delete-seq
-	-	M-q	mh-list-sequences
mh-quit	b	q	mh-quit
-	-	C-C C-f C-r	mh-to-field (`From:')

- - C-C C-f C-d mh-to-field (`Dcc:'

Variables

Version 3		Version 4	
Variable	**Value**	**Value**	**Variable**
mh-show-buffer-mode-line-buffer-id	"{%%b} %s/%d"	"{show-%s} %d"	mh-show-buffer-mode-line-buffer-id
mh-unshar-default-directory	""	nil	mh-store-default-directory

New Variables

mail-citation-hook
mail-header-separator
mh-auto-folder-collect
mh-comp-formfile
mh-repl-formfile
mh-delete-msg-hook
mh-forward-subject-format
mh-inc-prog
mh-mime-content-types
mh-default-folder-for-message-function
mh-mhn-args
mh-new-draft-cleaned-headers
mh-pick-mode-hook
mh-refile-msg-hook
mh-scan-prog
mh-send-prog
mh-show-hook
mh-show-mode-hook
mh-signature-file-name
mh-sortm-args
mh-repl-formfile

Go to the previous, next section.

Go to the previous, next section.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free

TOC

use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that wor not derived from the Program, and can be reasonably considered independent and separate wo in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
    1. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
    2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
    3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

    The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

    If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so

long as such parties remain in full compliance.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number

of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the

"copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type `show c'
for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program `Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License

instead of this License.

Go to the previous, next section.

TOC

Go to the previous, next section.

Command Index

d

m

Go to the previous, next section.

Go to the previous, next section.

Variable Index

m

Go to the previous, next section.

Go to the previous section.

Concept Index

.

b

c

d

e

f

TOC

m

n

o

p

t

u

v

w

x

~

Go to the previous section.

mh-e

TOC

(1)

Note that mh-e is supported with MH 6 and either Emacs 18 or Emacs 19. Reportedly, large parts of it work with MH 5 and also with Lucid/XEmacs and Epoch, but there are no guarantees. It is also distributed with Lucid/XEmacs, as well as with MH itself.

(2)

The keys mentioned in these chapters refer to the default key bindings. If you've changed the bindings, refer to the command summaries at the beginning of each major section in section Using mh-e, for a mapping between default key bindings and function names.

(3)

This is emphasized because pressing ESC twice or holding it down a second too long so that it repeats gives you an error message.

(4)

You wouldn't ordinarily do this.

(5)

A mode changes Emacs to make it easier to edit a particular type of text.

(6)

If you're running Emacs under the X Window System, then you would also see a menubar. I've left out the menubar in all of the example screens.

(7)

If you want to see your old mail as well, use M-r to pull all your messages into mh-e. Or, give a prefix argument to `mh-rmail` so it will prompt you for folder to visit like M-f (for example, C-u M-x mh-rmail RET bob RET). Both M-r and M-f are described in section Organizing Your Mail with Folders.

(8)

You can call them directly from Emacs if you're running the X Window System: type M-! xterm -e mhn message-number. You can leave out the `xterm -e` if you use `mhn -list` or `mhn -store`.

(9)

For you Emacs wizards, this is implemented as an Emacs minor mode.

(10)

I highly recommend that you use a draft folder so that you can edit several drafts in parallel. To do so, create a folder (e.g. `+drafts'), and add a profile component called `Draft-Folder:' which contains `+drafts' (see mh-profile(5)).

(11)

If you'd rather have the header cleaned up, use C-u r instead of r when replying (see section Replying to Mail).

(12)

This RFC (Request For Comments) is available via the URL `ftp://ds.internic.net/rfc/rfc1521.txt'.

(13)

The GNU Emacs Lisp Reference Manual may be available online in the Info system by typing C-h i m Emacs Lisp RET. If not, you can order a printed manual, which has the desirable side-effect of helping to support the Free Software Foundation which made all this great software available. You can find an order form by running C-h C-d, or you can request an order form from *gnu@prep.ai.mit.edu*.

(14)

Stephen Gildea's favorite binding is (global-set-key "\C-cr" 'mh-rmail).

(15)

To see which options your copy of MH was compiled with, use M-x mh-version (section Miscellaneous Commands).

(16)

Supercite is an example of a full-bodied, full-featured citation package. It is in Emacs versions 19.15 and later, and its URL is `ftp://archive.cis.ohio-state.edu/pub/gnu/emacs/elisp-archive/packages/sc3.1.tar.Z'

(17)

After reading this article, I questioned Brian about his version of MHE, and received some great ideas for improving mh-e such as a dired-like method of selecting folders; and removing the prompting when sending mail, filling in the blanks in the draft buffer instead. I passed them on to Stephen Gildea, the current maintainer, and he was excited about the ideas as well. Perhaps one day, mh-e will again resemble MHE, although none of these ideas are manifest in Version 5.0.

gzip

The data compression program

Edition 1.2.4, for Gzip Version 1.2.4

July 1993

by Jean-loup Gailly

gzip

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.
2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.
3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
   1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a w based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
    1. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
    2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
    3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

    The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

    If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.
5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to

refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,

INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type `show c'
for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program `Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Overview

`gzip` reduces the size of the named files using Lempel-Ziv coding (LZ77). Whenever possible, each file is replaced by one with the extension `.gz', while keeping the same ownership modes, access and modification times. (The default extension is `-gz' for VMS, `z' for MSDOS, OS/2 FAT and Atari.) If no files are specified or if a file name is "-", the standard input is compressed to the standard output. `gzip` will only attempt to compress regular files. In particular, it will ignore symbolic links.

If the new file name is too long for its file system, `gzip` truncates it. `gzip` attempts to truncate only the parts of the file name longer than 3 characters. (A part is delimited by dots.) If the name consists of small parts only, the longest parts are truncated. For example, if file names are limited to 14 characters, gzip.msdos.exe is compressed to gzi.msd.exe.gz. Names are not truncated on systems which do not have a limit on file name length.

By default, `gzip` keeps the original file name and timestamp in the compressed file. These are used when decompressing the file with the `-N' option. This is useful when the compressed file name was truncated or when the time stamp was not preserved after a file transfer.

Compressed files can be restored to their original form using `gzip -d' or `gunzip` or `zcat`. If the original name saved in the compressed file is not suitable for its file system, a new name is constructed from the original one to make it legal.

`gunzip` takes a list of files on its command line and replaces each file whose name ends with `.gz', `.z', `.Z', `-gz', `-z' or `_z' and which begins with the correct magic number with an uncompressed file without the original extension. `gunzip` also recognizes the special extensions `.tgz' and `.taz' as shorthands for `.tar.gz' and `.tar.Z' respectively. When compressing, `gzip` uses the `.tgz' extension if necessary instead of truncating a file with a `.tar' extension.

`gunzip` can currently decompress files created by `gzip`, `zip`, `compress` or `pack`. The detection of the input format is automatic. When using the first two formats, `gunzip` checks a 32 bit CRC (cyclic redundancy check). For `pack`, `gunzip` checks the uncompressed length. The `compress` format was not designed to allow consistency checks. However `gunzip` is sometimes able to detect a bad `.Z' file. If you get an error when uncompressing a `.Z' file, do not assume that the `.Z' file is correct simply because the standard `uncompress` does not complain. This generally means that the standard `uncompress` does not check its input, and happily generates garbage output. The SCO `compress -H' format (`lzh` compression method) does not include a CRC but also allows some consistency checks.

Files created by `zip` can be uncompressed by `gzip` only if they have a single member compressed with the 'deflation' method. This feature is only intended to help conversion of `tar.zip` files to the `tar.gz` format. To extract `zip` files with several members, use `unzip` instead of `gunzip`.

`zcat` is identical to `gunzip -c'. `zcat` uncompresses either a list of files on the command line or its standard input and writes the uncompressed data on standard output. `zcat` will uncompress files that have the correct magic number whether they have a `.gz' suffix or not.

`gzip` uses the Lempel-Ziv algorithm used in `zip` and PKZIP. The amount of compression obtained depends on the size of the input and the distribution of common substrings. Typically, text such as source code or English is reduced by 60-70%. Compression is generally much better than that achieved by LZW (as used in `compress`), Huffman coding (as used in `pack`), or adaptive Huffman coding (`compact`).

Compression is always performed, even if the compressed file is slightly larger than the original. The worst case expansion is a few bytes for the `gzip` file header, plus 5 bytes every 32K block, or an expansion ratio of 0.015% for large files. Note that the actual number of used disk blocks almost never increases. `gzip` preserves the mode, ownership and timestamps of files when compressing or decompressing.

Sample Output

Here are some realistic examples of running `gzip`.

This is the output of the command `gzip -h':

```
gzip 1.2.4 (18 Aug 93)
usage: gzip [-cdfhlLnNrtvV19] [-S suffix] [file ...]
 -c --stdout      write on standard output, keep original files unchanged
 -d --decompress  decompress
 -f --force       force overwrite of output file and compress links
 -h --help        give this help
 -l --list        list compressed file contents
 -L --license     display software license
 -n --no-name     do not save or restore the original name and time stamp
 -N --name        save or restore the original name and time stamp
 -q --quiet       suppress all warnings
 -r --recursive   operate recursively on directories
 -S .suf  --suffix .suf     use suffix .suf on compressed files
 -t --test        test compressed file integrity
 -v --verbose     verbose mode
 -V --version     display version number
 -1 --fast        compress faster
 -9 --best        compress better
 file...          files to (de)compress. If none given, use standard input.
```

This is the output of the command `gzip -v texinfo.tex':

```
texinfo.tex:             71.6% -- replaced with texinfo.tex.gz
```

The following command will find all `gzip` files in the current directory and subdirectories, and extract them in

place without destroying the original:

```
find . -name '*.gz' -print | sed 's/^\(.*\)[.]gz$/gunzip < "&" > "\1"/' | sh
```

Invoking gzip

The format for running the `gzip` program is:

```
gzip option ...
```

`gzip` supports the following options:

`--stdout'

`--to-stdout'

`-c'

Write output on standard output; keep original files unchanged. If there are several input files, the output consists of a sequence of independently compressed members. To obtain better compression, concatenate all input files before compressing them.

`--decompress'

`--uncompress'

`-d'

Decompress.

`--force'

`-f'

Force compression or decompression even if the file has multiple links or the corresponding file already exists, or if the compressed data is read from or written to a terminal. If the input data is not in a format recognized by `gzip`, and if the option --stdout is also given, copy the input data without change to the standard ouput: let `zcat` behave as `cat`. If `-f' is not given, and when not running in the background, `gzip` prompts to verify whether an existing file should be overwritten.

`--help'

`-h'

Print an informative help message describing the options then quit.

`--list'

`-l'

For each compressed file, list the following fields:

```
compressed size: size of the compressed file
uncompressed size: size of the uncompressed file
ratio: compression ratio (0.0% if unknown)
uncompressed_name: name of the uncompressed file
```

The uncompressed size is given as `-1' for files not in `gzip` format, such as compressed `.Z' files. To get the uncompressed size for such a file, you can use:


```
zcat file.Z | wc -c
```

In combination with the --verbose option, the following fields are also displayed:

```
method: compression method (deflate,compress,lzh,pack)
crc: the 32-bit CRC of the uncompressed data
date & time: time stamp for the uncompressed file
```

The crc is given as ffffffff for a file not in gzip format.

With --verbose, the size totals and compression ratio for all files is also displayed, unless some sizes are unknown. With --quiet, the title and totals lines are not displayed.

`--license'

`-L'

Display the `gzip` license then quit.

`--no-name'

`-n'

When compressing, do not save the original file name and time stamp by default. (The original name is always saved if the name had to be truncated.) When decompressing, do not restore the original file name if present (remove only the `gzip` suffix from the compressed file name) and do not restore the original time stamp if present (copy it from the compressed file). This option is the default when decompressing.

`--name'

`-N'

When compressing, always save the original file name and time stamp; this is the default. When decompressing, restore the original file name and time stamp if present. This option is useful on systems which have a limit on file name length or when the time stamp has been lost after a file transfer.

`--quiet'

`-q'

Suppress all warning messages.

`--recursive'

`-r'

Travel the directory structure recursively. If any of the file names specified on the command line are directories, `gzip` will descend into the directory and compress all the files it finds there (or decompress them in the case of `gunzip`).

`--suffix suf'

`-S suf'

Use suffix `suf' instead of `.gz'. Any suffix can be given, but suffixes other than `.z' and `.gz' should be avoided to avoid confusion when files are transferred to other systems. A null suffix forces gunzip to try decompression on all given files regardless of suffix, as in:

```
gunzip -S "" *        (*.* for MSDOS)
```

Previous versions of gzip used the `.z' suffix. This was changed to avoid a conflict with `pack`.

`--test'

`-t'

Test. Check the compressed file integrity.

`--verbose'

`-v'

Verbose. Display the name and percentage reduction for each file compressed.

`--version'

`-V'

Version. Display the version number and compilation options, then quit.

`--fast'

`--best'

`-n'

Regulate the speed of compression using the specified digit n, where `-1' or `--fast' indicates the fastest compression method (less compression) and `--best' or `-9' indicates the slowest compression method (optimal compression). The default compression level is `-6' (that is, biased towards high compression at expense of speed).

Advanced usage

Multiple compressed files can be concatenated. In this case, `gunzip` will extract all members at once. If one member is damaged, other members might still be recovered after removal of the damaged member. Better compression can be usually obtained if all members are decompressed and then recompressed in a single step.

This is an example of concatenating `gzip` files:

```
gzip -c file1  > foo.gz
gzip -c file2 >> foo.gz
```

Then

```
gunzip -c foo
```

is equivalent to

```
cat file1 file2
```

In case of damage to one member of a `.gz' file, other members can still be recovered (if the damaged member is removed). However, you can get better compression by compressing all members at once:

```
cat file1 file2 | gzip > foo.gz
```

compresses better than

```
gzip -c file1 file2 > foo.gz
```

If you want to recompress concatenated files to get better compression, do:

```
zcat old.gz | gzip > new.gz
```

If a compressed file consists of several members, the uncompressed size and CRC reported by the `--list' option applies to the last member only. If you need the uncompressed size for all members, you can use:

```
zcat file.gz | wc -c
```

If you wish to create a single archive file with multiple members so that members can later be extracted independently, use an archiver such as `tar` or `zip`. GNU `tar` supports the `-z' option to invoke `gzip` transparently. `gzip` is designed as a complement to `tar`, not as a replacement.

Environment

The environment variable `GZIP` can hold a set of default options for `gzip`. These options are interpreted first and can be overwritten by explicit command line parameters. For example:

```
for sh:    GZIP="-8v --name"; export GZIP
for csh:   setenv GZIP "-8v --name"
for MSDOS: set GZIP=-8v --name
```

On Vax/VMS, the name of the environment variable is `GZIP_OPT`, to avoid a conflict with the symbol set for invocation of the program.

Using `gzip` on tapes

When writing compressed data to a tape, it is generally necessary to pad the output with zeroes up to a block boundary. When the data is read and the whole block is passed to `gunzip` for decompression, `gunzip` detects that there is extra trailing garbage after the compressed data and emits a warning by default. You have to use the `--quiet' option to suppress the warning. This option can be set in the `GZIP` environment variable, as in:

```
for sh:    GZIP="-q"  tar -xfz --block-compress /dev/rst0
for csh:   (setenv GZIP "-q"; tar -xfz --block-compress /dev/rst0)
```

In the above example, `gzip` is invoked implicitly by the `-z' option of GNU `tar`. Make sure that the same block size (`-b' option of `tar`) is used for reading and writing compressed data on tapes. (This example assumes you are using the GNU version of `tar`.)

Reporting Bugs

If you find a bug in `gzip`, please send electronic mail to `jloup@chorus.fr' or, if this fails, to `bug-gnu-utils@prep.ai.mit.edu'. Include the version number, which you can find by running `gzip -V'. Also include in your message the hardware and operating system, the compiler used to compile `gzip`, a description of the bug behavior, and the input to `gzip` that triggered the bug.

Concept Index

b

c

e

i

o

s

t

Info 1.0

Info 1.0

Info

The

On-line, Menu-driven

GNU Documentation System

Published by the Free Software Foundation
675 Massachusetts Avenue,
Cambridge, MA 02139 USA

@paragraphindent 3

Getting Started

This first part of the Info manual describes how to get around inside of Info. The second part of the manual describes various advanced Info commands, and how to write an Info as distinct from a Texinfo file. The third part is about how to generate Info files from Texinfo files.

This manual is primarily designed for use on a computer, so that you can try Info commands while reading about them. Reading it on paper is less effective, since you must take it on faith that the commands described really do what the manual says. By all means go through this manual now that you have it; but please try going through the on-line version as well.

There are two ways of looking at the online version of this manual:

1. Type `info` at your shell's command line. This approach uses a stand-alone program designed just to read Info files.
2. Type `emacs` at the command line; then type C-h i (Control h, followed by i). This approach uses the Info mode of the Emacs program, an editor with many other capabilities.

In either case, then type mInfo (just the letters), followed by RET---the "Return" or "Enter" key. At this point, you should be ready to follow the instructions in this manual as you read them on the screen.

Starting Info on a Small Screen

(In Info, you only see this section if your terminal has a small number of lines; most readers pass by it without seeing it.)

Since your terminal has an unusually small number of lines on its screen, it is necessary to give you special advice at the beginning.

If you see the text `--All----' at near the bottom right corner of the screen, it means the entire text you are looking at fits on the screen. If you see `--Top----' instead, it means that there is more text below that does not fit. To move forward through the text and see another screen full, press the Space bar, SPC. To move back up, press the key labeled `Delete' or DEL.

How to use Info

You are talking to the program Info, for reading documentation.

Right now you are looking at one Node of Information. A node contains text describing a specific topic at a specific level of detail. This node's topic is "how to use Info".

The top line of a node is its header. This node's header (look at it now) says that it is the node named `Help' in the file `info'. It says that the `Next' node after this one is the node called `Help-P'. An advanced Info command lets you go to any node whose name you know.

Besides a `Next', a node can have a `Previous' or an `Up'. This node has a `Previous' but no `Up', as you can see.

Now it is time to move on to the `Next' node, named `Help-P'.

```
>> Type `n' to move there.  Type just one character;
   do not type the quotes and do not type a RET afterward.
```

`>>' in the margin means it is really time to try a command.

Returning to the Previous node

This node is called `Help-P'. The `Previous' node, as you see, is `Help', which is the one you just came from using the n command. Another n command now would take you to the next node, `Help-^L'.

```
>> But do not do that yet.  First, try the p command, which takes
   you to the `Previous' node.  When you get there, you can do an
   n again to return here.
```

This all probably seems insultingly simple so far, but *do not* be led into skimming. Things will get more complicated soon. Also, do not try a new command until you are told it is time to. Otherwise, you may make Info skip past an important warning that was coming up.

```
>> Now do an n to get to the node `Help-^L' and learn more.
```

The Space, Delete, B and ^L commands.

This node's header tells you that you are now at node `Help-^L', and that p would get you back to `Help-P'. The node's title is underlined; it says what the node is about (most nodes have titles).

This is a big node and it does not all fit on your display screen. You can tell that there is more that is not visible because you can see the string `--Top-----' rather than `--All----' near the bottom right corner of the screen.

The Space, Delete and B commands exist to allow you to "move around" in a node that does not all fit on the screen at once. Space moves forward, to show what was below the bottom of the screen. Delete moves backward, to show what was above the top of the screen (there is not anything above the top until you have typed some spaces).

```
>> Now try typing a Space (afterward, type a Delete to return here).
```

When you type the space, the two lines that were at the bottom of the screen appear at the top, followed by more lines. Delete takes the two lines from the top and moves them to the bottom, *usually*, but if there are not a full screen's worth of lines above them they may not make it all the way to the bottom.

If you type Space when there is no more to see, it rings the bell and otherwise does nothing. The same goes for Delete when the header of the node is visible.

If your screen is ever garbaged, you can tell Info to print it out again by typing C-l (Control-L, that is--hold down "Control" and type an L or l).

```
>> Type C-l now.
```

To move back to the beginning of the node you are on, you can type a lot of Deletes. You can also type simply b for beginning.

```
>> Try that now.  (We have put in enough verbiage to push this past
the first screenful, but screens are so big nowadays that perhaps it
isn't enough.  You may need to shrink your Emacs or Info window.)
Then come back, with Spaces.
```

If your screen is very tall, all of this node might fit at once. In that case, "b" won't do anything. Sorry; what can we do?

You have just learned a considerable number of commands. If you want to use one but have trouble remembering which, you should type a ? which prints out a brief list of commands. When you are finished looking at the list, make it go away by typing a SPC.

```
>> Type a ? now.  After it finishes, type a SPC.
```

(If you are using the standalone Info reader, type `l' to return here.)

From now on, you will encounter large nodes without warning, and will be expected to know how to use Space and Delete to move around in them without being told. Since not all terminals have the same size

screen, it would be impossible to warn you anyway.

```
>> Now type n to see the description of the m command.
```

Menus

Menus and the m command

With only the n and p commands for moving between nodes, nodes are restricted to a linear sequence. Menus allow a branching structure. A menu is a list of other nodes you can move to. It is actually just part of the text of the node formatted specially so that Info can interpret it. The beginning of a menu is always identified by a line which starts with `* Menu:'. A node contains a menu if and only if it has a line in it which starts that way. The only menu you can use at any moment is the one in the node you are in. To use a menu in any other node, you must move to that node first.

After the start of the menu, each line that starts with a `*' identifies one subtopic. The line usually contains a brief name for the subtopic (followed by a `:'), the name of the node that talks about that subtopic, and optionally some further description of the subtopic. Lines in the menu that do not start with a `*' have no special meaning--they are only for the human reader's benefit and do not define additional subtopics. Here is an example:

```
* Foo:  FOO's Node      This tells about FOO
```

The subtopic name is Foo, and the node describing it is `FOO's Node'. The rest of the line is just for the reader's Information. [[ But this line is not a real menu item, simply because there is no line above it which starts with `* Menu:'.]]

When you use a menu to go to another node (in a way that will be described soon), what you specify is the subtopic name, the first thing in the menu line. Info uses it to find the menu line, extracts the node name from it, and goes to that node. The reason that there is both a subtopic name and a node name is that the node name must be meaningful to the computer and may therefore have to be ugly looking. The subtopic name can be chosen just to be convenient for the user to specify. Often the node name is convenient for the user to specify and so both it and the subtopic name are the same. There is an abbreviation for this:

```
* Foo::   This tells about FOO
```

This means that the subtopic name and node name are the same; they are both `Foo'.

```
>> Now use Spaces to find the menu in this node, then come back to
   the front with a b and some Spaces.  As you see, a menu is
   actually visible in its node.  If you cannot find a menu in a node
   by looking at it, then the node does not have a menu and the
   m command is not available.
```

The command to go to one of the subnodes is m---but *do not do it yet!* Before you use m, you must understand the difference between commands and arguments. So far, you have learned several commands that do not need arguments. When you type one, Info processes it and is instantly ready for another command. The m command is different: it is incomplete without the name of the subtopic. Once you have typed m, Info tries to

read the subtopic name.

Now look for the line containing many dashes near the bottom of the screen. There is one more line beneath that one, but usually it is blank. If it is empty, Info is ready for a command, such as n or b or Space or m. If that line contains text ending in a colon, it mean Info is trying to read the argument to a command. At such times, commands do not work, because Info tries to use them as the argument. You must either type the argument and finish the command you started, or type Control-g to cancel the command. When you have done one of those things, the line becomes blank again.

The command to go to a subnode via a menu is m. After you type the m, the line at the bottom of the screen says `Menu item: '. You must then type the name of the subtopic you want, and end it with a RET.

You can abbreviate the subtopic name. If the abbreviation is not unique, the first matching subtopic is chosen. Some menus put the shortest possible abbreviation for each subtopic name in capital letters, so you can see how much you need to type. It does not matter whether you use upper case or lower case when you type the subtopic. You should not put any spaces at the end, or inside of the item name, except for one space where a space appears in the item in the menu.

You can also use the completion feature to help enter the subtopic name. If you type the Tab key after entering part of a name, it will magically fill in more of the name--as much as follows uniquely from what you have entered.

If you move the cursor to one of the menu subtopic lines, then you do not need to type the argument: you just type a Return, and it stands for the subtopic of the line you are on.

Here is a menu to give you a chance to practice.

* Menu: The menu starts here.

This menu gives you three ways of going to one place, Help-FOO.

* Foo: Help-FOO. A node you can visit for fun.
* Bar: Help-FOO. Strange! two ways to get to the same place.
* Help-FOO:: And yet another!

```
>>  Now type just an m and see what happens:
```

Now you are "inside" an m command. Commands cannot be used now; the next thing you will type must be the name of a subtopic.

You can change your mind about doing the m by typing Control-g.

```
>> Try that now;  notice the bottom line clear.

>> Then type another m.

>> Now type `BAR' item name.  Do not type Return yet.
```

While you are typing the item name, you can use the Delete key to cancel one character at a time if you make a mistake.

```
>> Type one to cancel the `R'.  You could type another `R' to
```

replace it. You do not have to, since `BA' is a valid abbreviation.

>> Now you are ready to go. Type a RET.

After visiting Help-FOO, you should return here.

>> Type n to see more commands.

Here is another way to get to Help-FOO, a menu. You can ignore this if you want, or else try it (but then please come back to here).

The u command

Congratulations! This is the node `Help-FOO'. Unlike the other nodes you have seen, this one has an `Up': `Help-M', the node you just came from via the m command. This is the usual convention--the nodes you reach from a menu have `Up' nodes that lead back to the menu. Menus move Down in the tree, and `Up' moves Up. `Previous', on the other hand, is usually used to "stay on the same level but go backwards"

You can go back to the node `Help-M' by typing the command u for "Up". That puts you at the *front* of the node--to get back to where you were reading you have to type some SPCs.

>> Now type u to move back up to `Help-M'.

Some advanced Info commands

The course is almost over, so please stick with it to the end.

If you have been moving around to different nodes and wish to retrace your steps, the l command (l for last) will do that, one node-step at a time. As you move from node to node, Info records the nodes where you have been in a special history list. The l command revisits nodes in the history list; each successive l command moves one step back through the history.

If you have been following directions, ad l command now will get you back to `Help-M'. Another l command would undo the u and get you back to `Help-FOO'. Another l would undo the m and get you back to `Help-M'.

>> Try typing three l's, pausing in between to see what each
    l does.

Then follow directions again and you will end up back here.

Note the difference between l and p: l moves to where *you* last were, whereas p always moves to the node which the header says is the `Previous' node (from this node, to `Help-M').

The `d' command gets you instantly to the Directory node. This node, which is the first one you saw when you entered Info, has a menu which leads (directly, or indirectly through other menus), to all the nodes that exist.

>> Try doing a `d', then do an l to return here (yes,
   do return).

Sometimes, in Info documentation, you will see a cross reference. Cross references look like this: See sec The node reached by the cross reference in Info. That is a real, live cross reference which is named `Cross points at the node named `Help-Cross'.

If you wish to follow a cross reference, you must use the `f' command. The `f' must be followed by the cross reference name (in this case, `Cross'). While you enter the name, you can use the Delete key to edit your input. If you change your mind about following any reference, you can use Control-g to cancel the command.

Completion is available in the `f' command; you can complete among all the cross reference names in the current node by typing a Tab.

```
>> Type `f', followed by `Cross', and a RET.
```

To get a list of all the cross references in the current node, you can type ? after an `f'. The `f' continues to await a cross reference name even after printing the list, so if you don't actually want to follow a reference, you should type a Control-g to cancel the `f'.

```
>> Type "f?" to get a list of the cross references in this node.  Then
   type a Control-g and see how the `f' gives up.

>> Now type n to see the last node of the course.
```

The node reached by the cross reference in Info

This is the node reached by the cross reference named `Cross'.

While this node is specifically intended to be reached by a cross reference, most cross references lead to nodes that "belong" someplace else far away in the structure of Info. So you cannot expect the footnote to have a `Next', `Previous' or `Up' pointing back to where you came from. In general, the l (el) command is the only way to get back there.

```
>> Type l to return to the node where the cross reference was.
```

Quitting Info

To get out of Info, back to what you were doing before, type q for Quit.

This is the end of the course on using Info. There are some other commands that are meant for experienced users; they are useful, and you can find them by looking in the directory node for documentation on Info. Finding them will be a good exercise in using Info in the usual manner.

```
>> Type `d' to go to the Info directory node; then type
   `mInfo' and Return, to get to the node about Info and
   see what other help is available.
```

Info for Experts

This chapter describes various advanced Info commands, and how to write an Info as distinct from a Texinfo file. (However, in most cases, writing a Texinfo file is better, since you can use it *both* to generate an Info file and to make a printed manual. See section 'Overview of Texinfo' in Texinfo: The GNU Documentation Format.)

Advanced Info Commands

g, s, 1, -- 9, and e

If you know a node's name, you can go there by typing g, the name, and RET. Thus, gTopRET would go to the node called `Top' in this file (its directory node). gExpertRET would come back here.

Unlike m, g does not allow the use of abbreviations.

To go to a node in another file, you can include the filename in the node name by putting it at the front, in parentheses. Thus, g(dir)TopRET would go to the Info Directory node, which is node `Top' in the file `dir'.

The node name `*' specifies the whole file. So you can look at all of the current file by typing g*RET or all of any other file with g(FILENAME)RET.

The s command allows you to search a whole file for a string. It switches to the next node if and when that is necessary. You type s followed by the string to search for, terminated by RET. To search for the same string again, just s followed by RET will do. The file's nodes are scanned in the order they are in in the file, which has no necessary relationship to the order that they may be in in the tree structure of menus and `next' pointers. But normally the two orders are not very different. In any case, you can always do a b to find out what node you have reached, if the header is not visible (this can happen, because s puts your cursor at the occurrence of the string, not at the beginning of the node).

If you grudge the system each character of type-in it requires, you might like to use the commands 1, 2, 3, 4, ... 9. They are short for the m command together with an argument. 1 goes through the first item in the current node's menu; 2 goes through the second item, etc.

If you display supports multiple fonts, and you are using Emacs' Info mode to read Info files, the `*' for the fifth menu item is underlines, and so is the `*' for the ninth item; these underlines make it easy to see at a glance which number to use for an item.

On ordinary terminals, you won't have underlining. If you need to actually count items, it is better to use m instead, and specify the name.

The Info command e changes from Info mode to an ordinary Emacs editing mode, so that you can edit the text of the current node. Type C-c C-c to switch back to Info. The e command is allowed only if the variable `Info-enable-edit` is non-`nil`.

Adding a new node to Info

To add a new topic to the list in the Info directory, you must:

1. Create some nodes, in some file, to document that topic.
2. Put that topic in the menu in the directory. See section How to Create Menus.

Usually, the way to create the nodes is with Texinfo see section 'Overview of Texinfo' in Texinfo: The GNU Documentation Format); this has the advantage that you can also make a printed manual from them. However, if hyou want to edit an Info file, here is how.

The new node can live in an existing documentation file, or in a new one. It must have a ^_ character before it (invisible to the user; this node has one but you cannot see it), and it ends with either a ^_, a ^L, or the end of file. Note: If you put in a ^L to end a new node, be sure that there is a ^_ after it to start the next one, since ^L cannot *start* a node. Also, a nicer way to make a node boundary be a page boundary as well is to put a ^L *right after* the ^_.

The ^_ starting a node must be followed by a newline or a ^L newline, after which comes the node's header line. The header line must give the node's name (by which Info finds it), and state the names of the `Next', `Previous', and `Up' nodes (if there are any). As you can see, this node's `Up' node is the node `Top', which points at all the documentation for Info. The `Next' node is `Menus'.

The keywords Node, Previous, Up, and Next, may appear in any order, anywhere in the header line, but the recommended order is the one in this sentence. Each keyword must be followed by a colon, spaces and tabs, and then the appropriate name. The name may be terminated with a tab, a comma, or a newline. A space does not end it; node names may contain spaces. The case of letters in the names is insignificant.

A node name has two forms. A node in the current file is named by what appears after the `Node: ' in that node's first line. For example, this node's name is `Add'. A node in another file is named by `(filename)node-within-file', as in `(info)Add' for this node. If the file name starts with "./", then it is relative to the current directory; otherwise, it is relative starting from the standard Info file directory of your site. The name `(filename)Top' can be abbreviated to just `(filename)'. By convention, the name `Top' is used for the "highest" node in any single file--the node whose `Up' points out of the file. The Directory node is `(dir)'. The `Top' node of a document file listed in the Directory should have an `Up: (dir)' in it.

The node name * is special: it refers to the entire file. Thus, g* shows you the whole current file. The use of the node * is to make it possible to make old-fashioned, unstructured files into nodes of the tree.

The `Node:' name, in which a node states its own name, must not contain a filename, since Info when searching for a node does not expect one to be there. The `Next', `Previous' and `Up' names may contain them. In this node, since the `Up' node is in the same file, it was not necessary to use one.

Note that the nodes in this file have a file name in the header line. The file names are ignored by Info, but they serve as comments to help identify the node for the user.

How to Create Menus

Any node in the Info hierarchy may have a menu---a list of subnodes. The m command searches the current node's menu for the topic which it reads from the terminal.

A menu begins with a line starting with `* Menu:'. The rest of the line is a comment. After the starting lin every line that begins with a `* ' lists a single topic. The name of the topic--the argument that the user mu give to the m command to select this topic--comes right after the star and space, and is followed by a colon, spaces and tabs, and the name of the node which discusses that topic. The node name, like node names following `Next', `Previous' and `Up', may be terminated with a tab, comma, or newline; it may also be terminated with a period.

If the node name and topic name are the same, then rather than giving the name twice, the abbreviation `* NAME::' may be used (and should be used, whenever possible, as it reduces the visual clutter in the menu).

It is considerate to choose the topic names so that they differ from each other very near the beginning--this allows the user to type short abbreviations. In a long menu, it is a good idea to capitalize the beginning of each item name which is the minimum acceptable abbreviation for it (a long menu is more than 5 or so entries).

The nodes listed in a node's menu are called its "subnodes", and it is their "superior". They should each have an `Up:' pointing at the superior. It is often useful to arrange all or most of the subnodes in a sequence of `Next' and `Previous' pointers so that someone who wants to see them all need not keep revisiting the Menu.

The Info Directory is simply the menu of the node `(dir)Top'---that is, node `Top' in file ``../info/dir'`. You can put new entries in that menu just like any other menu. The Info Directory is *not* the same as the file directory called `` `info' ``. It happens that many of Info's files live on that file directory, but they do not have to; and files on that directory are not automatically listed in the Info Directory node.

Also, although the Info node graph is claimed to be a "hierarchy", in fact it can be *any* directed graph. Shared structures and pointer cycles are perfectly possible, and can be used if they are appropriate to the meaning to be expressed. There is no need for all the nodes in a file to form a connected structure. In fact, this file has two connected components. You are in one of them, which is under the node `Top'; the other contains the node `Help' which the h command goes to. In fact, since there is no garbage collector, nothing terrible happens if a substructure is not pointed to, but such a substructure is rather useless since nobody can ever find out that it exists.

Creating Cross References

A cross reference can be placed anywhere in the text, unlike a menu item which must go at the front of a line. A cross reference looks like a menu item except that it has `*note' instead of *. It *cannot* be terminated by a `)', because `)"s are so often part of node names. If you wish to enclose a cross reference in parentheses, terminate it with a period first. Here are two examples of cross references pointers:

```
*Note details: commands.  (See *note 3: Full Proof.)
```

They are just examples. The places they "lead to" do not really exist!

Tag Tables for Info Files

You can speed up the access to nodes of a large Info file by giving it a tag table. Unlike the tag table for a program, the tag table for an Info file lives inside the file itself and is used automatically whenever Info reads in the file.

To make a tag table, go to a node in the file using Emacs Info mode and type M-x Info-tagify. Then you must

use C-x C-s to save the file.

Once the Info file has a tag table, you must make certain it is up to date. If, as a result of deletion of text, any node moves back more than a thousand characters in the file from the position recorded in the tag table, Info will no longer be able to find that node. To update the tag table, use the `Info-tagify` command again.

An Info file tag table appears at the end of the file and looks like this:

```
^_
Tag Table:
File: info, Node: Cross-refs^?21419
File: info,  Node: Tags^?22145
^_
End Tag Table
```

Note that it contains one line per node, and this line contains the beginning of the node's header (ending just after the node name), a Delete character, and the character position in the file of the beginning of the node.

Checking an Info File

When creating an Info file, it is easy to forget the name of a node when you are making a pointer to it from another node. If you put in the wrong name for a node, this is not detected until someone tries to go through the pointer using Info. Verification of the Info file is an automatic process which checks all pointers to nodes and reports any pointers which are invalid. Every `Next', `Previous', and `Up' is checked, as is every menu item and every cross reference. In addition, any `Next' which does not have a `Previous' pointing back is reported. Only pointers within the file are checked, because checking pointers to other files would be terribly slow. But those are usually few.

To check an Info file, do M-x Info-validate while looking at any node of the file with Emacs Info mode.

Emacs Info-mode Variables

The following variables may modify the behaviour of Info-mode in Emacs; you may wish to set one or several of these variables interactively, or in your `` `~/.emacs' `` init file. See section 'Examining and Setting Variables' in The GNU Emacs Manual.

`Info-enable-edit`

Set to `nil`, disables the `e' (`Info-edit`) command. A non-`nil` value enables it. See section Adding a new node to Info.

`Info-enable-active-nodes`

When set to a non-`nil` value, allows Info to execute Lisp code associated with nodes. The Lisp code is executed when the node is selected.

`Info-directory-list`

The list of directories to search for Info files. Each element is a string (directory name) or `nil` (try default directory).

`Info-directory`

The standard directory for Info documentation files. Only used when the function `Info-direct` is called.

Creating an Info File from a Makeinfo file

`makeinfo` is a utility that converts a Texinfo file into an Info file; `texinfo-format-region` and `texinfo-format-buffer` are GNU Emacs functions that do the same.

See section 'Creating an Info File' in the Texinfo Manual, to learn how to create an Info file from a Texinfo file.

See section 'Overview of Texinfo' in Texinfo: The GNU Documentation Format, to learn how to write a Texinfo file.

@nwnode Using Stand-alone Info, Options, , Top

Using the Stand-alone Info Reader

@lowersections @paragraphindent 2

What is Info?

This text documents the use of the GNU Info program, version 2.10.

Info is a program which is used to view info files on an ASCII terminal. info files are the result of processing texinfo files with the program `makeinfo` or with the Emacs command `M-x texinfo-format-buffer`. Finally, texinfo is a documentation language which allows a printed manual and online documentation (an info file) to be produced from a single source file.

Command Line Options

GNU Info accepts several options to control the initial node being viewed, and to specify which directories to search for info files. Here is a template showing an invocation of GNU Info from the shell:

```
info [--option-name option-value] menu-item...
```

The following option-names are available when invoking Info from the shell:

`--directory directory-path`

`-d directory-path`

Adds directory-path to the list of directory paths searched when Info needs to find a file. You may issue `--directory` multiple times; once for each directory which contains info files. Alternatively, you may specify a value for the environment variable `INFOPATH`; if `--directory` is not given, the value of `INFOPATH` is used. The value of `INFOPATH` is a colon separated list of directory names. If you do not supply `INFOPATH` or `--directory-path` a default path is used.

`--file filename`

`-f filename`

Specifies a particular info file to visit. Instead of visiting the file `dir`, Info will start with `(filename)Top` as the first file and node.

`--node nodename`

`-n nodename`

Specifies a particular node to visit in the initial file loaded. This is especially useful in conjunction with `--file`(1). You may specify `--node` multiple times; for an interactive Info, each nodename is visited in its own window, for a non-interactive Info (such as when `--output` is given) each nodename is processed sequentially.

`--output filename`

`-o filename`

Specify filename as the name of a file to output to. Each node that Info visits will be output to filename instead of interactively viewed. A value of `-` for filename specifies the standard output.

`--subnodes`

This option only has meaning when given in conjunction with `--output`. It means to recursively output the nodes appearing in the menus of each node being output. Menu items which resolve to external info files are not output, and neither are menu items which are members of an index. Each node is only output once.

`--help`

`-h`

Produces a relatively brief description of the available Info options.

`--version`

Prints the version information of Info and exits.

`menu-item`

Remaining arguments to Info are treated as the names of menu items. The first argument would be a menu item in the initial node visited, while the second argument would be a menu item in the first argument's node. You can easily move to the node of your choice by specifying the menu names which describe the path to that node. For example,

```
info emacs buffers
```

first selects the menu item `Emacs' in the node `(dir)Top', and then selects the menu item `Buffers' in the node `(emacs)Top'.

Moving the Cursor

Many people find that reading screens of text page by page is made easier when one is able to indicate particular pieces of text with some kind of pointing device. Since this is the case, GNU Info (both the Emacs and standalone versions) have several commands which allow you to move the cursor about the screen. The notation used in this manual to describe keystrokes is identical to the notation used within the Emacs manual, and the GNU Readline manual. See section 'Character Conventions' in the GNU Emacs Manual, if you are unfamilar with the notation.

The following table lists the basic cursor movement commands in Info. Each entry consists of the key sequence you should type to execute the cursor movement, the `M-x`(2) command name (displayed in parentheses), and a short description of what the command does. All of the cursor motion commands can take an numeric argument (see section Miscellaneous Commands}), to find out how to supply them. With a numeric argument, the motion commands are simply executed that many times; for example, a numeric argument of 4 given to `next-line` causes the cursor to move down 4 lines. With a negative numeric argument, the motion is reversed; an argument of -4 given to the `next-line` command would cause the cursor to move *up* 4 lines.

`C-n` (`next-line`)

Moves the cursor down to the next line.

`C-p` (`prev-line`)

Move the cursor up to the previous line.

`C-a` (`beginning-of-line`)

Move the cursor to the start of the current line.

`C-e` (`end-of-line`)

Moves the cursor to the end of the current line.

`C-f` (`forward-char`)

Move the cursor forward a character.

`C-b` (`backward-char`)

Move the cursor backward a character.

`M-f` (`forward-word`)

Moves the cursor forward a word.

`M-b` (`backward-word`)

Moves the cursor backward a word.

`M-<` (`beginning-of-node`)

`b`

Moves the cursor to the start of the current node.

`M->` (`end-of-node`)

Moves the cursor to the end of the current node.

`M-r` (`move-to-window-line`)

Moves the cursor to a specific line of the window. Without a numeric argument, `M-r` moves the cursor to the start of the line in the center of the window. With a numeric argument of n, `M-r` moves the cursor

to the start of the nth line in the window.

Moving Text Within a Window

Sometimes you are looking at a screenful of text, and only part of the current paragraph you are reading is visible on the screen. The commands detailed in this section are used to shift which part of the current node is visible on the screen.

`SPC` (`scroll-forward`)

`C-v`

Shift the text in this window up. That is, show more of the node which is currently below the bottom of the window. With a numeric argument, show that many more lines at the bottom of the window; a numeric argument of 4 would shift all of the text in the window up 4 lines (discarding the top 4 lines), and show you four new lines at the bottom of the window. Without a numeric argument, SPC takes the bottom two lines of the window and places them at the top of the window, redisplaying almost a completely new screenful of lines.

`DEL` (`scroll-backward`)

`M-v`

Shift the text in this window down. The inverse of `scroll-forward`.

The `scroll-forward` and `scroll-backward` commands can also move forward and backward through the node structure of the file. If you press SPC while viewing the end of a node, or DEL while viewing the beginning of a node, what happens is controlled by the variable `scroll-behaviour`. See section Manipulating Variables}, for more information.

`C-l` (`redraw-display`)

Redraw the display from scratch, or shift the line containing the cursor to a specified location. With no numeric argument, `C-l' clears the screen, and then redraws its entire contents. Given a numeric argument of n, the line containing the cursor is shifted so that it is on the nth line of the window.

`C-x w` (`toggle-wrap`)

Toggles the state of line wrapping in the current window. Normally, lines which are longer than the screen width wrap, i.e., they are continued on the next line. Lines which wrap have a `\' appearing in the rightmost column of the screen. You can cause such lines to be terminated at the rightmost column by changing the state of line wrapping in the window with `C-x w`. When a line which needs more space than one screen width to display is displayed, a `$' appears in the rightmost column of the screen, and the remainder of the line is invisible.

Selecting a New Node

This section details the numerous Info commands which select a new node to view in the current window.

The most basic node commands are `n', `p', `u', and `l'.

When you are viewing a node, the top line of the node contains some Info pointers which describe where the next, previous, and up nodes are. Info uses this line to move about the node structure of the file when you use the following commands:

`n` (`next-node`)

Selects the `Next' node.

`p` (`prev-node`)

Selects the `Prev' node.

`u` (`up-node`)

Selects the `Up' node.

You can easily select a node that you have already viewed in this window by using the `l' command -- this name stands for "last", and actually moves through the list of already visited nodes for this window. `l' with a negative numeric argument moves forward through the history of nodes for this window, so you can quickly step between two adjacent (in viewing history) nodes.

`l` (`history-node`)

Selects the most recently selected node in this window.

Two additional commands make it easy to select the most commonly selected nodes; they are `t' and `d'.

`t` (`top-node`)

Selects the node `Top' in the current info file.

`d` (`dir-node`)

Selects the directory node (i.e., the node `(dir)').

Here are some other commands which immediately result in the selection of a different node in the current window:

`<` (`first-node`)

Selects the first node which appears in this file. This node is most often `Top', but it doesn't have to be.

`>` (`last-node`)

Selects the last node which appears in this file.

`]` (`global-next-node`)

Moves forward or down through node structure. If the node that you are currently viewing has a `Next' pointer, that node is selected. Otherwise, if this node has a menu, the first menu item is selected. If there is no `Next' and no menu, the same process is tried with the `Up' node of this node.

`[` (`global-prev-node`)

Moves backward or up through node structure. If the node that you are currently viewing has a `Prev' pointer, that node is selected. Otherwise, if the node has an `Up' pointer, that node is selected, and if it has a menu, the last item in the menu is selected.

TOC

You can get the same behaviour as `global-next-node` and `global-prev-node` while simply scrolling through the file with SPC and DEL; See section Manipulating Variables, for more information.

`g` (`goto-node`)

Reads the name of a node and selects it. No completion is done while reading the node name, since the desired node may reside in a separate file. The node must be typed exactly as it appears in the info file. A file name may be included as with any node specification, for example

```
g(emacs)Buffers
```

finds the node `Buffers' in the info file `` `emacs' ``.

`C-x k` (`kill-node`)

Kills a node. The node name is prompted for in the echo area, with a default of the current node. Killing a node means that Info tries hard to forget about it, removing it from the list of history nodes kept for the window where that node is found. Another node is selected in the window which contained the killed node.

`C-x C-f` (`view-file`)

Reads the name of a file and selects the entire file. The command

```
C-x C-f filename
```

is equivalent to typing

```
g(filename)*
```

`C-x C-b` (`list-visited-nodes`)

Makes a window containing a menu of all of the currently visited nodes. This window becomes the selected window, and you may use the standard Info commands within it.

`C-x b` (`select-visited-node`)

Selects a node which has been previously visited in a visible window. This is similar to `C-x C-b' followed by `m', but no window is created.

Searching an Info File

GNU Info allows you to search for a sequence of characters throughout an entire info file, search through the indices of an info file, or find areas within an info file which discuss a particular topic.

`s` (`search`)

Reads a string in the echo area and searches for it.

`C-s` (`isearch-forward`)

Interactively searches forward through the info file for a string as you type it.

`C-r` (`isearch-backward`)

Interactively searches backward through the info file for a string as you type it.

`i` (`index-search`)

Looks up a string in the indices for this info file, and selects a node where the found index entry points

to.

`,` (`next-index-match`)
Moves to the node containing the next matching index item from the last `i' command.

The most basic searching command is `s' (`search`). The `s' command prompts you for a string in the echo area, and then searches the remainder of the info file for an ocurrence of that string. If the string is found, the node containing it is selected, and the cursor is left positioned at the start of the found string. Subsequent `s' commands show you the default search string within `[' and `]'; pressing RET instead of typing a new string will use the default search string.

Incremental searching is similar to basic searching, but the string is looked up while you are typing it, instead of waiting until the entire search string has been specified.

Selecting Cross References

We have already discussed the `Next', `Prev', and `Up' pointers which appear at the top of a node. In addition to these pointers, a node may contain other pointers which refer you to a different node, perhaps in another info file. Such pointers are called cross references, or xrefs for short.

Parts of an Xref

Cross references have two major parts: the first part is called the label; it is the name that you can use to refer to the cross reference, and the second is the target; it is the full name of the node that the cross reference points to.

The target is separated from the label by a colon `:'; first the label appears, and then the target. For example, in the sample menu cross reference below, the single colon separates the label from the target.

```
* Foo Label: Foo Target.        More information about Foo.
```

Note the `.' which ends the name of the target. The `.' is not part of the target; it serves only to let Info know where the target name ends.

A shorthand way of specifying references allows two adjacent colons to stand for a target name which is the same as the label name:

```
* Foo Commands::                Commands pertaining to Foo.
```

In the above example, the name of the target is the same as the name of the label, in this case `Foo Commands`.

You will normally see two types of cross references while viewing nodes: menu references, and note references. Menu references appear within a node's menu; they begin with a `*' at the beginning of a line, and continue with a label, a target, and a comment which describes what the contents of the node pointed to contains.

Note references appear within the body of the node text; they begin with `*Note`, and continue with a label and a target.

Like `Next', `Prev' and `Up' pointers, cross references can point to any valid node. They are used to refer you to a place where more detailed information can be found on a particular subject. Here is a cross reference which points to a node within the Texinfo documentation: See section 'Writing an Xref' in the Texinfo Manual, for more information on creating your own texinfo cross references.

Selecting Xrefs

The following table lists the Info commands which operate on menu items.

`1` (`menu-digit`)

`2` ... `9`

Within an Info window, pressing a single digit, (such as `1'), selects that menu item, and places its node in the current window. For convenience, there is one exception; pressing `0' selects the *last* item in the node's menu.

`0` (`last-menu-item`)

Select the last item in the current node's menu.

`m` (`menu-item`)

Reads the name of a menu item in the echo area and selects its node. Completion is available while reading the menu label.

`M-x find-menu`

Moves the cursor to the start of this node's menu.

This table lists the Info commands which operate on note cross references.

`f` (`xref-item`)

`r`

Reads the name of a note cross reference in the echo area and selects its node. Completion is available while reading the cross reference label.

Finally, the next few commands operate on menu or note references alike:

`TAB` (`move-to-next-xref`)

Moves the cursor to the start of the next nearest menu item or note reference in this node. You can then use RET (`select-reference-this-line` to select the menu or note reference.

`M-TAB` (`move-to-prev-xref`)

Moves the cursor the start of the nearest previous menu item or note reference in this node.

`RET` (`select-reference-this-line`)

Selects the menu item or note reference appearing on this line.

Manipulating Multiple Windows

A window is a place to show the text of a node. Windows have a view area where the text of the node is displayed, and an associated mode line, which briefly describes the node being viewed.

GNU Info supports multiple windows appearing in a single screen; each window is separated from the next by its modeline. At any time, there is only one active window, that is, the window in which the cursor appears. There are commands available for creating windows, changing the size of windows, selecting which window is active, and for deleting windows.

The Mode Line

A mode line is a line of inverse video which appears at the bottom of an info window. It describes the contents of the window just above it; this information includes the name of the file and node appearing in that window, the number of screen lines it takes to display the node, and the percentage of text that is above the top of the window. It can also tell you if the indirect tags table for this info file needs to be updated, and whether or not the info file was compressed when stored on disk.

Here is a sample mode line for a window containing an uncompressed file named `dir', showing the node `Top'.

```
-----Info: (dir)Top, 40 lines --Top-----------------------------------
           ^^   ^   ^^^         ^^
         (file)Node #lines    where
```

When a node comes from a file which is compressed on disk, this is indicated in the mode line with two small `z"s. In addition, if the info file containing the node has been split into subfiles, the name of the subfile containing the node appears in the modeline as well:

```
--zz-Info: (emacs)Top, 291 lines --Top-- Subfile: emacs-1.Z--------------
```

When Info makes a node internally, such that there is no corresponding info file on disk, the name of the node is surrounded by asterisks (`*'). The name itself tells you what the contents of the window are; the sample mode line below shows an internally constructed node showing possible completions:

```
-----Info: *Completions*, 7 lines --All-------------------------------
```

Window Commands

It can be convenient to view more than one node at a time. To allow this, Info can display more than one window. Each window has its own mode line (see section The Mode Line) and history of nodes viewed in that window (see section Selecting a New Node).

`C-x o` (`next-window`)

Selects the next window on the screen. Note that the echo area can only be selected if it is already in use, and you have left it temporarily. Normally, `C-x o' simply moves the cursor into the next window

TOC

on the screen, or if you are already within the last window, into the first window on the screen. Giv numeric argument, `C-x o' moves over that many windows. A negative argument causes `C-x o' to select the previous window on the screen.

`M-x prev-window`

Selects the previous window on the screen. This is identical to `C-x o' with a negative argument.

`C-x 2` (`split-window`)

Splits the current window into two windows, both showing the same node. Each window is one half the size of the original window, and the cursor remains in the original window. The variable `automatic-tiling` can cause all of the windows on the screen to be resized for you automatically, please see section Manipulating Variables for more information.

`C-x 0` (`delete-window`)

Deletes the current window from the screen. If you have made too many windows and your screen appears cluttered, this is the way to get rid of some of them.

`C-x 1` (`keep-one-window`)

Deletes all of the windows excepting the current one.

`ESC C-v` (`scroll-other-window`)

Scrolls the other window, in the same fashion that `C-v' might scroll the current window. Given a negative argument, the "other" window is scrolled backward.

`C-x ^` (`grow-window`)

Grows (or shrinks) the current window. Given a numeric argument, grows the current window that many lines; with a negative numeric argument, the window is shrunk instead.

`C-x t` (`tile-windows`)

Divides the available screen space among all of the visible windows. Each window is given an equal portion of the screen in which to display its contents. The variable `automatic-tiling` can cause `tile-windows` to be called when a window is created or deleted. See section Manipulating Variables.

The Echo Area

The echo area is a one line window which appears at the bottom of the screen. It is used to display informative or error messages, and to read lines of input from you when that is necessary. Almost all of the commands available in the echo area are identical to their Emacs counterparts, so please refer to that documentation for greater depth of discussion on the concepts of editing a line of text. The following table briefly lists the commands that are available while input is being read in the echo area:

`C-f` (`echo-area-forward`)

Moves forward a character.

`C-b` (`echo-area-backward`)

Moves backward a character.

`C-a` (`echo-area-beg-of-line`)

Moves to the start of the input line.

`C-e` (`echo-area-end-of-line`)

Moves to the end of the input line.

`M-f` (`echo-area-forward-word`)

Moves forward a word.

`M-b` (`echo-area-backward-word`)

Moves backward a word.

`C-d` (`echo-area-delete`)

Deletes the character under the cursor.

`DEL` (`echo-area-rubout`)

Deletes the character behind the cursor.

`C-g` (`echo-area-abort`)

Cancels or quits the current operation. If completion is being read, `C-g' discards the text of the input line which does not match any completion. If the input line is empty, `C-g' aborts the calling function.

`RET` (`echo-area-newline`)

Accepts (or forces completion of) the current input line.

`C-q` (`echo-area-quoted-insert`)

Inserts the next character verbatim. This is how you can insert control characters into a search string, for example.

printing character (`echo-area-insert`)

Inserts the character.

`M-TAB` (`echo-area-tab-insert`)

Inserts a TAB character.

`C-t` (`echo-area-transpose-chars`)

Transposes the characters at the cursor.

The next group of commands deal with killing, and yanking text. For an in depth discussion of killing and yanking, see section 'Killing and Deleting' in the GNU Emacs Manual

`M-d` (`echo-area-kill-word`)

Kills the word following the cursor.

`M-DEL` (`echo-area-backward-kill-word`)

Kills the word preceding the cursor.

`C-k` (`echo-area-kill-line`)

Kills the text from the cursor to the end of the line.

`C-x DEL` (`echo-area-backward-kill-line`)

Kills the text from the cursor to the beginning of the line.

`C-y` (`echo-area-yank`)

Yanks back the contents of the last kill.

`M-y` (`echo-area-yank-pop`)

Yanks back a previous kill, removing the last yanked text first.

Sometimes when reading input in the echo area, the command that needed input will only accept one of a list

of several choices. The choices represent the possible completions, and you must respond with one of them. Since there are a limited number of responses you can make, Info allows you to abbreviate what you type, only typing as much of the response as is necessary to uniquely identify it. In addition, you can request Info to fill in as much of the response as is possible; this is called completion.

The following commands are available when completing in the echo area:

`TAB` (`echo-area-complete`)

`SPC`

Inserts as much of a completion as is possible.

`?` (`echo-area-possible-completions`)

Displays a window containing a list of the possible completions of what you have typed so far. For example, if the available choices are:

```
bar
foliate
food
forget
```

and you have typed an `f', followed by `?', the possible completions would contain:

```
foliate
food
forget
```

i.e., all of the choices which begin with `f'. Pressing SPC or TAB would result in `fo' appearing in the echo area, since all of the choices which begin with `f' continue with `o'. Now, typing `l' followed by `TAB' results in `foliate' appearing in the echo area, since that is the only choice which begins with `fol'.

`ESC C-v` (`echo-area-scroll-completions-window`)

Scrolls the completions window, if that is visible, or the "other" window if not.

Printing Out Nodes

You may wish to print out the contents of a node as a quick reference document for later use. Info provides you with a command for doing this. In general, we recommend that you use TeX to format the document and print sections of it, by running `tex` on the texinfo source file.

`M-x print-node`

Pipes the contents of the current node through the command in the environment variable `INFO_PRINT_COMMAND`. If the variable doesn't exist, the node is simply piped to `lpr`.

Miscellaneous Commands

GNU Info contains several commands which self-document GNU Info:

`M-x describe-command`

Reads the name of an Info command in the echo area and then displays a brief description of what that command does.

`M-x describe-key`

Reads a key sequence in the echo area, and then displays the name and documentation of the Info command that the key sequence invokes.

`M-x describe-variable`

Reads the name of a variable in the echo area and then displays a brief description of what the variable affects.

`M-x where-is`

Reads the name of an Info command in the echo area, and then displays a key sequence which can be typed in order to invoke that command.

`C-h` (`get-help-window`)

`?`

Creates (or moves into) the window displaying `*Help*`, and places a node containing a quick reference card into it. This window displays the most concise information about GNU Info available.

`h` (`get-info-help-node`)

Tries hard to visit the node `(info)Help`. The info file `` `info.texi' `` distributed with GNU Info contains this node. Of course, the file must first be processed with `makeinfo`, and then placed into the location of your info directory.

Here are the commands for creating a numeric argument:

`C-u` (`universal-argument`)

Starts (or multiplies by 4) the current numeric argument. `C-u' is a good way to give a small numeric argument to cursor movement or scrolling commands; `C-u C-v' scrolls the screen 4 lines, while `C-u C-u C-n' moves the cursor down 16 lines.

`M-1` (`add-digit-to-numeric-arg`)

`M-2` ... `M-9`

Adds the digit value of the invoking key to the current numeric argument. Once Info is reading a numeric argument, you may just type the digits of the argument, without the Meta prefix. For example, you might give `C-l' a numeric argument of 32 by typing:

```
C-u 3 2 C-l
```

or

```
M-3 2 C-l
```

`C-g' is used to abort the reading of a multi-character key sequence, to cancel lengthy operations (such as multi-file searches) and to cancel reading input in the echo area.

`C-g` (`abort-key`)

Cancels current operation.

The `q' command of Info simply quits running Info.

`q` (`quit`)

Exits GNU Info.

If the operating system tells GNU Info that the screen is 60 lines tall, and it is actually only 40 lines tall, here is a way to tell Info that the operating system is correct.

`M-x set-screen-height`

Reads a height value in the echo area and sets the height of the displayed screen to that value.

Finally, Info provides a convenient way to display footnotes which might be associated with the current node that you are viewing:

`ESC C-f` (`show-footnotes`)

Shows the footnotes (if any) associated with the current node in another window. You can have Info automatically display the footnotes associated with a node when the node is selected by setting the variable `automatic-footnotes`. See section Manipulating Variables.

Manipulating Variables

GNU Info contains several variables whose values are looked at by various Info commands. You can change the values of these variables, and thus change the behaviour of Info to more closely match your environment and info file reading manner.

`M-x set-variable`

Reads the name of a variable, and the value for it, in the echo area and then sets the variable to that value. Completion is available when reading the variable name; often, completion is available when reading the value to give to the variable, but that depends on the variable itself. If a variable does *not* supply multiple choices to complete over, it expects a numeric value.

`M-x describe-variable`

Reads the name of a variable in the echo area and then displays a brief description of what the variable affects.

Here is a list of the variables that you can set in Info.

`automatic-footnotes`

When set to `On`, footnotes appear and disappear automatically. This variable is `On` by default. When a node is selected, a window containing the footnotes which appear in that node is created, and the footnotes are displayed within the new window. The window that Info creates to contain the footnotes is called `*Footnotes*'. If a node is selected which contains no footnotes, and a `*Footnotes*' window is on the screen, the `*Footnotes*' window is deleted. Footnote windows created in this fashion are not automatically tiled so that they can use as little of the display as is possible.

`automatic-tiling`

When set to `On`, creating or deleting a window resizes other windows. This variable is `Off` by default. Normally, typing `C-x 2' divides the current window into two equal parts. When

`automatic-tiling` is set to `On`, all of the windows are resized automatically, keeping an equal number of lines visible in each window. There are exceptions to the automatic tiling; specifically, windows `*Completions*' and `*Footnotes*' are *not* resized through automatic tiling; they remain the original size.

`visible-bell`

When set to `On`, GNU Info attempts to flash the screen instead of ringing the bell. This variable is `Off` by default. Of course, Info can only flash the screen if the terminal allows it; in the case that the terminal does not allow it, the setting of this variable has no effect. However, you can make Info perform quietly by setting the `errors-ring-bell` variable to `Off`.

`errors-ring-bell`

When set to `On`, errors cause the bell to ring. The default setting of this variable is `On`.

`gc-compressed-files`

When set to `On`, Info garbage collects files which had to be uncompressed. The default value of this variable is `Off`. Whenever a node is visited in Info, the info file containing that node is read into core, and Info reads information about the tags and nodes contained in that file. Once the tags information is read by Info, it is never forgotten. However, the actual text of the nodes does not need to remain in core unless a particular info window needs it. For non-compressed files, the text of the nodes does not remain in core when it is no longer in use. But de-compressing a file can be a time consuming operation, and so Info tries hard not to do it twice. `gc-compressed-files` tells Info it is okay to garbage collect the text of the nodes of a file which was compressed on disk.

`show-index-match`

When set to `On`, the portion of the matched search string is highlighted in the message which explains where the matched search string was found. The default value of this variable is `On`. When Info displays the location where an index match was found, (see section Searching an Info File), the portion of the string that you had typed is highlighted by displaying it in the inverse case from its surrounding characters.

`scroll-behaviour`

Controls what happens when forward scrolling is requested at the end of a node, or when backward scrolling is requested at the beginning of a node. The default value for this variable is `Continuous`. There are three possible values for this variable:

`Continuous`

Tries to get the first item in this node's menu, or failing that, the `Next' node, or failing that, the `Next' of the `Up'. This behaviour is identical to using the `]' (`global-next-node`) and `[' (`global-prev-node`) commands.

`Next Only`

Only tries to get the `Next' node.

`Page Only`

Simply gives up, changing nothing. If `scroll-behaviour` is `Page Only`, no scrolling command can change the node that is being viewed.

- scroll-step The number of lines to scroll when the cursor moves out of the window. Scrolling happens automatically if the cursor has moved out of the visible portion of the node text when it is time to display. Usually the scrolling is done so as to put the cursor on the center line of the current window. However, if the variable `scroll-step` has a nonzero value, Info attempts to scroll the node text by that many lines; if that is

enough to bring the cursor back into the window, that is what is done. The default value of this variable is thus placing the cursor (and the text it is attached to) in the center of the window. Setting this variable to 1 causes a kind of "smooth scrolling" which some people prefer.

- ISO-Latin When set to `On`, Info accepts and displays ISO Latin-1 characters. By default, Info assumes an ASCII character set. `ISO-Latin` tells Info that it is running in an environment where the European standard character set is in use, and allows you to input such characters to Info, as well as display them.

@raisesections

Info 1.0

TOC

(1)

Of course, you can specify both the file and node in a `--node` command; but don't forget to escape the open and close parentheses from the shell as in: `info --node '(emacs)Buffers'`

(2)

`M-x` is also a command; it invokes `execute-extended-command`. See section 'Executing an extended command' in the GNU Emacs Manual, for more detailed information.

ISPELL V3.1

ISPELL V3.1

Using ispell from emacs

Checking a single word

The simplest emacs command for calling ispell is 'M-$' (meta-dollar. On some terminals, you must type ESC-$.) This checks the spelling of the word under the cursor. If the word is found in the dictionary, then a message is printed in the echo area. Otherwise, ISPELL attempts to generate near misses.

If any near misses are found, they are displayed in a separate window, each preceded by a digit or character. If one of these is the word you wanted, just type its digit or character, and it will replace the original word in your buffer.

If no near miss is right, or if none are displayed, you have five choices:

I

Insert the word in your private dictionary. Use this if you know that the word is spelled correctly.

A

Accept the word for the duration of this editing session, but do not put it in your private dictionary. Use this if you are not sure about the spelling of the word, but you do not want to look it up immediately, or for terms that appear in your document but are not truly words. The next time you start ispell, it will have forgotten any accepted words.

SPC

Leave the word alone, and consider it misspelled if it is checked again.

R

Replace the word. This command prompts you for a string in the minibuffer. You may type more than one word, and each word you type is checked again, possibly finding other near misses. This command provides a handy way to close in on a word that you have no idea how to spell. You can keep trying different spellings until you find one that is close enough to get a near miss.

L

Lookup. Display words from the dictionary that contain a specified substring. The substring is a regular expression, which means it can contain special characters to be more selective about which words get displayed. See section 'Regexps' in emacs.

If the only special character in the regular expression is a leading ^, then a very fast binary search will be used, instead of scanning the whole file.

Only a few matching words can be displayed in the ISPELL window. If you want to see more, use the `look` program directly from the shell.

Of course, you can also type ^G to stop the command without changing anything.

If you make a change that you don't like, just use emacs' normal undo feature See section 'undo' in emacs.

Checking a whole buffer

If you want to check the spelling of all the words in a buffer, type the command M-x ispell-buffer. This command scans the file, and makes a list of all the misspelled words. When it is done, it moves the cursor to the first word on the list, and acts like you just typed M-$ See section Checking a single word.

When you finish with one word, the cursor is automatically moved to the next. If you want to stop in the middle of the list type X or ^G.

Checking a region

You may check the words in the region with the command M-x ispell-region. See See section 'mark' in emacs.

The commands available are the same as for checking a whole buffer.

Using Multiple Dictionaries

Your site may have multiple dictionaries installed: a default one (usually `` `english.hash' ``), and several others for different languages (e.g. `` `deutsch.hash' ``) or variations on a language (such as British spelling for English).

`ispell-change-dictionary`

> This is the command to change the dictionary. It prompts for a new dictionary name, with completion on the elements of `ispell-dictionary`.
>
> It changes `ispell-dictionary` and kills the old ispell process, if one was running. A new one will be started as soon as necessary.
>
> By just answering RET you can find out what the current dictionary is.

`ispell-dictionary`

> If non-nil, a dictionary to use instead of the default one. This is passed to the ispell process using the `-d` switch and is used as key in `ispell-dictionary-alist`.
>
> You should set this variable before your first call to ispell (e.g. in your `` `.emacs' ``), or use the M-x ispell-change-dictionary command to change it, as changing this variable only takes effect in a newly started ispell process.

`ispell-dictionary-alist`

> An alist of dictionaries and their associated parameters.
>
> Each element of this list is also a list:
>
> ```
> (dictionary-name
>     casechars not-casechars otherchars many-otherchars-p
>     ispell-args)
> ```
>
> dictionary-name is a possible value of variable `ispell-dictionary`, `nil` means the default dictionary.

casechars is a regular expression of valid characters that comprise a word.

not-casechars is the opposite regexp of casechars.

otherchars is a regular expression of other characters that are valid in word constructs. Otherchars cannot be adjacent to each other in a word, nor can they begin or end a word. This implies we can't check `Stevens" as a correct possessive and other correct formations.

Hint: regexp syntax requires the hyphen to be declared first here.

many-otherchars-p is non-nil if many otherchars are to be allowed in a word instead of only one.

ispell-args is a list of additional arguments passed to the ispell subprocess.

Note that the casechars and otherchars slots of the alist should contain the same character set as casechars and otherchars in the language `.aff' file (e.g., `english.aff').

Old Emacs

Until ispell becomes part of the standard emacs distribution, you will have to explicitly request that it be loaded. Put the following lines in your emacs init file See section 'init file' in emacs.

```
(autoload 'ispell-word "ispell" "Check the spelling of word in buffer." t)
(autoload 'ispell-region "ispell" "Check the spelling of region." t)
(autoload 'ispell-buffer "ispell" "Check the spelling of buffer." t)
(global-set-key "\e$" 'ispell-word)
```

(It will do no harm to have these lines in your init file even after ispell is installed by default.)

Your private dictionary

Whenever ispell is started the file `.ispell_words' is read from your home directory (if it exists). This file contains a list of words, one per line. The order of the words is not important, but the case is. Ispell will consider all of the words good, and will use them as possible near misses.

The I command adds words to `.ispell_words', so normally you don't have to worry about the file. You may want to check it from time to time to make sure you have not accidentally inserted a misspelled word.

All commands in emacs mode

DIGIT

Select a near miss

I

Insert into private dictionary

A

Accept for this session

SPC

Skip this time

R

Replace with one or more words

L

Lookup: search the dictionary using a regular expression

M-$

Check word

M-x ispell-buffer

Check buffer

M-x ispell-region

Check region

M-x ispell-change-dictionary

Select different dictionary.

Definition of a near miss

Two words are near each other if they can be made identical with one of the following changes to one of the words:

```
Insert a blank space
Interchange two adjacent letters.
Change one letter.
Delete one letter.
Add one letter.
```

Someday, perhaps ispell will be extended so that words that sound alike would also be considered near misses. If you would like to implement this, see Knuth, Volume 3, page 392 for a description of the Soundex algorithm which might apply.

Where it came from

Ispell has a long and convoluted history. Originally called SPELL, it was written by Ralph E. Gorin in 1971. That version was written in assembly language for the DEC PDP-10 to run under the WAITS operating system at the Stanford Artificial Intelligence Laboratory. Subsequent versions, also in PDP-10 assembly language, were developed for the BBN TENEX, MIT ITS, and DEC TOPS-10 and TOPS-20 operating systems. It was later revised by W. E. Matson (1974), and W. B. Ackerman (1978), changing its name to ISPELL in the process.

In 1983, Pace Willisson (pace@ai.mit.edu) converted this version to the C language and modified it to work under Unix.

In 1987, Walt Buehring revised and enhanced ispell, and posted it to the Usenet along with a dictionary. In addition, Walt wrote the first version of "ispell.el", the emacs interface.

Geoff Kuenning (geoff@ITcorp.com, that's me, and by the way I pronounce it "Kenning") picked up this version, fixed many bugs, and added further enhancements. In 1988 I got ambitious and rewrote major portions of the code, resulting in the table-driven multi-lingual version. Ken Stevens (stevens@hplabs.hp.com) made overwhelming contributions to the elisp support to produce the version you are using now.

Due to a misunderstanding involving the Free Software Foundation, it later became necessary to rename this version to ispell to avoid confusion on the part of users.

Many other enhancements and bug fixes were provided by other people. Although I omit mention here due to space, many of these people have also made significant contributions to the version of ispell you see here. For a full list of people who have contributed to ispell, refer to the file `Contributors' which is distributed with the ispell sources.

Geoff Kuenning
geoff@ITcorp.com

Kpathsearch library

for version 1.6

February 1994

Karl Berry

Kpathsearch library

@defcodeindex cm @defcodeindex fl @defcodeindex op

Introduction

This manual corresponds to version 1.6 of the Kpathsearch library, released in February 1994.

The library's fundamental purpose is to look up a file in a list of directories specified by the user.

The following software, all of which I maintain, uses this library:

- web2c
- Xdvik
- Dvipsk (see section 'Introduction' in Dvips: A DVI driver)
- GNU font utilities (see section 'Introduction' in GNU font utilities)

The library is still under development. I do not promise to keep the interface unchanged. If you have comments or suggestions, please send them to me (see section Reporting bugs for the address).

Currently, I distribute the library under the GNU General Public License (see section GNU GENERAL PUBLIC LICENSE). In summary, this means if you write a program using the library, you must (offer to) distribute the source, and allow anyone to modify the source and distribute their modifications.

If you have a problem with this, please contact me. I would consider putting the library under the GNU Library General Public License, which would permit you to distribute the source only to the library, not the your program using it. But I will only do this if someone actually says they will not use the library under the GPL conditions, and would use it under the LGPL.

This manual contains a few references to the C source for the library; they're there to help programmers reconcile my statements in this documentation with the Awful Truth of the code. If you're not a programmer, just ignore them.

Installation

Here are the basic steps for configuration and installation:

@flindex configure

1. `configure`. This automatically determines most system dependencies.

@flindex Makefile, editing @flindex c-auto.h, editing

2. If necessary, edit the definitions in `` `Makefile' `` or `` `c-auto.h' ``. See section System dependencies, below, for additional definitions you may need to make.
3. `make`.
4. `make install`. This installs the library, header files, and documentation.
5. `make distclean`. This removes all files created by the build.

Since I only distribute Kpathsea as part of another package, you will probably be doing the above in a top-level directory that contains a `Makefile', `kpathsea', and the other package. But you can do the installation in `kpathsea' itself, if you only want to install the library, not the other package.

System dependencies

Although `configure` can reliably determine most aspects of your system, there are a few things it does not do.

Default paths

The directories into which files are installed are defined in the top-level `` `Makefile' ``. The default paths which programs use to search for inputs are defined in `` `kpathsea/paths.h' ``. You will almost certainly want to change either or both of these to match your preferred local directory structure.

Both of these files are automatically created: the `` `Makefile' `` at `configure' time, `` `kpathsea/paths.h' `` at compile time. Thus, to change the default paths, the simplest thing to do is edit the template files `` `Makefile.in' `` and `` `paths.h.in' `` before running `configure'.

The Make definitions are all repeated in several `` `Makefile' ``s; but changing the top-level `` `Makefile' `` should suffice, as it passes down all the variable definitions, thus overriding the submakes. (The definitions are repeated so people can potentially run Make in the subdirectories.)

The file `` `kpathsea/HIER' `` has some explanation of the default setup.

A caveat: If you put `$HOME' or `~' in any of the paths, do not search that component recursively: when you run as root, you might wind up searching (say) `` `/zoneinfo/Brazil/tex/macros' ``. And of course it will take quite some time to look at every directory on the system.

wchar_t

The upshot of all the following is that if you get error messages regarding `wchar_t`, try defining `NO_FOIL_X_WCHAR_T`. This is reported to be necessary on Linux.

`wchar_t` has caused infinite trouble. None of my code ever uses `wchar_t`; all I want to do is include X header files and various system header files, compiling with GCC. This seems an impossible task.

The X11R5 `` `<Xlib.h>' `` and GCC's `` `<stddef.h>' `` have conflicting definitions for wchar_t.

The particulars: `` `<X11/Xlib.h>' `` from MIT X11R5 defines `wchar_t` if `X_WCHAR` is defined, which is defined if `X_NOT_STDC_ENV` is defined, and we define *that* if `STDC_HEADERS` is not defined (`configure' decides if STDC_HEADERS gets defined). But when compiling with gcc on SunOS 4.1.x, `STDC_HEADERS` is not defined (`` `string.h' `` doesn't declare the `mem'* functions), so we do get X's `wchar_t`---and we also get gcc's `wchar_t` from its `` `<stddef.h>' ``. Result: conflicting definitions.

On the other hand, SunOS 4.1.1 with some other X configurations actually needs GCC to define `wchar_t`, and fails otherwise.

My current theory is to define `wchar_t` to a nonsense symbol before the X include files are read; that way its definition (if any) will be ignored by other system include files. Going along with that, define `X_WCHAR` to tell X not to use `` `<stddef.h>' ``, that we've already included, but instead to make its own definition.

But this is not the end of the story. The X11 include files distributed with DG/UX 5.4.2 for the Aviion have been modified to include `` `<_int_wchar_t.h>' `` if `X_WCHAR`, so our `#define` will not have any typedef to change--but the uses of `wchar_t` in the X include files will be changed to reference this undefined symbol. So there's nothing to foil in this case; I don't know how to detect this automatically, so it's up to you to define `NO_FOIL_X_WCHAR_T` yourself.

putenv

If you are on a Net2/BSD or other system which has a `putenv` that knows how to avoid garbage when the same environment variable is set many times, define `SMART_PUTENV`. (And if you can write a test for this that could be incorporated into the `configure` script, please send it to the address given in section Reporting bugs.)

The `configure` script

(This section is largely from the Autoconf manual, by David MacKenzie. See section 'Running `configure` scripts' in Autoconf.)

The `configure` script that comes with this distribution was generated by the Autoconf program. Thus, you can regenerate `configure` by rerunning Autoconf. You might want to do this if a new version of Autoconf is released, for example.

The purpose of `configure` is to adapt the present source to the particulars of your system. For example, the name of the directory header file (`` `dirent.h' `` or `` `sys/dir.h' ``), whether the GNU C compiler `gcc` is available, and so on.

@flindex #! in `configure` script Normally, you do not need to give any options to `configure`; just to the directory with the source code and type `configure'. Exceptions: if `.' is not in your `PATH`, you r type `./configure'; if you are using a non-Bourne compatible shell on systems that do not support `#!', you must type `sh configure'.

@opindex -v option to `configure` @flindex /dev/null Running `configure` takes a minute or two. While it is running, it prints some messages that tell what it is doing. If you don't want to see the messages, run `configure` with its standard output redirected to `/dev/null'; e.g., `configure >/dev/null'. On the other hand, if you want to see even more messages, give `configure` the `-v' option.

To compile the package in a different directory from the one containing the source code, you must use a variant of Make that supports the `VPATH` variable, such as GNU Make. `cd` to the directory where you want the object files and executables to go and run `configure` with the option `--srcdir=dir', where dir is the directory that contains the source code. Using this option is unnecessary if the source code is in the parent directory of the one in which you are compiling; `configure` automatically checks for the source code in `..' if it does not find it in `.'.

@flindex /usr/local default prefix `configure` (in the top-level directory) guesses the default installation prefix (we'll call it `$(prefix)`, which is the corresponding Make variable) by looking for a directory which contains an executable named `tex', and using its parent. In the package subdirectories, such as `kpathsea', `configure` doesn't try to guess the prefix (to avoid a conflict with the guess made at the top level); the default `/usr/local' is left.

@opindex --prefix option to `configure` @opindex --exec-prefix option to `configure` You can override this default guess for the installation prefix by giving `configure` the option `--prefix=path'. You can also specify separate installation prefixes for architecture-specific files and architecture-independent files by giving `configure` the option `--exec-prefix=xpath' (which substitutes for the Make variable `$(exec_prefix)'). Then xpath will be the prefix for installing programs and libraries. Data files and documentation will still use `$(prefix)'.

@flindex config.status @opindex --no-create option to `configure` @opindex --recheck option to `config.status` You can tell `configure` to figure out the configuration for your system, and record it in a file `config.status', without actually configuring the package. To do this, give `configure` the `--no-create' option. Later, you can run `./config.status' to actually configure the package. This option is useful mainly in `Makefile' rules for updating `config.status' and the `Makefile' itself. You can also give `config.status' the `--recheck' option, which makes it rerun `configure` with the same arguments you used before. This is useful if you change `configure`.

`configure` ignores any other arguments that you give it.

On systems that require unusual options for compilation or linking that the package's `configure` script does not know about, you can give `configure` initial values for variables by setting them in the environment. In Bourne-compatible shells, you can do that on the command line like this:

```
CC='gcc -traditional' LIBS=-lposix sh configure
```

The Make variables that you might want to override with environment variables when running `configure` are:

(For these variables, any value given in the environment overrides the value that `configure` would

choose.)

`CC`

The C compiler program. The default is `gcc` if that is in your `PATH`, `cc` otherwise.

`INSTALL`

The program to use to install files. The default is `install` if you have it, `cp` otherwise.

(For these variables, any value given in the environment is added to the value that `configure` chooses.)

`DEFS`

Configuration options, in the form `-Dfoo -Dbar...'.

`LIBS`

Libraries to link with, in the form `-lfoo -lbar...'.

@flindex configure.in Of course, problems requiring manual intervention (e.g., setting these variables) should ideally be fixed by updating either the Autoconf macros or the `` `configure.in' `` file for that package.

Reporting bugs

@flindex tex-k@cs.umb.edu bug address If you encounter problems, report them to `tex-k@cs.umb.edu'. Include the version number of the library, the system you are using, and enough information to reproduce the bug in your report. To get on this mailing list yourself, email `tex-k-request@cs.umb.edu' with a message whose body contains a line

```
subscribe your-preferred-email-address
```

To avoid wasted effort and time (both mine and yours), I strongly advise applying the principles given in the GNU C manual (see section 'Reporting Bugs' in The GNU CC manual) to your bug reports for any software.

Please also report bugs in this documentation--not only factual errors, but unclear explanations, typos, wrong fonts, ...

When compiling with old C compilers, you may get some warnings about "illegal pointer combinations". These are spurious; ignore them. I do not want to clutter up the source with casts to get rid of them. In general, if you have trouble with a system C compiler, I advise trying the GNU C compiler.

Path specifications

This chapter describes the user interface of the path specifications that the Kpathsearch library implements.

Conceptually, there are two distinct stages: generating a list of directories in which to search, and then looking up files using that list. The sections below describe each of these in turn.

In the implementation, however, these stages are interleaved--directory lists are only generated as needed for a particular file lookup, then they are cached for future lookups. (Analogous to lazy evaluation in programming languages.) This implies that directories that are created *during* the run are not seen.

Directory list generation

Kpathsearch constructs a directory list from an environment variable var set by the user, (possibly) a setting from a configuration file, and a default path set at compile time. Each of these are colon-separated lists of directories. If var is set, its value is used; otherwise, if a config file defines a value, that value is used; otherwise, the compilation default is used. In any case, once the path specification to use is determined, its evaluation is independent of its source.

The "colon" and "slash" mentioned below aren't necessarily `:' and `/' on non-Unix systems; the library tries to adapt these characters to other operating systems' conventions.

The following subdirectories explain the various kinds of expansion the path is subjected to. After expansion, nonexistent directories in the path is ignored.

Default expansion

If an environment variable or config file value has a leading or trailing or doubled colon, the default path is inserted at that point.

Putting an extra colon into the default value has unpredictable results, and may cause the program to crash, so installers beware.

Tilde expansion

A leading `~' or `~user' in a path component is replaced by the current or user's home directory, respectively.

If user is invalid, or the home directory cannot be determined, Kpathsea uses `.' instead.

Variable expansion

A construct `$foo' or `${foo}' is replaced by the expansion of the environment variable `foo'. In the first case, the variable name consists of consecutive alphanumeric-or-underscore characters. In the second, the variable name consists of everything between the braces.

Remember to quote the `$"s and braces as necessary for your shell.

Shell variable values cannot be seen by Kpathsea.

Subdirectory expansion

If a component directory d contains `//', all subdirectories of d are included in the path: first those subdirectories directly under d, then the subsubdirectories under those, and so on. At each level, the order in which the directories are searched is unspecified.

If you specify any filename components after the `//', only subdirectories which have those components are included. For example, `/a//b' would expand into directories `/a/1/b', `/a/2/b', `/a/1/1/b', and so on, but not `/a/b/c' or `/a/1'.

Subdirectory problems

Perhaps the first problem is best put as a question-and-answer.

Question: I know all about slow starting TeX `:-)'. How do I organize the directory scheme to avoid the slowness, while at the same time enjoying a structured inputs directory?

(Naturally, this applies to any Kpathsea-using program, not just TeX.)

I will give the false Short Answer first, then the Real Explanation.

The Short Answer: in your equivalent of `/usr/local/lib/tex/macros//' and `.../fonts//', make each subdirectory contain either 1) only directories; or 2) only files.

As long as you do not have (literally) hundreds of subdirectories, this should cure the problem. It has in every case I have been told about.

The Real Explanation: the thing that makes TeX slow is calling `stat` (if you don't know what `stat` (2) is, ignore this explanation) on "too many" pathnames, where "too many" is some nebulous number depending on things like whether the filesystem is NFS-mounted or not, whether it's on a fast disk, whether your Fast File System implementation is really Fast, etc., etc.

(Side note: If you're curious, you can find this number by writing a program that does nothing but read filenames (presumably from a file) and stat them, and see how many pathnames make the execution time noticeable. On the systems I use (Suns with an NFS-mounted directory, ISC 2.2.1 and a local directory), it's several hundred, at least. On an NFS-mounted mounted directory under Solaris 2.1, 150 is quite slow, according to `hammer@kis.uni-freiburg.de'.)

Whether or not it's directories or files that are being stat-ed is irrelevant (this is why the Short Answer is false). It's sheer numbers that count.

@flindex debug.h If you think your directory structure is ok, and you're still experiencing slowness, I advise running TeX (or whatever program) under a debugger, setting the bit `DEBUG_STAT` in the variable `kpathsea_debug` (see `debug.h') to one and seeing exactly what is getting stat-ed. If only a few things are getting stat-ed, and TeX is still slow, tell me.

I should also mention "the trick", which I stole from GNU find. (Matthew Farwell `<dylan@ibmpcug.co.uk>' suggested it, and David MacKenzie `<djm@gnu.ai.mit.edu>' implemented it, I believe.)

The trick is that in every real Unix implementation (that I know about) (as opposed to the POSIX specification), a directory which contains no subdirectories will have exactly two links (specifically, one each for `.' and `..'). That is to say, the `st_nlink` field in the stat structure will be two. Thus, the path searching code doesn't have to stat every entry in the bottom-level directories--it can check `st_nlink`, and if it's two, it knows there are no subdirectories.

But if you have a directory that contains *one* subdirectory and five hundred files, st_nlink will be 3, and Kpathsea has to stat every one of those 501 entries. Therein lies slowness.

You can disable the trick by undefining UNIX_ST_LINK in `kpathsea/config.h'.

The subdirectory searching has one other known (and irreconciliable) deficiency. If a directory d being

searched for subdirectories contains plain files and symbolic links to other directories, but no true subdirectories, d will be considered a leaf directory, i.e., the symbolic links will not be followed.

The directory immediately followed by the `//', however, is always searched for subdirectories, even if it is a "leaf". We do this since presumably you would not have asked for the directory to be searched for subdirectories if you didn't want it to be.

This is a consequence of the trick explained above. You can work around this problem by simply creating an empty dummy subdirectory in d; then d will no longer be a leaf, and the symlinks will be followed.

Path specification example

For example, the following value for an environment variable says to search the following: the current user's `fonts' directory and alll its subdirectories, then the directory `fonts' in user `karl's home directory, and finally the system default directories specified at compilation time.

```
~/fonts//:~karl/fonts:
```

File lookups

Given the directory list generated from the rules in the previous section, looking up a file presents no problem at all: we just look in each directory in the list in turn, and return the first one found.

The only complication is if the filename is absolute or explicitly relative, i.e., (under Unix-like operating systems) starts with `/' or `./' or `../'. Then the library does not use the directory list at all. Instead, the file is simply searched for in the given directory.

TeX support

Although the basic features in Kpathsea can be used for any type of path searching, I wrote Kpathsea specifically for TeX system programs. I had been struggling with the programs I was using (Dvips, Xdvi, and TeX itself) having slightly different notions of how to specify paths; and debugging was painful, since no code was shared.

Therefore, Kpathsea provides some TeX-specific features. Indeed, many of the supposedly generic path searching features were provided because it seemed useful in conTeXt, particularly for font lookup.

TeX environment variables

Kpathsea defines a sequence of environment variables to search for each type of TeX file. This makes it easy for different programs to check the same environment variables, in the same order.

The following table lists the environment variables searched for each file type in the order they are searched (and a brief description of the file type). That is, only if the first variable is unset is the second variable checked, and so on. If none are set, a default set at compilation time is used.

`.base'

@flindex .base (Metafont memory dump) `MFBASES'

`.bib'

@flindex .bib (BibTeX source) `BIBINPUTS'

`.bst'

@flindex .bst (BibTeX style file) `BSTINPUTS', `TEXINPUTS'

`.fmt'

@flindex .fmt (TeX memory dump) `TEXFORMATS'

`gf'

@flindex .gf (generic font bitmap) `GFFONTS', `GLYPHFONTS', `TEXFONTS'

`.mf'

@flindex .mf (Metafont source) `MFINPUTS'

`mf.pool'

@flindex .pool (Metafont program strings) `MFPOOL'

`pk'

@flindex .pk (packed bitmap font) `PKFONTS', `TEXPKS', `GLYPHFONTS', `TEXFONTS'

`.tex'

@flindex .tex (TeX source) `TEXINPUTS'

`tex.pool'

@flindex .pool (TeX program strings) `TEXPOOL'

`.tfm'

@flindex .tfm (TeX font metrics) `TFMFONTS', `TEXFONTS'

`.vf'

@flindex .vf (virtual font) `VFFONTS', `TEXFONTS'

@flindex kpathsea/filefmt.h These lists are defined in the source file `kpathsea/filefmt.h'.

For the font variables, the intent is that:

1. `TEXFONTS' is the default for everything.
2. `GLYPHFONTS' the default for bitmap (or, more precisely, non-metric) files.
3. Each format has its own variable.
4. Not shown in the table is that each program can and should have its own path as well---`XDVIFONTS' and `DVIPSFONTS' in the case of Xdvik and Dvipsk.

Glyph lookup

@flindex kpathsea/tex-glyph.c Kpathsea provides a routine (`kpse_find_glyph_format` in `kpathsea/tex-glyph.c`) which searches for a bitmap font in GF or PK format (or either) given a font name (e.g., `cmr10') and a resolution (e.g., 300).

The search is based solely on filenames, not file contents--if a PK file is named `cmr10.300gf', it will be found as a GF file.

Here is an outline of the search strategy (details in the sections below) for a file name at resolution dpi. The search stops at the first successful lookup.

1. Look for an existing file name.dpi in the specified formats.
2. If name is an alias for a file a in some "fontmap" file, look for a.dpi.
3. Run an external script (typically named `MakeTeXPK`) to generate the font.
4. Look for fallback.dpi, where fallback is some last-resort font (typically `cmr10').

Basic glyph lookup

When Kpathsea looks for a bitmap font name at resolution dpi in a format format, it first checks if a file `name.dpiformat' exists; for example, `cmr10.300pk'. Kpathsea looks for a PK file first, then a GF file.

If that fails, Kpathsea looks for a font with a close-enough dpi. "Close enough" is defined (by the macro `KPSE_BITMAP_TOLERANCE` in `` `kpathsea/tex-glyph.h' ``) to be `dpi / 500 + 1`, which is slightly more than the 0.2% allowed by the DVI standard.

The filename is not hardwired to `name.dpiformat'; you can change it by setting the environment variable `KPATHSEA_BITMAP_NAME' to a different specification. The default spec is

```
$KPATHSEA_NAME.$KPATHSEA_DPI$KPATHSEA_FORMAT
```

Kpathsea sets these environment variables as it runs, as appropriate for each lookup. You can use any environment variables you like, not just these, in the spec.

Fontmap

If a bitmap font is not found, Kpathsea looks through any fontmap files for an alias for the original font name. I implemented this for two reasons:

1. An alias name is limited in length only by your available memory, not by whatever bizarre limitations your filesystem might impose. Therefore, if you want to ask for `` `Adobe-Lucida-Bold-Sans=Typewriter--10' `` instead of `` `plcbst10' ``, you can.
2. A few fonts have historically had multiple names: specifically, LaTeX's "circle font" has variously been known as `` `circle10' ``, `` `lcircle10' ``, and `` `lcirc10' ``. Aliases can make all the names equivalent, so that it no longer matters what the name of the installed file is; TeX documents will find their favorite name.

The format of fontmap files that implement these ideas is straightforward: the first word on each line is the true filename; the second word is the alias; subsequent words are ignored. A word is just a sequence of non-whitespace characters. Blank lines are ignored; comments start with `%' and continue to end-of-line.

If an alias has an extension, it matches only those files with that extension; otherwise, it matches anything with the same root, regardless of extension. For example, an alias `foo.tfm' matches only when exactly `` `foo.tfm' `` is being searched for; but an alias `foo' matches `` `foo.vf' ``, `` `foo.300pk' ``, or whatever.

As an example, here are the fontmap entries that make the circle fonts equivalent:


```
circle10        lcircle10
circle10        lcirc10
lcircle10       circle10
lcircle10       lcirc10
lcirc10         circle10
lcirc10         lcircle10
```

`MakeTeX'... scripts

If Kpathsea cannot find a bitmap font, by either its original name or a fontmap alias, it can be configured to invoke an external program to create it. The same mechanism can be used for other nonexistent files.

The script is passed the name of the file to create and possibly other arguments, as explained below. It must echo the full pathname of the file it created (and nothing else) to standard output; it can write diagnostics to standard error.

`MakeTeX'... script names

@flindex tex-make.c The following table shows the default name of the script for each possible file types. (The source is the variable `kpse_make_specs` in `kpathsea/tex-make.c'.)

`MakeTeXPK'
: Glyph fonts.

`MakeTeXTeX'
: TeX input files.

`MakeTeXMF'
: Metafont input files.

`MakeTeXTFM'
: TFM files.

These names are overridden by an environment variable specific to the program---`DVIPSMAKEPK' and `XDVIMAKEPK', in the case of Dvipsk and Xdvik.

`MakeTeX'... script arguments

The first argument to a `MakeTeX'... script is always the name of the file to be created. In the case of `MakeTeXPK', there are three or four additional arguments passed, via corresponding environment variables:

1. The dpi to make the font at (`KPATHSEA_DPI').
2. The "base dpi" the program is operating at (`MAKETEX_BASE_DPI').
3. A "magstep" string suitable for assigning to the Metafont `mag` variable (`MAKETEX_MAG').
4. Possibly, a Metafont mode name for assigning to the Metafont `mode` variable (`MAKETEX_MODE').

Kpathsea sets `KPATHSEA_DPI' appropriately for each font. It's up to the program using Kpathsea to set the others.

You can change the specification for the arguments passed to the external script by setting the environment

variable named as the script name, but all capitals---`MAKETEXPK', for example. If you've changed the script name by setting (say) `DVIPSMAKEPK' to `foo', then the spec is taken from the environment variable `FOO'.

The spec can contain any variable references, to the above variables or any others you might have set. As an example, the default spec for `MakeTeXPK' is

```
$KPATHSEA_DPI $MAKETEX_BASE_DPI $MAKETEX_MAG $MAKETEX_MODE
```

The name of the file to be created being passed as the first argument cannot be changed.

Fallback font

If a bitmap font cannot be found or created at the requested size, Kpathsea looks for the font at a set of fallback resolutions. You specify these resolutions as a colon-separated list (like search paths). Kpathsea looks first for a program-specific environment variable (`DVIPSSIZES' and `XDVISIZES', in the case of Dvipsk and Xdvik), then the environment variable `TEXSIZES', then a default specified at compilation time (the Make variable `default_texsizes`). You can set this list to be empty if you prefer to find fonts at their stated size or not at all.

Finally, if the font cannot be found even at the fallback resolutions, Kpathsea looks for a fallback font, typically `cmr10'. Programs must enable this feature by assigning to the global variable `kpse_fallback_font`; the default is no such fallback font.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

```
Copyright (C) 1989, 1991 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the

software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.
2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work bas on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
    1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
    2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
    3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

    These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

    Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

    In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.
4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
    1. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
    2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
    3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

    The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus

any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
`Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
```

Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Regain your programming freedom

Until a few years ago, programmers in the United States could write any program they wished. This freedom has now been taken away by two developments: software patents, which grant the patent holder an absolute monopoly on some programming technique, and user interface copyright, which forbid compatible implementations of an existing user interface.

In Europe, especially through the GATT treaty, things are rapidly approaching the same pass.

Software patents

The U.S. Patent and Trademark Office has granted numerous software patents on software techniques. Patents are an absolute monopoly--independent reinvention is precluded. This monopoly lasts for seventeen years, i.e., forever (with respect to computer science).

One patent relevant to TeX is patent 4,956,809, issued to the Mark Williams company on September 11, 1990, applied for in 1982, which covers (among other things)

> representing in a standardized order consisting of a standard binary structure file stored on auxiliary memory or transported on a communications means, said standardized order being different from a different order used on at least one of the different computers;
>
> Converting in each of the different computers binary data read from an auxiliary data storage or communications means from the standardized order to the natural order of the respective host computer after said binary data are read from said auxiliary data storage or communications means and before said binary data are used by the respective host computer; and
>
> Converting in each of the different computers binary data written into auxiliary data storage or communications means from the natural order of the respective host computer to the standardized order prior to said writing.

... in other words, storing data on disk in a machine-independent order, as the DVI, TFM, GF, and PK file formats specify. Even though TeX is "prior art" in this respect, the patent was granted (the patent examiners not being computer scientists, even less computer typographers). Since there is a strong presumption in the courts of a patent's validity once it has been granted, there is a good chance that users or implementors of TeX could be successfully sued on the issue.

As another example, the X window system, which was intended to be able to be used freely by everyone, is now being threatened by two patents: 4,197,590 on the use of exclusive-or to redraw cursors, held by Cadtrak, a litigation company (this has been upheld twice in court); and 4,555,775, held by AT&T, on the use of backing store to redraw windows quickly.

Here is one excerpt from a recent mailing by the League for Programming Freedom (see section What to do?) which I feel sums up the situation rather well. It comes from an article in Think magazine, issue #5, 1990. The comments after the quote were written by Richard Stallman.

> "You get value from patents in two ways," says Roger Smith, IBM Assistant General Counsel, intellectual property law. "Through fees, and through licensing negotiations that give IBM access to other patents.
>
> "The IBM patent portfolio gains us the freedom to do what we need to do through cross-licensing--it gives us access to the inventions of others that are the key to rapid innovation. Access is far more valuable to IBM than the fees it receives from its 9,000 active patents. There's no direct calculation of this value, but it's many times larger than the fee income, perhaps an order of magnitude larger."

This information should dispel the belief that the patent system will "protect" a small software developer from competition from IBM. IBM can always find patents in its collection which the small developer is infringing, and thus obtain a cross-license.

However, the patent system does cause trouble for the smaller companies which, like IBM, need access to patented techniques in order to do useful work in software. Unlike IBM, the smaller companies do not have 9,000 patents and cannot usually get a cross-license. No matter how hard they try, they cannot have enough patents to do this.

Only the elimination of patents from the software field can enable most software developers to continue with their work.

The value IBM gets from cross-licensing is a measure of the amount of harm that the patent system would do to IBM if IBM could not avoid it. IBM's estimate is that the trouble could easily be ten times the good one can expect from one's own patents--even for a company with 9,000 of them.

User interface copyright

(This section is copied from the GCC manual, by Richard Stallman.)

> *This section is a political message from the League for Programming Freedom to the users of the GNU font utilities. It is included here as an expression of support for the League on my part.*

Apple, Lotus and Xerox are trying to create a new form of legal monopoly: a copyright on a class of user interfaces. These monopolies would cause serious problems for users and developers of computer software and systems.

Until a few years ago, the law seemed clear: no one could restrict others from using a user interface; programmers were free to implement any interface they chose. Imitating interfaces, sometimes with changes, was standard practice in the computer field. The interfaces we know evolved gradually in this way; for example, the Macintosh user interface drew ideas from the Xerox interface, which in turn drew on work done at Stanford and SRI. 1-2-3 imitated VisiCalc, and dBase imitated a database program from JPL.

Most computer companies, and nearly all computer users, were happy with this state of affairs. The companies that are suing say it does not offer "enough incentive" to develop their products, but they must have considered it "enough" when they made their decision to do so. It seems they are not satisfied with the opportunity to continue to compete in the marketplace--not even with a head start.

If Xerox, Lotus, and Apple are permitted to make law through the courts, the precedent will hobble the software industry:

- Gratuitous incompatibilities will burden users. Imagine if each car manufacturer had to arrange the pedals in a different order.
- Software will become and remain more expensive. Users will be "locked in" to proprietary interfaces, for which there is no real competition.
- Large companies have an unfair advantage wherever lawsuits become commonplace. Since they can easily afford to sue, they can intimidate small companies with threats even when they don't really have a case.
- User interface improvements will come slower, since incremental evolution through creative imitation will no longer be permitted.
- Even Apple, etc., will find it harder to make improvements if they can no longer adapt the good ideas that others introduce, for fear of weakening their own legal positions. Some users suggest that this stagnation may already have started.
- If you use GNU software, you might find it of some concern that user interface copyright will make it hard for the Free Software Foundation to develop programs compatible with the interfaces that you already know.

What to do?

(This section is copied from the GCC manual, by Richard Stallman.)

To protect our freedom from lawsuits like these, a group of programmers and users have formed a new grass-roots political organization, the League for Programming Freedom.

The purpose of the League is to oppose new monopolistic practices such as user-interface copyright and software patents; it calls for a return to the legal policies of the recent past, in which these practices were not allowed. The League is not concerned with free software as an issue, and not affiliated with the Free Software Foundation.

The League's membership rolls include John McCarthy, inventor of Lisp, Marvin Minsky, founder of the Artificial Intelligence lab, Guy L. Steele, Jr., author of well-known books on Lisp and C, as well as Richard Stallman, the developer of GNU CC. Please join and add your name to the list. Membership dues in the League are $42 per year for programmers, managers and professionals; $10.50 for students; $21 for others.

The League needs both activist members and members who only pay their dues.

To join, or for more information, phone (617) 492-0023 or write to:

```
League for Programming Freedom
1 Kendall Square #143
P.O. Box 9171
Cambridge, MA 02139
```

You can also send electronic mail to `league@prep.ai.mit.edu`.

Here are some suggestions from the League for things you can do to protect your freedom to write programs:

- Don't buy from Xerox, Lotus or Apple. Buy from their competitors or from the defendants they ar suing.
- Don't develop software to work with the systems made by these companies.
- Port your existing software to competing systems, so that you encourage users to switch.
- Write letters to company presidents to let them know their conduct is unacceptable.
- Tell your friends and colleagues about this issue and how it threatens to ruin the computer industry.
- Above all, don't work for the look-and-feel plaintiffs, and don't accept contracts from them.
- Write to Congress to explain the importance of this issue.

```
House Subcommittee on Intellectual Property
2137 Rayburn Bldg
Washington, DC 20515

Senate Subcommittee on Patents, Trademarks and Copyrights
United States Senate
Washington, DC 20510
```

(These committees have received lots of mail already; let's give them even more.)

Express your opinion! You can make a difference.

Index

$

/

:

a

b

c

TOC

d

e

f

g

h

i

k

l

m

TOC

n

p

r

s

t

u

v

W

X

~

to the GNU C++ Library

last updated April 29, 1992

for version 2.0

Doug Lea (dl@g.oswego.edu)

Go to the next section.

Note: The GNU C++ library is still in test release. You will be performing a valuable service if you report any bugs you encounter.

Contributors to GNU C++ library

Aside from Michael Tiemann, who worked out the front end for GNU C++, and Richard Stallman, who worked out the back end, the following people (not including those who have made their contributions to GNU CC) should not go unmentioned.

- Doug Lea contributed most otherwise unattributed classes.
- Per Bothner contributed the iostream I/O classes.
- Dirk Grunwald contributed the Random number generation classes, and PairingHeaps.
- Kurt Baudendistel contributed Fixed precision reals.
- Doug Schmidt contributed ordered hash tables, a perfect hash function generator, and several other utilities.
- Marc Shapiro contributed the ideas and preliminary code for Plexes.
- Eric Newton contributed the curses window classes.
- Some of the I/O code is derived from BSD 4.4, and was developed by the University of California, Berkeley.
- The code for converting accurately between floating point numbers and their string representations was written by David M. Gay of AT&T.

Go to the next section.

Go to the previous, next section.

Installing GNU C++ library

1. Read through the README file and the Makefile. Make sure that all paths, system-dependent compile switches, and program names are correct.
2. Check that files `values.h', `stdio.h', and `math.h' declare and define values appropriate for your system.
3. Type `make all' to compile the library, test, and install. Current details about contents of the tests and utilities are in the `README' file.

Go to the previous, next section.

Go to the previous, next section.

Trouble in Installation

Here are some of the things that have caused trouble for people installing GNU C++ library.

1. Make sure that your GNU C++ version number is at least as high as your libg++ version number. For example, libg++ 1.22.0 requires g++ 1.22.0 or later releases.
2. Double-check system constants in the header files mentioned above.

Go to the previous, next section.

Go to the previous, next section.

GNU C++ library aims, objectives, and limitations

The GNU C++ library, libg++ is an attempt to provide a variety of C++ programming tools and other support to GNU C++ programmers.

Differences in distribution policy are only part of the difference between libg++.a and AT&T libC.a. libg++ is not intended to be an exact clone of libC. For one, libg++ contains bits of code that depend on special features of GNU g++ that are either different or lacking in the AT&T version, including slightly different inlining and overloading strategies, dynamic local arrays, etc. All of these differences are minor. For example, while the AT&T and GNU stream classes are implemented in very different ways, the vast majority of C++ programs compile and run under either version with no visible difference. Additionally, all g++-specific constructs are conditionally compiled; The library is designed to be compatible with any 2.0 C++ compiler.

libg++ has also contained workarounds for some limitations in g++: both g++ and libg++ are still undergoing rapid development and testing--a task that is helped tremendously by the feedback of active users. This manual is also still under development; it has some catching up to do to include all the facilities now in the library.

libg++ is not the only freely available source of C++ class libraries. Some notable alternative sources are Interviews and NIHCL. (InterViews has been available on the X-windows X11 tapes and also from interviews.stanford.edu. NIHCL is available by anonymous ftp from GNU archives (such as the pub directory of prep.ai.mit.edu), although it is not supported by the FSF - and needs some work before it will work with g++.)

As every C++ programmer knows, the design (moreso than the implementation) of a C++ class library is something of a challenge. Part of the reason is that C++ supports two, partially incompatible, styles of object-oriented programming -- The "forest" approach, involving a collection of free-standing classes that can be mixed and matched, versus the completely hierarchical (smalltalk style) approach, in which all classes are derived from a common ancestor. Of course, both styles have advantages and disadvantages. So far, libg++ has adopted the "forest" approach. Keith Gorlen's OOPS library adopts the hierarchical approach, and may be an attractive alternative for C++ programmers who prefer this style.

Currently (and/or in the near future) libg++ provides support for a few basic kinds of classes:

The first kind of support provides an interface between C++ programs and C libraries. This includes basic header files (like `` `stdio.h' ``) as well as things like the File and stream classes. Other classes that interface to other aspects of C libraries (like those that maintain environmental information) are in various stages of development; all will undergo implementation modifications when the forthcoming GNU libc library is released.

The second kind of support contains general-purpose basic classes that transparently manage

variable-sized objects on the freestore. This includes Obstacks, multiple-precision Integers and Ratio arbitrary length Strings, BitSets, and BitStrings.

Third, several classes and utilities of common interest (e.g., Complex numbers) are provided.

Fourth, a set of pseudo-generic prototype files are available as a mechanism for generating common container classes. These are described in more detail in the introduction to container prototypes. Currently, only a textual substitution mechanism is available for generic class creation.

Go to the previous, next section.

Go to the previous, next section.

GNU C++ library stylistic conventions

- C++ source files have file extension `.cc'. Both C-compatibility header files and class declaration files have extension `.h'.
- C++ class names begin with capital letters, except for `istream` and `ostream`, for AT&T C++ compatibility. Multi-word class names capitalize each word, with no underscore separation.
- Include files that define C++ classes begin with capital letters (as do the names of the classes themselves). `stream.h' is uncapitalized for AT&T C++ compatibility.
- Include files that supply function prototypes for other C functions (system calls and libraries) are all lower case.
- All include files define a preprocessor variable _X_h, where X is the name of the file, and conditionally compile only if this has not been already defined. The `#pragma once` facility is also used to avoid re-inclusion.
- Structures and objects that must be publicly defined, but are not intended for public use have names beginning with an underscore. (for example, the `_Srep` struct, which is used only by the String and SubString classes.)
- The underscore is used to separate components of long function names, e.g., `set_File_exception_handler()`.
- When a function could be usefully defined either as a member or a friend, it is generally a member if it modifies and/or returns itself, else it is a friend. There are cases where naturalness of expression wins out over this rule.
- Class declaration files are formatted so that it is easy to quickly check them to determine function names, parameters, and so on. Because of the different kinds of things that may appear in class declarations, there is no perfect way to do this. Any suggestions on developing a common class declaration formatting style are welcome.
- All classes use the same simple error (exception) handling strategy. Almost every class has a member function named `error(char* msg)` that invokes an associated error handler function via a pointer to that function, so that the error handling function may be reset by programmers. By default nearly all call `*lib_error_handler`, which prints the message and then aborts execution. This system is subject to change. In general, errors are assumed to be non-recoverable: Library classes do not include code that allows graceful continuation after exceptions.

Go to the previous, next section.

Go to the previous, next section.

Support for representation invariants

Most GNU C++ library classes possess a method named `OK()`, that is useful in helping to verify correct performance of class operations.

The `OK()` operations checks the "representation invariant" of a class object. This is a test to check whether the object is in a valid state. In effect, it is a (sometimes partial) verification of the library's promise that (1) class operations always leave objects in valid states, and (2) the class protects itself so that client functions cannot corrupt this state.

While no simple validation technique can assure that all operations perform correctly, calls to `OK()` can at least verify that operations do not corrupt representations. For example for `String a, b, c; ... a = b + c;`, a call to `a.OK();` will guarantee that `a` is a valid `String`, but does not guarantee that it contains the concatenation of `b + c`. However, given that `a` is known to be valid, it is possible to further verify its properties, for example via `a.after(b) == c && a.before(c) == b`. In other words, `OK()` generally checks only those internal representation properties that are otherwise inaccessible to users of the class. Other class operations are often useful for further validation.

Failed calls to `OK()` call a class's `error` method if one exists, else directly call `abort`. Failure indicates an implementation error that should be reported.

With only rare exceptions, the internal support functions for a class never themselves call `OK()` (although many of the test files in the distribution call `OK()` extensively).

Verification of representational invariants can sometimes be very time consuming for complicated data structures.

Go to the previous, next section.

Go to the previous, next section.

Introduction to container class prototypes

As a temporary mechanism enabling the support of generic classes, the GNU C++ Library distribution contains a directory (`g++-include') of files designed to serve as the basis for generating container classes of specified elements. These files can be used to generate `.h' and `.cc' files in the current directory via a supplied shell script program that performs simple textual substitution to create specific classes.

While these classes are generated independently, and thus share no code, it is possible to create versions that do share code among subclasses. For example, using `typedef void* ent`, and then generating a `entList` class, other derived classes could be created using the `void*` coercion method described in Stroustrup, pp204-210.

This very simple class-generation facility is useful enough to serve current purposes, but will be replaced with a more coherent mechanism for handling C++ generics in a way that minimally disrupts current usage. Without knowing exactly when or how parametric classes might be added to the C++ language, provision of this simplest possible mechanism, textual substitution, appears to be the safest strategy, although it does require certain redundancies and awkward constructions.

Specific classes may be generated via the `genclass' shell script program. This program has arguments specifying the kinds of base types(s) to be used. Specifying base types requires two arguments. The first is the name of the base type, which may be any named type, like `int` or `String`. Only named types are supported; things like `int*` are not accepted. However, pointers like this may be used by supplying the appropriate typedefs (e.g., editing the resulting files to include `typedef int* intp;`). The type name must be followed by one of the words `val` or `ref`, to indicate whether the base elements should be passed to functions by-value or by-reference.

You can specify basic container classes using `genclass base [val,ref] proto`, where `proto` is the name of the class being generated. Container classes like dictionaries and maps that require two types may be specified via `genclass -2 keytype [val, ref], basetype [val, ref] proto`, where the key type is specified first and the contents type second. The resulting classnames and filenames are generated by prepending the specified type names to the prototype names, and separating the filename parts with dots. For example, `genclass int val List` generates class `intList` residing in files `int.List.h' and `int.List.cc'. `genclass -2 String ref int val VHMap` generates (the awkward, but unavoidable) class name `StringintVHMap`. Of course, programmers may use `typedef` or simple editing to create more appropriate names. The existence of dot seperators in file names allows the use of GNU make to help automate configuration and recompilation. An example Makefile exploiting such capabilities may be found in the `libg++/proto-kit' directory.

The `genclass` utility operates via simple text substitution using `sed`. All occurrences of the pseudo-types `<T>` and `<C>` (if there are two types) are replaced with the indicated type, and occurrences

of <T&> and <C&> are replaced by just the types, if `val` is specified, or types followed by "&" if `ref` specified.

Programmers will frequently need to edit the `.h' file in order to insert additional `#include` directives or other modifications. A simple utility, `prepend-header' to prepend other `.h' files to generated files is provided in the distribution.

One dubious virtue of the prototyping mechanism is that, because sources files, not archived library classes, are generated, it is relatively simple for programmers to modify container classes in the common case where slight variations of standard container classes are required.

It is often a good idea for programmers to archive (via `ar`) generated classes into `.a' files so that only those class functions actually used in a given application will be loaded. The test subdirectory of the distribution shows an example of this.

Because of `#pragma interface` directives, the `.cc' files should be compiled with `-O` or `-DUSE_LIBGXX_INLINES` enabled.

Many container classes require specifications over and above the base class type. For example, classes that maintain some kind of ordering of elements require specification of a comparison function upon which to base the ordering. This is accomplished via a prototype file `defs.hP' that contains macros for these functions. While these macros default to perform reasonable actions, they can and should be changed in particular cases. Most prototypes require only one or a few of these. No harm is done if unused macros are defined to perform nonsensical actions. The macros are:

`DEFAULT_INITIAL_CAPACITY`

> The initial capacity for containers (e.g., hash tables) that require an initial capacity argument for constructors. Default: 100

`<T>EQ(a, b)`

> return true if a is considered equal to b for the purposes of locating, etc., an element in a container. Default: (a == b)

`<T>LE(a, b)`

> return true if a is less than or equal to b Default: (a <= b)

`<T>CMP(a, b)`

> return an integer < 0 if a<b, 0 if a==b, or > 0 if a>b. Default: (a <= b)? (a==b)? 0 : -1 : 1

`<T>HASH(a)`

> return an unsigned integer representing the hash of a. Default: hash(a) ; where extern unsigned int hash(<T&>). (note: several useful hash functions are declared in builtin.h and defined in hash.cc)

Nearly all prototypes container classes support container traversal via `Pix` pseudo indices, as described elsewhere.

All object containers must perform either a `X::X(X&)` (or `X::X()` followed by `X::operator =(X&)`) to copy objects into containers. (The latter form is used for containers built from C++ arrays, like `VHSets`). When containers are destroyed, they invoke `X::~X()`. Any objects used in containers must have well behaved constructors and destructors. If you want to create containers that merely reference (point to) objects that reside elsewhere, and are not copied or destroyed inside the container,

you must use containers of pointers, not containers of objects.

All prototypes are designed to generate *HOMOGENOUS* container classes. There is no universally applicable method in C++ to support heterogenous object collections with elements of various subclasses of some specified base class. The only way to get heterogenous structures is to use collections of pointers-to-objects, not collections of objects (which also requires you to take responsibility for managing storage for the objects pointed to yourself).

For example, the following usage illustrates a commonly encountered danger in trying to use container classes for heterogenous structures:

```
class Base { int x; ...}
class Derived : public Base { int y; ... }

BaseVHSet s; // class BaseVHSet generated via something like
             // `genclass Base ref VHSet'

void f()
{
  Base b;
  s.add(b); // OK

  Derived d;
  s.add(d);  // (CHOP!)
}
```

At the line flagged with `(CHOP!)', a `Base::Base(Base&)` is called inside `Set::add(Base&)`---*not* `Derived::Derived(Derived&)`. Actually, in `VHSet`, a `Base::operator =(Base&)`, is used instead to place the element in an array slot, but with the same effect. So only the Base part is copied as a `VHSet` element (a so-called chopped-copy). In this case, it has an `x` part, but no `y` part; and a Base, not Derived, vtable. Objects formed via chopped copies are rarely sensible.

To avoid this, you must resort to pointers:

```
typedef Base* BasePtr;

BasePtrVHSet s; // class BaseVHSet generated via something like
                // `genclass BasePtr val VHSet'

void f()
{
  Base* bp = new Base;
  s.add(b);

  Base* dp = new Derived;
```


```
  s.add(d);  // works fine.

  // Don't forget to delete bp and dp sometime.
  // The VHSet won't do this for you.
}
```

Example

The prototypes can be difficult to use on first attempt. Here is an example that may be helpful. The utilities in the `proto-kit' simplify much of the actions described, but are not used here.

Suppose you create a class `Person`, and want to make an Map that links the social security numbers associated with each person. You start off with a file `Person.h'

```
#include <String.h>

class Person
{
  String nm;
  String addr;
  //...
public:
  const String& name() { return nm; }
  const String& address() { return addr; }
  void          print() { ... }
  //...
}
```

And in file `SSN.h',

```
typedef unsigned int SSN;
```

Your first decision is what storage/usage strategy to use. There are several reasonable alternatives here: You might create an "object collection" of Persons, a "pointer collection" of pointers-to-Persons, or even a simple String map, housing either copies of pointers to the names of Persons, since other fields are unused for purposes of the Map. In an object collection, instances of class Person "live" inside the Map, while in a pointer collection, the instances live elsewhere. Also, as above, if instances of subclasses of Person are to be used inside the Map, you must use pointers. In a String Map, the same difference holds, but now only for the name fields. Any of these choices might make sense in particular applications.

The second choice is the Map implementation strategy. Either a tree or a hash table might make sense. Suppose you want an AVL tree Map. There are two things to now check. First, as an object collection, the AVLMap requires that the elsement class contain an `X(X&)` constructor. In C++, if you don't specify such a constructor, one is constructed for you, but it is a very good idea to always do this yourself, to

avoid surprises. In this example, you'd use something like

```
class Person
{ ...;
    Person(const Person& p) :nm(p.nm), addr(p.addr) {}
};
```

Also, an AVLMap requires a comparison function for elements in order to maintain order. Rather than requiring you to write a particular comparison function, a `defs' file is consulted to determine how to compare items. You must create and edit such a file.

Before creating `Person.defs.h', you must first make one additional decision. Should the Map member functions like `m.contains(p)` take arguments (`p`) by reference (i.e., typed as `int Map::contains(const Person& p)` or by value (i.e., typed as `int Map::contains(const Person p)`. Generally, for user-defined classes, you want to pass by reference, and for builtins and pointers, to pass by value. SO you should pick by-reference.

You can now create `Person.defs.h' via `genclass Person ref defs`. This creates a simple skeleton that you must edit. First, add `#include "Person.h"` to the top. Second, edit the `<T>CMP(a,b)` macro to compare on name, via

```
#define <T>CMP(a, b) ( compare(a.name(), b.name()) )
```

which invokes the `int compare(const String&, const String&)` function from `String.h'. Of course, you could define this in any other way as well. In fact, the default versions in the skeleton turn out to be OK (albeit inefficient) in this particular example.

You may also want to create file `SSN.defs.h'. Here, choosing call-by-value makes sense, and since no other capabilities (like comparison functions) of the SSNs are used (and the defaults are OK anyway), you'd type

```
genclass SSN val defs
```

and then edit to place `#include "SSN.h"` at the top.

Finally, you can generate the classes. First, generate the base class for Maps via

```
genclass -2 Person ref SSN val Map
```

This generates only the abstract class, not the implementation, in file `Person.SSN.Map.h' and `Person.SSN.Map.cc'. To create the AVL implementation, type

```
genclass -2 Person ref SSN val AVLMap
```

This creates the class `PersonSSNAVLMap`, in `Person.SSN.AVLMap.h' and `Person.SSN.AVLMap.cc'.

To use the AVL implementation, compile the two generated `.cc' files, and specify `#include

"Person.SSN.AVLMap.h"' in the application program. All other files are included in the right ways automatically.

One last consideration, peculiar to Maps, is to pick a reasonable default contents when declaring an AVLMap. Zero might be appropriate here, so you might declare a Map,

```
PersonSSNAVLMap m((SSN)0);
```

Suppose you wanted a `VHMap` instead of an `AVLMap` Besides generating different implementations, there are two differences in how you should prepare the `` `defs' `` file. First, because a VHMap uses a C++ array internally, and because C++ array slots are initialized differently than single elements, you must ensure that class Person contains (1) a no-argument constructor, and (2) an assignment operator. You could arrange this via

```
class Person
{ ...;
    Person() {}
  void operator = (const Person& p) { nm = p.nm; addr = p.addr; }
};
```

(The lack of action in the constructor is OK here because `Strings` possess usable no-argument constructors.)

You also need to edit `` `Person.defs.h' `` to indicate a usable hash function and default capacity, via something like

```
#include <builtin.h>
#define <T>HASH(x)  (hashpjw(x.name().chars()))
#define DEFAULT_INITIAL_CAPACITY 1000
```

Since the `hashpjw` function from `` `builtin.h' `` is appropriate here. Changing the default capacity to a value expected to exceed the actual capacity helps to avoid "hidden" inefficiencies when a new VHMap is created without overriding the default, which is all too easy to do.

Otherwise, everything is the same as above, substituting `VHMap` for `AVLMap`.

Go to the previous, next section.

Go to the previous, next section.

Variable-Sized Object Representation

One of the first goals of the GNU C++ library is to enrich the kinds of basic classes that may be considered as (nearly) "built into" C++. A good deal of the inspiration for these efforts is derived from considering features of other type-rich languages, particularly Common Lisp and Scheme. The general characteristics of most class and friend operators and functions supported by these classes has been heavily influenced by such languages.

Four of these types, Strings, Integers, BitSets, and BitStrings (as well as associated and/or derived classes) require representations suitable for managing variable-sized objects on the free-store. The basic technique used for all of these is the same, although various details necessarily differ from class to class.

The general strategy for representing such objects is to create chunks of memory that include both header information (e.g., the size of the object), as well as the variable-size data (an array of some sort) at the end of the chunk. Generally the maximum size of an object is limited to something less than all of addressable memory, as a safeguard. The minimum size is also limited so as not to waste allocations expanding very small chunks. Internally, chunks are allocated in blocks well-tuned to the performance of the `new` operator.

Class elements themselves are merely pointers to these chunks. Most class operations are performed via inline "translation" functions that perform the required operation on the corresponding representation. However, constructors and assignments operate by copying entire representations, not just pointers.

No attempt is made to control temporary creation in expressions and functions involving these classes. Users of previous versions of the classes will note the disappearance of both "Tmp" classes and reference counting. These were dropped because, while they did improve performance in some cases, they obscure class mechanics, lead programmers into the false belief that they need not worry about such things, and occasionally have paradoxical behavior.

These variable-sized object classes are integrated as well as possible into C++. Most such classes possess converters that allow automatic coercion both from and to builtin basic types. (e.g., char* to and from String, long int to and from Integer, etc.). There are pro's and con's to circular converters, since they can sometimes lead to the conversion from a builtin type through to a class function and back to a builtin type without any special attention on the part of the programmer, both for better and worse.

Most of these classes also provide special-case operators and functions mixing basic with class types, as a way to avoid constructors in cases where the operations do not rely on anything special about the representations. For example, there is a special case concatenation operator for a String concatenated with a char, since building the result does not rely on anything about the String header. Again, there are arguments both for and against this approach. Supporting these cases adds a non-trivial degree of (mainly inline) function proliferation, but results in more efficient operations. Efficiency wins out over parsimony here, as part of the goal to produce classes that provide sufficient functionality and efficiency so that programmers are not tempted to try to manipulate or bypass the underlying representations.

Go to the previous, next section.

Go to the previous, next section.

Some guidelines for using expression-oriented classes

The fact that C++ allows operators to be overloaded for user-defined classes can make programming with library classes like `Integer`, `String`, and so on very convenient. However, it is worth becoming familiar with some of the inherent limitations and problems associated with such operators.

Many operators are *constructive*, i.e., create a new object based on some function of some arguments. Sometimes the creation of such objects is wasteful. Most library classes supporting expressions contain facilities that help you avoid such waste.

For example, for `Integer a, b, c; ...; c = a + b + a;`, the plus operator is called to sum a and b, creating a new temporary object as its result. This temporary is then added with a, creating another temporary, which is finally copied into c, and the temporaries are then deleted. In other words, this code might have an effect similar to `Integer a, b, c; ...; Integer t1(a); t1 += b; Integer t2(t1); t2 += a; c = t2;`.

For small objects, simple operators, and/or non-time/space critical programs, creation of temporaries is not a big problem. However, often, when fine-tuning a program, it may be a good idea to rewrite such code in a less pleasant, but more efficient manner.

For builtin types like ints, and floats, C and C++ compilers already know how to optimize such expressions to reduce the need for temporaries. Unfortunately, this is not true for C++ user defined types, for the simple (but very annoying, in this context) reason that nothing at all is guaranteed about the semantics of overloaded operators and their interrelations. For example, if the above expression just involved ints, not Integers, a compiler might internally convert the statement into something like `c = a; c += b; c+= a;` , or perhaps something even more clever. But since C++ does not know that Integer operator += has any relation to Integer operator +, A C++ compiler cannot do this kind of expression optimization itself.

In many cases, you can avoid construction of temporaries simply by using the assignment versions of operators whenever possible, since these versions create no temporaries. However, for maximum flexibility, most classes provide a set of "embedded assembly code" procedures that you can use to fully control time, space, and evaluation strategies. Most of these procedures are "three-address" procedures that take two `const` source arguments, and a destination argument. The procedures perform the appropriate actions, placing the results in the destination (which is may involve overwriting old contents). These procedures are designed to be fast and robust. In particular, aliasing is always handled correctly, so that, for example `add(x, x, x);` is perfectly OK. (The names of these procedures are listed along with the classes.)

For example, suppose you had an Integer expression `a = (b - a) * -(d / c);`

This would be compiled as if it were `Integer t1=b-a; Integer t2=d/c; Integer`


```
t3=-t2; Integer t4=t1*t3; a=t4;
```

But, with some manual cleverness, you might yourself some up with `sub(a, b, a); mul(a, c, a); div(a, c, a);`

A related phenomenon occurs when creating your own constructive functions returning instances of such types. Suppose you wanted to write function `Integer f(const Integer& a) { Integer r = a; r += a; return r; }`

This function, when called (as in `a = f(a);`) demonstrates a similar kind of wasted copy. The returned value r must be copied out of the function before it can be used by the caller. In GNU C++, there is an alternative via the use of named return values. Named return values allow you to manipulate the returned object directly, rather than requiring you to create a local inside a function and then copy it out as the returned value. In this example, this can be done via `Integer f(const Integer& a) return r(a) { r += a; return; }`

A final guideline: The overloaded operators are very convenient, and much clearer to use than procedural code. It is almost always a good idea to make it right, *then* make it fast, by translating expression code into procedural code after it is known to be correct.

Go to the previous, next section.

Go to the previous, next section.

TOC

Pseudo-indexes

Many useful classes operate as containers of elements. Techniques for accessing these elements from a container differ from class to class. In the GNU C++ library, access methods have been partially standardized across different classes via the use of pseudo-indexes called `Pixes`. A `Pix` acts in some ways like an index, and in some ways like a pointer. (Their underlying representations are just `void*` pointers). A `Pix` is a kind of "key" that is translated into an element access by the class. In virtually all cases, `Pixes` are pointers to some kind internal storage cells. The containers use these pointers to extract items.

`Pixes` support traversal and inspection of elements in a collection using analogs of array indexing. However, they are pointer-like in that `0` is treated as an invalid `Pix`, and unsafe insofar as programmers can attempt to access nonexistent elements via dangling or otherwise invalid `Pixes` without first checking for their validity.

In general it is a very bad idea to perform traversals in the the midst of destructive modifications to containers.

Typical applications might include code using the idiom

```
for (Pix i = a.first(); i != 0; a.next(i)) use(a(i));
```

for some container `a` and function `use`.

Classes supporting the use of `Pixes` always contain the following methods, assuming a container `a` of element types of `Base`.

`Pix i = a.first()`
: Set i to index the first element of a or 0 if a is empty.

`a.next(i)`
: advance i to the next element of a or 0 if there is no next element;

`Base x = a(i); a(i) = x;`
: a(i) returns a reference to the element indexed by i.

`int present = a.owns(i)`
: returns true if Pix i is a valid Pix in a. This is often a relatively slow operation, since the collection must usually traverse through elements to see if any correspond to the Pix.

Some container classes also support backwards traversal via

`Pix i = a.last()`
: Set i to the last element of a or 0 if a is empty.

`a.prev(i)`
: sets i to the previous element in a, or 0 if there is none.

Collections supporting elements with an equality operation possess

`Pix j = a.seek(x)`

sets j to the index of the first occurrence of x, or 0 if x is not contained in a.

Bag classes possess

`Pix j = a.seek(x, Pix from = 0)`

sets j to the index of the next occurrence of x following i, or 0 if x is not contained in a. If i == 0, the first occurrence is returned.

Set, Bag, and PQ classes possess

`Pix j = a.add(x) (or a.enq(x) for priority queues)`

add x to the collection, returning its Pix. The Pix of an item can change in collections where further additions and deletions involve the actual movement of elements (currently in OXPSet, OXPBag, XPPQ, VOHSet), but in all other cases, an item's Pix may be considered a permanent key to its location.

Go to the previous, next section.

Go to the previous, next section.

Header files for interfacing C++ to C

The following files are provided so that C++ programmers may invoke common C library and system calls. The names and contents of these files are subject to change in order to be compatible with the forthcoming GNU C library. Other files, not listed here, are simply C++-compatible interfaces to corresponding C library files.

`values.h'

A collection of constants defining the numbers of bits in builtin types, minimum and maximum values, and the like. Most names are the same as those found in `values.h' found on Sun systems.

`std.h'

A collection of common system calls and `libc.a' functions. Only those functions that can be declared without introducing new type definitions (socket structures, for example) are provided. Common `char*` functions (like `strcmp`) are among the declarations. All functions are declared along with their library names, so that they may be safely overloaded.

`string.h'

This file merely includes `<std.h>', where string function prototypes are declared. This is a workaround for the fact that system `string.h' and `strings.h' files often differ in contents.

`osfcn.h'

This file merely includes `<std.h>', where system function prototypes are declared.

`libc.h'

This file merely includes `<std.h>', where C library function prototypes are declared.

`math.h'

A collection of prototypes for functions usually found in libm.a, plus some `#defined` constants that appear to be consistent with those provided in the AT&T version. The value of `HUGE` should be checked before using. Declarations of all common math functions are preceded with `overload` declarations, since these are commonly overloaded.

`stdio.h'

Declaration of `FILE` (`_iobuf`), common macros (like `getc`), and function prototypes for `libc.a' functions that operate on `FILE*`'s. The value `BUFSIZ` and the declaration of `_iobuf` should be checked before using.

`assert.h'

C++ versions of assert macros.

`generic.h'

String concatenation macros useful in creating generic classes. They are similar in function to the AT&T CC versions.

`new.h'

Declarations of the default global operator new, the two-argument placement version, and associated error handlers.

Go to the previous, next section.

Go to the previous, next section.

TOC

Utility functions for built in types

Files `builtin.h' and corresponding `.cc' implementation files contain various convenient inline and non-inline utility functions. These include useful enumeration types, such as TRUE, FALSE ,the type definition for pointers to libg++ error handling functions, and the following functions.

`long abs(long x); double abs(double x);`

inline versions of abs. Note that the standard libc.a version, `int abs(int)` is *not* declared as inline.

`void clearbit(long& x, long b);`

clears the b'th bit of x (inline).

`void setbit(long& x, long b);`

sets the b'th bit of x (inline)

`int testbit(long x, long b);`

returns the b'th bit of x (inline).

`int even(long y);`

returns true if x is even (inline).

`int odd(long y);`

returns true is x is odd (inline).

`int sign(long x); int sign(double x);`

returns -1, 0, or 1, indicating whether x is less than, equal to, or greater than zero (inline).

`long gcd(long x, long y);`

returns the greatest common divisor of x and y.

`long lcm(long x, long y);`

returns the least common multiple of x and y.

`long lg(long x);`

returns the floor of the base 2 log of x.

`long pow(long x, long y); double pow(double x, long y);`

returns x to the integer power y using via the iterative O(log y) "Russian peasant" method.

`long sqr(long x); double sqr(double x);`

returns x squared (inline).

`long sqrt(long y);`

returns the floor of the square root of x.

`unsigned int hashpjw(const char* s);`

a hash function for null-terminated char* strings using the method described in Aho, Sethi, & Ullman, p 436.

`unsigned int multiplicativehash(int x);`

a hash function for integers that returns the lower bits of multiplying x by the golden ratio times pow(2, 32). See Knuth, Vol 3, p 508.

`unsigned int foldhash(double x);`

a hash function for doubles that exclusive-or's the first and second words of x, returning the result as an integer.

`double start_timer()`

Starts a process timer.

`double return_elapsed_time(double last_time)`

Returns the process time since last_time. If last_time == 0 returns the time since the last start_timer. Returns -1 if start_timer was not first called.

File `` `Maxima.h' `` includes versions of `MAX, MIN` for builtin types.

File `` `compare.h' `` includes versions of `compare(x, y)` for builtin types. These return negative if the first argument is less than the second, zero for equal, and positive for greater.

Go to the previous, next section.

Go to the previous, next section.

Library dynamic allocation primitives

Libg++ contains versions of `malloc, free, realloc` that were designed to be well-tuned to C++ applications. The source file `` `malloc.c' `` contains some design and implementation details. Here are the major user-visible differences from most system malloc routines:

1. These routines *overwrite* storage of freed space. This means that it is never permissible to use a `delete`'d object in any way. Doing so will either result in trapped fatal errors or random aborts within malloc, free, or realloc.
2. The routines tend to perform well when a large number of objects of the same size are allocated and freed. You may find that it is not worth it to create your own special allocation schemes in such cases.
3. The library sets top-level `operator new()` to call malloc and `operator delete()` to call free. Of course, you may override these definitions in C++ programs by creating your own operators that will take precedence over the library versions. However, if you do so, be sure to define *both* `operator new()` and `operator delete()`.
4. These routines do *not* support the odd convention, maintained by some versions of malloc, that you may call `realloc` with a pointer that has been `free`'d.
5. The routines automatically perform simple checks on `free`'d pointers that can often determine whether users have accidentally written beyond the boundaries of allocated space, resulting in a fatal error.
6. The function `malloc_usable_size(void* p)` returns the number of bytes actually allocated for `p`. For a valid pointer (i.e., one that has been `malloc`'d or `realloc`'d but not yet `free`'d) this will return a number greater than or equal to the requested size, else it will normally return 0. Unfortunately, a non-zero return can not be an absolutely perfect indication of lack of error. If a chunk has been `free`'d but then re-allocated for a different purpose somewhere elsewhere, then `malloc_usable_size` will return non-zero. Despite this, the function can be very valuable for performing run-time consistency checks.
7. `malloc` requires 8 bytes of overhead per allocated chunk, plus a mmaximum alignment adjustment of 8 bytes. The number of bytes of usable space is exactly as requested, rounded to the nearest 8 byte boundary.
8. The routines do *not* contain any synchronization support for multiprocessing. If you perform global allocation on a shared memory multiprocessor, you should disable compilation and use of libg++ malloc in the distribution `` `Makefile' `` and use your system version of malloc.

Go to the previous, next section.

Go to the previous, next section.

The new input/output classes

The iostream classes implement most of the features of AT&T version 2.0 iostream library classes, and most of the features of the ANSI X3J16 library draft (which is based on the AT&T design). These classes are available in `libg++` for convenience and for compatibility with older releases; however, since the iostream classes are licensed under less stringent terms than `libg++`, they are now also available in a separate library called `libio`---and documented in a separate manual, corresponding to that library.

See section 'Introduction' in The GNU C++ Iostream Library.

Go to the previous, next section.

Go to the previous, next section.

The old I/O library

WARNING: This chapter describes classes that are *obsolete*. These classes are normally not available when libg++ is installed normally. The sources are currently included in the distribution, and you can configure libg++ to use these classes instead of the new iostream classes. This is only a temporary measure; you should convert your code to use iostreams as soon as possible. The iostream classes provide some compatibility support, but it is very incomplete (there is no longer a `File` class).

File-based classes

The `File` class supports basic IO on Unix files. Operations are based on common C stdio library functions.

`File` serves as the base class for istreams, ostreams, and other derived classes. It contains the interface between the Unix stdio file library and these more structured classes. Most operations are implemented as simple calls to stdio functions. `File` class operations are also fully compatible with raw system file reads and writes (like the system `read` and `lseek` calls) when buffering is disabled (see below). The `FILE*` stdio file pointer is, however maintained as protected. Classes derived from File may only use the IO operations provided by File, which encompass essentially all stdio capabilities.

The class contains four general kinds of functions: methods for binding `File`s to physical Unix files, basic IO methods, file and buffer control methods, and methods for maintaining logical and physical file status.

Binding and related tasks are accomplished via `File` constructors and destructors, and member functions `open, close, remove, filedesc, name, setname`.

If a file name is provided in a constructor or open, it is maintained as class variable `nm` and is accessible via `name`. If no name is provided, then `nm` remains null, except that `File`s bound to the default files stdin, stdout, and stderr are automatically given the names `(stdin), (stdout), (stderr)` respectively. The function `setname` may be used to change the internal name of the `File`. This does not change the name of the physical file bound to the File. The member function `close` closes a file. The `~File` destructor closes a file if it is open, except that stdin, stdout, and stderr are flushed but left open for the system to close on program exit since some systems may require this, and on others it does not matter. `remove` closes the file, and then deletes it if possible by calling the system function to delete the file with the name provided in the `nm` field.

Basic IO

- `read` and `write` perform binary IO via stdio `fread` and `fwrite`.
- `get` and `put` for chars invoke stdio `getc` and `putc` macros.

- `put(const char* s)` outputs a null-terminated string via stdio `fputs`.
- `unget` and `putback` are synonyms. Both call stdio `ungetc`.

File Control

`flush`, `seek`, `tell`, and `tell` call the corresponding stdio functions.

`flush(char)` and `fill()` call stdio `_flsbuf` and `_filbuf` respectively.

`setbuf` is mainly useful to turn off buffering in cases where nonsequential binary IO is being performed. `raw` is a synonym for `setbuf(_IONBF)`. After a `f.raw()`, using the stdio functions instead of the system `read, write`, etc., calls entails very little overhead. Moreover, these become fully compatible with intermixed system calls (e.g., `lseek(f.filedesc(), 0, 0)`). While intermixing `File` and system IO calls is not at all recommended, this technique does allow the `File` class to be used in conjunction with other functions and libraries already set up to operate on file descriptors. `setbuf` should be called at most once after a constructor or open, but before any IO.

File Status

File status is maintained in several ways.

A `File` may be checked for accessibility via `is_open()`, which returns true if the File is bound to a usable physical file, `readable()`, which returns true if the File can be read from (opened for reading, and not in a _fail state), or `writable()`, which returns true if the File can be written to.

`File` operations return their status via two means: failure and success are represented via the logical state. Also, the return values of invoked stdio and system functions that return useful numeric values (not just failure/success flags) are held in a class variable accessible via `iocount`. (This is useful, for example, in determining the number of items actually read by the `read` function.)

Like the AT&T i/o-stream classes, but unlike the description in the Stroustrup book, p238, `rdstate()` returns the bitwise OR of `_eof`, `_fail` and `_bad`, not necessarily distinct values. The functions `eof()`, `fail()`, `bad()`, and `good()` can be used to test for each of these conditions independently.

`_fail` becomes set for any input operation that could not read in the desired data, and for other failed operations. As with all Unix IO, `_eof` becomes true only when an input operations fails because of an end of file. Therefore, `_eof` is not immediately true after the last successful read of a file, but only after one final read attempt. Thus, for input operations, `_fail` and `_eof` almost always become true at the same time. `bad` is set for unbound files, and may also be set by applications in order to communicate input corruption. Conversely, `_good` is defined as 0 and is returned by `rdstate()` if all is well.

The state may be modified via `clear(flag)`, which, despite its name, sets the corresponding state_value flag. `clear()` with no arguments resets the state to `_good`. `failif(int cond)` sets the state to `_fail` only if `cond` is true.

Errors occuring during constructors and file opens also invoke the function `error`. `error` in turn calls a resetable error handling function pointed to by the non-member global variable

`File_error_handler` only if a system error has been generated. Since `error` cannot tell if the current system error is actually responsible for a failure, it may at times print out spurious messages. Three error handlers are provided. The default, `verbose_File_error_handler` calls the system function `perror` to print the corresponding error message on standard error, and then returns to the caller. `quiet_File_error_handler` does nothing, and simply returns. `fatal_File_error_handler` prints the error and then aborts execution. These three handlers, or any other user-defined error handlers can be selected via the non-member function `set_File_error_handler`.

All read and write operations communicate either logical or physical failure by setting the `_fail` flag. All further operations are blocked if the state is in a `_fail` or`_bad` condition. Programmers must explicitly use `clear()` to reset the state in order to continue IO processing after either a logical or physical failure. C programmers who are unfamiliar with these conventions should note that, unlike the stdio library, `File` functions indicate IO success, status, or failure solely through the state, not via return values of the functions. The `void*` operator or `rdstate()` may be used to test success. In particular, according to c++ conversion rules, the `void*` coercion is automatically applied whenever the `File&` return value of any `File` function is tested in an `if` or `while`. Thus, for example, an easy way to copy all of stdin to stdout until eof (at which point `get` fails) or some error is `char c; while(cin.get(c) && cout.put(c));`.

The current version of istreams and ostreams differs significantly from previous versions in order to obtain compatibility with AT&T 1.2 streams. Most code using previous versions should still work. However, the following features of `File` are not incorporated in streams (they are still present in `File`): `scan(const char* fmt...), remove(), read(), write(), setbuf(), raw()`. Additionally, the feature of previous streams that allowed free intermixing of stream and stdio input and output is no longer guaranteed to always behave as desired.

Go to the previous, next section.

Go to the previous, next section.

The Obstack class

The `Obstack` class is a simple rewrite of the C obstack macros and functions provided in the GNU CC compiler source distribution.

Obstacks provide a simple method of creating and maintaining a string table, optimized for the very frequent task of building strings character-by-character, and sometimes keeping them, and sometimes not. They seem especially useful in any parsing application. One of the test files demonstrates usage.

A brief summary:

`grow`
: places something on the obstack without committing to wrap it up as a single entity yet.

`finish`
: wraps up a constructed object as a single entity, and returns the pointer to its start address.

`copy`
: places things on the obstack, and *does* wrap them up. `copy` is always equivalent to first grow, then finish.

`free`
: deletes something, and anything else put on the obstack since its creation.

The other functions are less commonly needed:

`blank`
: is like grow, except it just grows the space by size units without placing anything into this space

`alloc`
: is like `blank`, but it wraps up the object and returns its starting address.

`chunk_size, base, next_free, alignment_mask, size, room`
: returns the appropriate class variables.

`grow_fast`
: places a character on the obstack without checking if there is enough room.

`blank_fast`
: like `blank`, but without checking if there is enough room.

`shrink(int n)`
: shrink the current chunk by n bytes.

`contains(void* addr)`
: returns true if the Obstack holds the address addr.

Here is a lightly edited version of the original C documentation:

These functions operate a stack of objects. Each object starts life small, and may grow to maturity. (Consider building a word syllable by syllable.) An object can move while it is growing. Once it has "finished" it never changes address again. So the "top of the stack" is typically an immature growing object, while the rest of the stack is of mature, fixed size and fixed address objects.

These routines grab large chunks of memory, using the GNU C++ `new` operator. On occasion, they free chunks, via `delete`.

Each independent stack is represented by a Obstack.

One motivation for this package is the problem of growing char strings in symbol tables. Unless you are a "fascist pig with a read-only mind" [Gosper's immortal quote from HAKMEM item 154, out of context] you would not like to put any arbitrary upper limit on the length of your symbols.

In practice this often means you will build many short symbols and a few long symbols. At the time you are reading a symbol you don't know how long it is. One traditional method is to read a symbol into a buffer, `realloc()`ating the buffer every time you try to read a symbol that is longer than the buffer. This is beaut, but you still will want to copy the symbol from the buffer to a more permanent symbol-table entry say about half the time.

With obstacks, you can work differently. Use one obstack for all symbol names. As you read a symbol, grow the name in the obstack gradually. When the name is complete, finalize it. Then, if the symbol exists already, free the newly read name.

The way we do this is to take a large chunk, allocating memory from low addresses. When you want to build a symbol in the chunk you just add chars above the current "high water mark" in the chunk. When you have finished adding chars, because you got to the end of the symbol, you know how long the chars are, and you can create a new object. Mostly the chars will not burst over the highest address of the chunk, because you would typically expect a chunk to be (say) 100 times as long as an average object.

In case that isn't clear, when we have enough chars to make up the object, *they are already contiguous in the chunk* (guaranteed) so we just point to it where it lies. No moving of chars is needed and this is the second win: potentially long strings need never be explicitly shuffled. Once an object is formed, it does not change its address during its lifetime.

When the chars burst over a chunk boundary, we allocate a larger chunk, and then copy the partly formed object from the end of the old chunk to the beginning of the new larger chunk. We then carry on accreting characters to the end of the object as we normally would.

A special version of grow is provided to add a single char at a time to a growing object.

Summary:

- We allocate large chunks.
- We carve out one object at a time from the current chunk.
- Once carved, an object never moves.
- We are free to append data of any size to the currently growing object.
- Exactly one object is growing in an obstack at any one time.
- You can run one obstack per control block.

- You may have as many control blocks as you dare.
- Because of the way we do it, you can `unwind' a obstack back to a previous state. (You may remove objects much as you would with a stack.)

The obstack data structure is used in many places in the GNU C++ compiler.

Differences from the the GNU C version

1. The obvious differences stemming from the use of classes and inline functions instead of structs and macros. The C `init` and `begin` macros are replaced by constructors.
2. Overloaded function names are used for grow (and others), rather than the C `grow`, `grow0`, etc.
3. All dynamic allocation uses the the built-in `new` operator. This restricts flexibility by a little, but maintains compatibility with usual C++ conventions.
4. There are now two versions of finish:
    1. finish() behaves like the C version.
    2. finish(char terminator) adds `terminator`, and then calls `finish()`. This enables the normal invocation of `finish(0)` to wrap up a string being grown character-by-character.
5. There are special versions of grow(const char* s) and copy(const char* s) that add the null-terminated string `s` after computing its length.
6. The shrink and contains functions are provided.

Go to the previous, next section.

Go to the previous, next section.

The AllocRing class

An AllocRing is a bounded ring (circular list), each of whose elements contains a pointer to some space allocated via `new char[some_size]`. The entries are used cyclicly. The size, n, of the ring is fixed at construction. After that, every nth use of the ring will reuse (or reallocate) the same space. AllocRings are needed in order to temporarily hold chunks of space that are needed transiently, but across constructor-destructor scopes. They mainly useful for storing strings containing formatted characters to print across various functions and coercions. These strings are needed across routines, so may not be deleted in any one of them, but should be recovered at some point. In other words, an AllocRing is an extremely simple minded garbage collection mechanism. The GNU C++ library used to use one AllocRing for such formatting purposes, but it is being phased out, and is now only used by obsolete functions. These days, AllocRings are probably not very useful.

Support includes:

`AllocRing a(int n)`

constructs an Alloc ring with n entries, all null.

`void* mem = a.alloc(sz)`

moves the ring pointer to the next entry, and reuses the space if their is enough, also allocates space via new char[sz].

`int present = a.contains(void* ptr)`

returns true if ptr is held in one of the ring entries.

`a.clear()`

deletes all space pointed to in any entry. This is called automatically upon destruction.

`a.free(void* ptr)`

If ptr is one of the entries, calls delete of the pointer, and resets to entry pointer to null.

Go to the previous, next section.

Go to the previous, next section.

The String class

The `String` class is designed to extend GNU C++ to support string processing capabilities similar to those in languages like Awk. The class provides facilities that ought to be convenient and efficient enough to be useful replacements for `char*` based processing via the C string library (i.e., `strcpy, strcmp,` etc.) in many applications. Many details about String representations are described in the Representation section.

A separate `SubString` class supports substring extraction and modification operations. This is implemented in a way that user programs never directly construct or represent substrings, which are only used indirectly via String operations.

Another separate class, `Regex` is also used indirectly via String operations in support of regular expression searching, matching, and the like. The Regex class is based entirely on the GNU Emacs regex functions. See section 'Syntax of Regular Expressions' in GNU Emacs Manual, for a full explanation of regular expression syntax. (For implementation details, see the internal documentation in files `` `regex.h' `` and `` `regex.c' ``.)

Constructors

Strings are initialized and assigned as in the following examples:

`String x; String y = 0; String z = "";`

Set x, y, and z to the nil string. Note that either 0 or "" may always be used to refer to the nil string.

`String x = "Hello"; String y("Hello");`

Set x and y to a copy of the string "Hello".

`String x = 'A'; String y('A');`

Set x and y to the string value "A"

`String u = x; String v(x);`

Set u and v to the same string as String x

`String u = x.at(1,4); String v(x.at(1,4));`

Set u and v to the length 4 substring of x starting at position 1 (counting indexes from 0).

`String x("abc", 2);`

Sets x to "ab", i.e., the first 2 characters of "abc".

`String x = dec(20);`

Sets x to "20". As here, Strings may be initialized or assigned the results of any `char*` function.

There are no directly accessible forms for declaring SubString variables.

The declaration `Regex r("[a-zA-Z_][a-zA-Z0-9_]*");` creates a compiled regular expression

suitable for use in String operations described below. (In this case, one that matches any C++ identi
The first argument may also be a String. Be careful in distinguishing the role of backslashes in quote
GNU C++ char* constants versus those in Regexes. For example, a Regex that matches either one or
more tabs or all strings beginning with "ba" and ending with any number of occurrences of "na" could be
declared as `Regex r = "\\(\t+\\)\\|\\(ba\\(na\\)*\\)"` Note that only one backslash is
needed to signify the tab, but two are needed for the parenthesization and virgule, since the GNU C++
lexical analyzer decodes and strips backslashes before they are seen by Regex.

There are three additional optional arguments to the Regex constructor that are less commonly useful:

`fast (default 0)`

> `fast` may be set to true (1) if the Regex should be "fast-compiled". This causes an additional compilation step that is generally worthwhile if the Regex will be used many times.

`bufsize (default max(40, length of the string))`

> This is an estimate of the size of the internal compiled expression. Set it to a larger value if you know that the expression will require a lot of space. If you do not know, do not worry: realloc is used if necessary.

`transtable (default none == 0)`

> The address of a byte translation table (a char[256]) that translates each character before matching.

As a convenience, several Regexes are predefined and usable in any program. Here are their declarations from `` `String.h' ``.

```
extern Regex RXwhite;      // = "[ \n\t]+"
extern Regex RXint;        // = "-?[0-9]+"
extern Regex RXdouble;     // = "-?\\(\\([0-9]+\\.[0-9]*\\)\\|
                           //    \\([0-9]+\\)\\|
                           //    \\(\\.[0-9]+\\)\\)
                           //    \\([eE][---+]?[0-9]+\\)?"
extern Regex RXalpha;      // = "[A-Za-z]+"
extern Regex RXlowercase;  // = "[a-z]+"
extern Regex RXuppercase;  // = "[A-Z]+"
extern Regex RXalphanum;   // = "[0-9A-Za-z]+"
extern Regex RXidentifier; // = "[A-Za-z_][A-Za-z0-9_]*"
```

Examples

Most `String` class capabilities are best shown via example. The examples below use the following declarations.

```
String x = "Hello";
String y = "world";
String n = "123";
```


```
String z;
char*  s = ",";
String lft, mid, rgt;
Regex  r = "e[a-z]*o";
Regex  r2("/[a-z]*/");
char   c;
int    i, pos, len;
double f;
String words[10];
words[0] = "a";
words[1] = "b";
words[2] = "c";
```

Comparing, Searching and Matching

The usual lexicographic relational operators (`==, !=, <, <=, >, >=`) are defined. A functional form `compare(String, String)` is also provided, as is `fcompare(String, String)`, which compares Strings without regard for upper vs. lower case.

All other matching and searching operations are based on some form of the (non-public) `match` and `search` functions. `match` and `search` differ in that `match` attempts to match only at the given starting position, while `search` starts at the position, and then proceeds left or right looking for a match. As seen in the following examples, the second optional `startpos` argument to functions using `match` and `search` specifies the starting position of the search: If non-negative, it results in a left-to-right search starting at position `startpos`, and if negative, a right-to-left search starting at position `x.length() + startpos`. In all cases, the index returned is that of the beginning of the match, or -1 if there is no match.

Three String functions serve as front ends to `search` and `match`. `index` performs a search, returning the index, `matches` performs a match, returning nonzero (actually, the length of the match) on success, and `contains` is a boolean function performing either a search or match, depending on whether an index argument is provided:

`x.index("lo")`

> returns the zero-based index of the leftmost occurrence of substring "lo" (3, in this case). The argument may be a String, SubString, char, char*, or Regex.

`x.index("l", 2)`

> returns the index of the first of the leftmost occurrence of "l" found starting the search at position x[2], or 2 in this case.

`x.index("l", -1)`

> returns the index of the rightmost occurrence of "l", or 3 here.

`x.index("l", -3)`

> returns the index of the rightmost occurrence of "l" found by starting the search at the 3rd to the

last position of x, returning 2 in this case.

`pos = r.search("leo", 3, len, 0)`

returns the index of r in the `char*` string of length 3, starting at position 0, also placing the length of the match in reference parameter len.

`x.contains("He")`

returns nonzero if the String x contains the substring "He". The argument may be a String, SubString, char, char*, or Regex.

`x.contains("el", 1)`

returns nonzero if x contains the substring "el" at position 1. As in this example, the second argument to `contains`, if present, means to match the substring only at that position, and not to search elsewhere in the string.

`x.contains(RXwhite);`

returns nonzero if x contains any whitespace (space, tab, or newline). Recall that `RXwhite` is a global whitespace Regex.

`x.matches("lo", 3)`

returns nonzero if x starting at position 3 exactly matches "lo", with no trailing characters (as it does in this example).

`x.matches(r)`

returns nonzero if String x as a whole matches Regex r.

`int f = x.freq("l")`

returns the number of distinct, nonoverlapping matches to the argument (2 in this case).

Substring extraction

Substrings may be extracted via the `at`, `before`, `through`, `from`, and `after` functions. These behave as either lvalues or rvalues.

`z = x.at(2, 3)`

sets String z to be equal to the length 3 substring of String x starting at zero-based position 2, setting z to "llo" in this case. A nil String is returned if the arguments don't make sense.

`x.at(2, 2) = "r"`

Sets what was in positions 2 to 3 of x to "r", setting x to "Hero" in this case. As indicated here, SubString assignments may be of different lengths.

`x.at("He") = "je";`

x("He") is the substring of x that matches the first occurrence of it's argument. The substitution sets x to "jello". If "He" did not occur, the substring would be nil, and the assignment would have no effect.

`x.at("l", -1) = "i";`

replaces the rightmost occurrence of "l" with "i", setting x to "Helio".

`z = x.at(r)`

sets String z to the first match in x of Regex r, or "ello" in this case. A nil String is returned if there is no match.

`z = x.before("o")`

sets z to the part of x to the left of the first occurrence of "o", or "Hell" in this case. The argument may also be a String, SubString, or Regex. (If there is no match, z is set to "".)

`x.before("ll") = "Bri";`

sets the part of x to the left of "ll" to "Bri", setting x to "Brillo".

`z = x.before(2)`

sets z to the part of x to the left of x[2], or "He" in this case.

`z = x.after("Hel")`

sets z to the part of x to the right of "Hel", or "lo" in this case.

`z = x.through("el")`

sets z to the part of x up and including "el", or "Hel" in this case.

`z = x.from("el")`

sets z to the part of x from "el" to the end, or "ello" in this case.

`x.after("Hel") = "p";`

sets x to "Help";

`z = x.after(3)`

sets z to the part of x to the right of x[3] or "o" in this case.

`z = " ab c"; z = z.after(RXwhite)`

sets z to the part of its old string to the right of the first group of whitespace, setting z to "ab c"; Use gsub(below) to strip out multiple occurrences of whitespace or any pattern.

`x[0] = 'J';`

sets the first element of x to 'J'. x[i] returns a reference to the ith element of x, or triggers an error if i is out of range.

`common_prefix(x, "Help")`

returns the String containing the common prefix of the two Strings or "Hel" in this case.

`common_suffix(x, "to")`

returns the String containing the common suffix of the two Strings or "o" in this case.

Concatenation

`z = x + s + ' ' + y.at("w") + y.after("w") + ".";`

sets z to "Hello, world."

`x += y;`

sets x to "Helloworld"

`cat(x, y, z)`

A faster way to say z = x + y.

`cat(z, y, x, x)`

Double concatenation; A faster way to say x = z + y + x.

`y.prepend(x);`

A faster way to say y = x + y.

`z = replicate(x, 3);`

sets z to "HelloHelloHello".

`z = join(words, 3, "/")`

sets z to the concatenation of the first 3 Strings in String array words, each separated by "/", setting z to "a/b/c" in this case. The last argument may be "" or 0, indicating no separation.

Other manipulations

`z = "this string has five words"; i = split(z, words, 10, RXwhite);`

sets up to 10 elements of String array words to the parts of z separated by whitespace, and returns the number of parts actually encountered (5 in this case). Here, words[0] = "this", words[1] = "string", etc. The last argument may be any of the usual. If there is no match, all of z ends up in words[0]. The words array is *not* dynamically created by split.

`int nmatches x.gsub("l","ll")`

substitutes all original occurrences of "l" with "ll", setting x to "Hellllo". The first argument may be any of the usual, including Regex. If the second argument is "" or 0, all occurrences are deleted. gsub returns the number of matches that were replaced.

`z = x + y; z.del("loworl");`

deletes the leftmost occurrence of "loworl" in z, setting z to "Held".

`z = reverse(x)`

sets z to the reverse of x, or "olleH".

`z = upcase(x)`

sets z to x, with all letters set to uppercase, setting z to "HELLO"

`z = downcase(x)`

sets z to x, with all letters set to lowercase, setting z to "hello"

`z = capitalize(x)`

sets z to x, with the first letter of each word set to uppercase, and all others to lowercase, setting z to "Hello"

`x.reverse(), x.upcase(), x.downcase(), x.capitalize()`

in-place, self-modifying versions of the above.

Reading, Writing and Conversion

`cout << x`

writes out x.

`cout << x.at(2, 3)`

writes out the substring "llo".

`cin >> x`

reads a whitespace-bounded string into x.

`x.length()`

returns the length of String x (5, in this case).

`s = (const char*)x`

can be used to extract the `char*` char array. This coercion is useful for sending a String as an argument to any function expecting a `const char*` argument (like `atoi`, and `File::open`). This operator must be used with care, since the conversion returns a pointer to `String` internals without copying the characters: The resulting `(char*)` is only valid until the next String operation, and you must not modify it. (The conversion is defined to return a const value so that GNU C++ will produce warning and/or error messages if changes are attempted.)

Go to the previous, next section.

Go to the previous, next section.

The Integer class.

The `Integer` class provides multiple precision integer arithmetic facilities. Some representation details are discussed in the Representation section.

`Integers` may be up to `b * ((1 << b) - 1)` bits long, where `b` is the number of bits per short (typically 1048560 bits when `b = 16`). The implementation assumes that a `long` is at least twice as long as a `short`. This assumption hides beneath almost all primitive operations, and would be very difficult to change. It also relies on correct behavior of *unsigned* arithmetic operations.

Some of the arithmetic algorithms are very loosely based on those provided in the MIT Scheme `bignum.c' release, which is Copyright (c) 1987 Massachusetts Institute of Technology. Their use here falls within the provisions described in the Scheme release.

Integers may be constructed in the following ways:

`Integer x;`

Declares an uninitialized Integer.

`Integer x = 2; Integer y(2);`

Set x and y to the Integer value 2;

`Integer u(x); Integer v = x;`

Set u and v to the same value as x.

Method: long **Integer::as_long()** *const*

Used to coerce an `Integer` back into longs via the `long` coercion operator. If the Integer cannot fit into a long, this returns MINLONG or MAXLONG (depending on the sign) where MINLONG is the most negative, and MAXLONG is the most positive representable long.

Method: int **Integer::fits_in_long()** *const*

Returns true iff the `Integer` is `< MAXLONG` and `> MINLONG`.

Method: double **Integer::as_double()** *const*

Coerce the `Integer` to a `double`, with potential loss of precision. `+/-HUGE` is returned if the Integer cannot fit into a double.

Method: int **Integer::fits_in_double()** *const*

Returns true iff the `Integer` can fit into a double.

All of the usual arithmetic operators are provided (`+, -, *, /, %, +=, ++, -=, --, *=, /=, %=, ==, !=, <, <=, >, >=`). All operators support special versions for mixed arguments of Integers and regular C++ longs in order to avoid useless coercions, as well as to allow automatic promotion of shorts and ints to longs, so that they may be applied without additional Integer coercion

operators. The only operators that behave differently than the corresponding int or long operators are `++` and `--`. Because C++ does not distinguish prefix from postfix application, these are declared as `void` operators, so that no confusion can result from applying them as postfix. Thus, for Integers x and y, `++x; y = x;` is correct, but `y = ++x;` and `y = x++;` are not.

Bitwise operators (~, &, |, ^, <<, >>, &=, |=, ^=, <<=, >>=) are also provided. However, these operate on sign-magnitude, rather than two's complement representations. The sign of the result is arbitrarily taken as the sign of the first argument. For example, `Integer(-3) & Integer(5)` returns `Integer(-1)`, not -3, as it would using two's complement. Also, ~, the complement operator, complements only those bits needed for the representation. Bit operators are also provided in the BitSet and BitString classes. One of these classes should be used instead of Integers when the results of bit manipulations are not interpreted numerically.

The following utility functions are also provided. (All arguments are Integers unless otherwise noted).

Function: void **divide(const** *Integer& x, const Integer& y, Integer& q, Integer& r)*

Sets q to the quotient and r to the remainder of x and y. (q and r are returned by reference).

Function: Integer **pow(const** *Integer& x, const Integer& p)*

Returns x raised to the power p.

Function: Integer **lpow(long** *x, long p)*

Returns x raised to the power p.

Function: Integer **gcd(const** *Integer& x, const Integer& p)*

Returns the greatest common divisor of x and y.

Function: Integer **lcm(const** *Integer& x, const Integer& p)*

Returns the least common multiple of x and y.

Function: Integer **abs(const** *Integer& x*

Returns the absolute value of x.

Method: void **Integer::negate()**

Negates `this` in place.

`Integer sqr(x)`

returns x * x;

`Integer sqrt(x)`

returns the floor of the square root of x.

`long lg(x);`

returns the floor of the base 2 logarithm of abs(x)

`int sign(x)`

returns -1 if x is negative, 0 if zero, else +1. Using `if (sign(x) == 0)` is a generally fast method of testing for zero than using relational operators.

`int even(x)`

returns true if x is an even number

`int odd(x)`

returns true if x is an odd number.

`void setbit(Integer& x, long b)`

sets the b'th bit (counting right-to-left from zero) of x to 1.

`void clearbit(Integer& x, long b)`

sets the b'th bit of x to 0.

`int testbit(Integer x, long b)`

returns true if the b'th bit of x is 1.

`Integer atoI(char* asciinumber, int base = 10);`

converts the base base char* string into its Integer form.

`void Integer::printon(ostream& s, int base = 10, int width = 0);`

prints the ascii string value of `(*this)` as a base `base` number, in field width at least `width`.

`ostream << x;`

prints x in base ten format.

`istream >> x;`

reads x as a base ten number.

`int compare(Integer x, Integer y)`

returns a negative number if x<y, zero if x==y, or positive if x>y.

`int ucompare(Integer x, Integer y)`

like compare, but performs unsigned comparison.

`add(x, y, z)`

A faster way to say z = x + y.

`sub(x, y, z)`

A faster way to say z = x - y.

`mul(x, y, z)`

A faster way to say z = x * y.

`div(x, y, z)`

A faster way to say z = x / y.

`mod(x, y, z)`

A faster way to say z = x % y.

`and(x, y, z)`

A faster way to say z = x & y.

`or(x, y, z)`

A faster way to say z = x | y.

`xor(x, y, z)`

A faster way to say z = x ^ y.

`lshift(x, y, z)`

A faster way to say z = x << y.

`rshift(x, y, z)`

A faster way to say z = x >> y.

`pow(x, y, z)`

A faster way to say z = pow(x, y).

`complement(x, z)`

A faster way to say z = ~x.

`negate(x, z)`

A faster way to say z = -x.

Go to the previous, next section.

Go to the previous, next section.

The Rational Class

Class `Rational` provides multiple precision rational number arithmetic. All rationals are maintained in simplest form (i.e., with the numerator and denominator relatively prime, and with the denominator strictly positive). Rational arithmetic and relational operators are provided (`+, -, *, /, +=, -=, *=, /=, ==, !=, <, <=, >, >=`). Operations resulting in a rational number with zero denominator trigger an exception.

Rationals may be constructed and used in the following ways:

`Rational x;`

Declares an uninitialized Rational.

`Rational x = 2; Rational y(2);`

Set x and y to the Rational value 2/1;

`Rational x(2, 3);`

Sets x to the Rational value 2/3;

`Rational x = 1.2;`

Sets x to a Rational value close to 1.2. Any double precision value may be used to construct a Rational. The Rational will possess exactly as much precision as the double. Double values that do not have precise floating point equivalents (like 1.2) produce similarly imprecise rational values.

`Rational x(Integer(123), Integer(4567));`

Sets x to the Rational value 123/4567.

`Rational u(x); Rational v = x;`

Set u and v to the same value as x.

`double(Rational x)`

A Rational may be coerced to a double with potential loss of precision. +/-HUGE is returned if it will not fit.

`Rational abs(x)`

returns the absolute value of x.

`void x.negate()`

negates x.

`void x.invert()`

sets x to 1/x.

`int sign(x)`

returns 0 if x is zero, 1 if positive, and -1 if negative.

`Rational sqr(x)`

returns x * x.

`Rational pow(x, Integer y)`
: returns x to the y power.

`Integer x.numerator()`
: returns the numerator.

`Integer x.denominator()`
: returns the denominator.

`Integer floor(x)`
: returns the greatest Integer less than x.

`Integer ceil(x)`
: returns the least Integer greater than x.

`Integer trunc(x)`
: returns the Integer part of x.

`Integer round(x)`
: returns the nearest Integer to x.

`int compare(x, y)`
: returns a negative, zero, or positive number signifying whether x is less than, equal to, or greater than y.

`ostream << x;`
: prints x in the form num/den, or just num if the denominator is one.

`istream >> x;`
: reads x in the form num/den, or just num in which case the denominator is set to one.

`add(x, y, z)`
: A faster way to say z = x + y.

`sub(x, y, z)`
: A faster way to say z = x - y.

`mul(x, y, z)`
: A faster way to say z = x * y.

`div(x, y, z)`
: A faster way to say z = x / y.

`pow(x, y, z)`
: A faster way to say z = pow(x, y).

`negate(x, z)`
: A faster way to say z = -x.

Go to the previous, next section.

Go to the previous, next section.

The Complex class.

Class `Complex` is implemented in a way similar to that described by Stroustrup. In keeping with libg++ conventions, the class is named `Complex`, not `complex`. Complex arithmetic and relational operators are provided (`+, -, *, /, +=, -=, *=, /=, ==, !=`). Attempted division by (0, 0) triggers an exception.

Complex numbers may be constructed and used in the following ways:

`Complex x;`

Declares an uninitialized Complex.

`Complex x = 2; Complex y(2.0);`

Set x and y to the Complex value (2.0, 0.0);

`Complex x(2, 3);`

Sets x to the Complex value (2, 3);

`Complex u(x); Complex v = x;`

Set u and v to the same value as x.

`double real(Complex& x);`

returns the real part of x.

`double imag(Complex& x);`

returns the imaginary part of x.

`double abs(Complex& x);`

returns the magnitude of x.

`double norm(Complex& x);`

returns the square of the magnitude of x.

`double arg(Complex& x);`

returns the argument (amplitude) of x.

`Complex polar(double r, double t = 0.0);`

returns a Complex with abs of r and arg of t.

`Complex conj(Complex& x);`

returns the complex conjugate of x.

`Complex cos(Complex& x);`

returns the complex cosine of x.

`Complex sin(Complex& x);`

returns the complex sine of x.

`Complex cosh(Complex& x);`

returns the complex hyperbolic cosine of x.

```
Complex sinh(Complex& x);
```

returns the complex hyperbolic sine of x.

```
Complex exp(Complex& x);
```

returns the exponential of x.

```
Complex log(Complex& x);
```

returns the natural log of x.

```
Complex pow(Complex& x, long p);
```

returns x raised to the p power.

```
Complex pow(Complex& x, Complex& p);
```

returns x raised to the p power.

```
Complex sqrt(Complex& x);
```

returns the square root of x.

```
ostream << x;
```

prints x in the form (re, im).

```
istream >> x;
```

reads x in the form (re, im), or just (re) or re in which case the imaginary part is set to zero.

Go to the previous, next section.

Go to the previous, next section.

Fixed precision numbers

Classes `Fix16`, `Fix24`, `Fix32`, and `Fix48` support operations on 16, 24, 32, or 48 bit quantities that are considered as real numbers in the range [-1, +1). Such numbers are often encountered in digital signal processing applications. The classes may be be used in isolation or together. Class `Fix32` operations are entirely self-contained. Class `Fix16` operations are self-contained except that the multiplication operation `Fix16 * Fix16` returns a `Fix32`. `Fix24` and `Fix48` are similarly related.

The standard arithmetic and relational operations are supported (`=`, `+`, `-`, `*`, `/`, `<<`, `>>`, `+=`, `-=`, `*=`, `/=`, `<<=`, `>>=`, `==`, `!=`, `<`, `<=`, `>`, `>=`). All operations include provisions for special handling in cases where the result exceeds +/- 1.0. There are two cases that may be handled separately: "overflow" where the results of addition and subtraction operations go out of range, and all other "range errors" in which resulting values go off-scale (as with division operations, and assignment or initialization with off-scale values). In signal processing applications, it is often useful to handle these two cases differently. Handlers take one argument, a reference to the integer mantissa of the offending value, which may then be manipulated. In cases of overflow, this value is the result of the (integer) arithmetic computation on the mantissa; in others it is a fully saturated (i.e., most positive or most negative) value. Handling may be reset to any of several provided functions or any other user-defined function via `set_overflow_handler` and `set_range_error_handler`. The provided functions for `Fix16` are as follows (corresponding functions are also supported for the others).

`Fix16_overflow_saturate`

> The default overflow handler. Results are "saturated": positive results are set to the largest representable value (binary 0.111111...), and negative values to -1.0.

`Fix16_ignore`

> Performs no action. For overflow, this will allow addition and subtraction operations to "wrap around" in the same manner as integer arithmetic, and for saturation, will leave values saturated.

`Fix16_overflow_warning_saturate`

> Prints a warning message on standard error, then saturates the results.

`Fix16_warning`

> The default range_error handler. Prints a warning message on standard error; otherwise leaving the argument unmodified.

`Fix16_abort`

> prints an error message on standard error, then aborts execution.

In addition to arithmetic operations, the following are provided:

`Fix16 a = 0.5;`

> Constructs fixed precision objects from double precision values. Attempting to initialize to a value outside the range invokes the range_error handler, except, as a convenience, initialization to 1.0 sets the variable to the most positive representable value (binary 0.1111111...) without invoking the handler.

`short& mantissa(a); long& mantissa(b);`

return a * pow(2, 15) or b * pow(2, 31) as an integer. These are returned by reference, to enable "manual" data manipulation.

`double value(a); double value(b);`

return a or b as floating point numbers.

Go to the previous, next section.

Go to the previous, next section.

Classes for Bit manipulation

libg++ provides several different classes supporting the use and manipulation of collections of bits in different ways.

- Class `Integer` provides "integer" semantics. It supports manipulation of bits in ways that are often useful when treating bit arrays as numerical (integer) quantities. This class is described elsewhere.
- Class `BitSet` provides "set" semantics. It supports operations useful when treating collections of bits as representing potentially infinite sets of integers.
- Class `BitSet32` supports fixed-length BitSets holding exactly 32 bits.
- Class `BitSet256` supports fixed-length BitSets holding exactly 256 bits.
- Class `BitString` provides "string" (or "vector") semantics. It supports operations useful when treating collections of bits as strings of zeros and ones.

These classes also differ in the following ways:

- BitSets are logically infinite. Their space is dynamically altered to adjust to the smallest number of consecutive bits actually required to represent the sets. Integers also have this property. BitStrings are logically finite, but their sizes are internally dynamically managed to maintain proper length. This means that, for example, BitStrings are concatenatable while BitSets and Integers are not.
- BitSet32 and BitSet256 have precisely the same properties as BitSets, except that they use constant fixed length bit vectors.
- While all classes support basic unary and binary operations `~, &, |, ^, -`, the semantics differ. BitSets perform bit operations that precisely mirror those for infinite sets. For example, complementing an empty BitSet returns one representing an infinite number of set bits. Operations on BitStrings and Integers operate only on those bits actually present in the representation. For BitStrings and Integers, the the & operation returns a BitString with a length equal to the minimum length of the operands, and `|, ^` return one with length of the maximum.
- Only BitStrings support substring extraction and bit pattern matching.

BitSet

BitSets are objects that contain logically infinite sets of nonnegative integers. Representational details are discussed in the Representation chapter. Because they are logically infinite, all BitSets possess a trailing, infinitely replicated 0 or 1 bit, called the "virtual bit", and indicated via 0* or 1*.

BitSet32 and BitSet256 have they same properties, except they are of fixed length, and thus have no virtual bit.

BitSets may be constructed as follows:

```
BitSet a;
```

declares an empty BitSet.

`BitSet a = atoBitSet("001000");`

sets a to the BitSet 0010*, reading left-to-right. The "0*" indicates that the set ends with an infinite number of zero (clear) bits.

`BitSet a = atoBitSet("00101*");`

sets a to the BitSet 00101*, where "1*" means that the set ends with an infinite number of one (set) bits.

`BitSet a = longtoBitSet((long)23);`

sets a to the BitSet 111010*, the binary representation of decimal 23.

`BitSet a = utoBitSet((unsigned)23);`

sets a to the BitSet 111010*, the binary representation of decimal 23.

The following functions and operators are provided (Assume the declaration of BitSets a = 0011010*, b = 101101*, throughout, as examples).

`~a`

returns the complement of a, or 1100101* in this case.

`a.complement()`

sets a to ~a.

`a & b; a &= b;`

returns a intersected with b, or 0011010*.

`a | b; a |= b;`

returns a unioned with b, or 1011111*.

`a - b; a -= b;`

returns the set difference of a and b, or 000010*.

`a ^ b; a ^= b;`

returns the symmetric difference of a and b, or 1000101*.

`a.empty()`

returns true if a is an empty set.

`a == b;`

returns true if a and b contain the same set.

`a <= b;`

returns true if a is a subset of b.

`a < b;`

returns true if a is a proper subset of b;

`a != b; a >= b; a > b;`

are the converses of the above.

`a.set(7)`

sets the 7th (counting from 0) bit of a, setting a to 001111010*

`a.clear(2)`

clears the 2nd bit bit of a, setting a to 00011110*

`a.clear()`

clears all bits of a;

`a.set()`

sets all bits of a;

`a.invert(0)`

complements the 0th bit of a, setting a to 10011110*

`a.set(0,1)`

sets the 0th through 1st bits of a, setting a to 110111110* The two-argument versions of clear and invert are similar.

`a.test(3)`

returns true if the 3rd bit of a is set.

`a.test(3, 5)`

returns true if any of bits 3 through 5 are set.

`int i = a[3]; a[3] = 0;`

The subscript operator allows bits to be inspected and changed via standard subscript semantics, using a friend class BitSetBit. The use of the subscript operator a[i] rather than a.test(i) requires somewhat greater overhead.

`a.first(1) or a.first()`

returns the index of the first set bit of a (2 in this case), or -1 if no bits are set.

`a.first(0)`

returns the index of the first clear bit of a (0 in this case), or -1 if no bits are clear.

`a.next(2, 1) or a.next(2)`

returns the index of the next bit after position 2 that is set (3 in this case) or -1. `first` and `next` may be used as iterators, as in `for (int i = a.first(); i >= 0; i = a.next(i))....`

`a.last(1)`

returns the index of the rightmost set bit, or -1 if there or no set bits or all set bits.

`a.prev(3, 0)`

returns the index of the previous clear bit before position 3.

`a.count(1)`

returns the number of set bits in a, or -1 if there are an infinite number.

`a.virtual_bit()`

returns the trailing (infinitely replicated) bit of a.

`a = atoBitSet("ababX", 'a', 'b', 'X');`

converts the char* string into a bitset, with 'a' denoting false, 'b' denoting true, and 'X' denoting infinite replication.

`a.printon(cout, '-', '.', 0)`

prints `a` to `cout` represented with `'-'` for falses, `'.'` for trues, and no replication marker.

`cout << a`

prints `a` to `cout` (representing lases by `'f'`, trues by `'t'`, and using `'*'` as the replication marker).

`diff(x, y, z)`

A faster way to say z = x - y.

`and(x, y, z)`

A faster way to say z = x & y.

`or(x, y, z)`

A faster way to say z = x | y.

`xor(x, y, z)`

A faster way to say z = x ^ y.

`complement(x, z)`

A faster way to say z = ~x.

BitString

BitStrings are objects that contain arbitrary-length strings of zeroes and ones. BitStrings possess some features that make them behave like sets, and others that behave as strings. They are useful in applications (such as signature-based algorithms) where both capabilities are needed. Representational details are discussed in the Representation chapter. Most capabilities are exact analogs of those supported in the BitSet and String classes. A BitSubString is used with substring operations along the same lines as the String SubString class. A BitPattern class is used for masked bit pattern searching.

Only a default constructor is supported. The declaration `BitString a;` initializes a to be an empty BitString. BitStrings may often be initialized via `atoBitString` and `longtoBitString`.

Set operations (`~, complement, &, &=, |, |=, -, ^, ^=`) behave just as the BitSet versions, except that there is no "virtual bit": complementing complements only those bits in the BitString, and all binary operations across unequal length BitStrings assume a virtual bit of zero. The `&` operation returns a BitString with a length equal to the minimum length of the operands, and `|, ^` return one with length of the maximum.

Set-based relational operations (`==, !=, <=, <, >=, >`) follow the same rules. A string-like lexicographic comparison function, `lcompare`, tests the lexicographic relation between two BitStrings. For example, lcompare(1100, 0101) returns 1, since the first BitString starts with 1 and the second with 0.

Individual bit setting, testing, and iterator operations (`set, clear, invert, test, first, next, last, prev`) are also like those for BitSets. BitStrings are automatically expanded when setting bits at positions greater than their current length.

The string-based capabilities are just as those for class String. BitStrings may be concatenated (`+`, searched (`index, contains, matches`), and extracted into BitSubStrings (`before, at, after`) which may be assigned and otherwise manipulated. Other string-based utility functions (`reverse, common_prefix, common_suffix`) are also provided. These have the same capabilities and descriptions as those for Strings.

String-oriented operations can also be performed with a mask via class BitPattern. BitPatterns consist of two BitStrings, a pattern and a mask. On searching and matching, bits in the pattern that correspond to 0 bits in the mask are ignored. (The mask may be shorter than the pattern, in which case trailing mask bits are assumed to be 0). The pattern and mask are both public variables, and may be individually subjected to other bit operations.

Converting to char* and printing (`(atoBitString, atoBitPattern, printon, ostream <<)`) are also as in BitSets, except that no virtual bit is used, and an 'X' in a BitPattern means that the pattern bit is masked out.

The following features are unique to BitStrings.

Assume declarations of BitString a = atoBitString("01010110") and b = atoBitSTring("1101").

`a = b + c;`

Sets a to the concatenation of b and c;

`a = b + 0; a = b + 1;`

sets a to b, appended with a zero (one).

`a += b;`

appends b to a;

`a += 0; a += 1;`

appends a zero (one) to a.

`a << 2; a <<= 2`

return a with 2 zeros prepended, setting a to 0001010110. (Note the necessary confusion of << and >> operators. For consistency with the integer versions, << shifts low bits to high, even though they are printed low bits first.)

`a >> 3; a >>= 3`

return a with the first 3 bits deleted, setting a to 10110.

`a.left_trim(0)`

deletes all 0 bits on the left of a, setting a to 1010110.

`a.right_trim(0)`

deletes all trailing 0 bits of a, setting a to 0101011.

`cat(x, y, z)`

A faster way to say z = x + y.

`diff(x, y, z)`

A faster way to say z = x - y.

`and(x, y, z)`

A faster way to say z = x & y.

`or(x, y, z)`

A faster way to say z = x | y.

`xor(x, y, z)`

A faster way to say z = x ^ y.

`lshift(x, y, z)`

A faster way to say z = x << y.

`rshift(x, y, z)`

A faster way to say z = x >> y.

`complement(x, z)`

A faster way to say z = ~x.

Go to the previous, next section.

Go to the previous, next section.

Random Number Generators and related classes

The two classes `RNG` and `Random` are used together to generate a variety of random number distributions. A distinction must be made between *random number generators*, implemented by class `RNG`, and *random number distributions*. A random number generator produces a series of randomly ordered bits. These bits can be used directly, or cast to other representations, such as a floating point value. A random number generator should produce a *uniform* distribution. A random number distribution, on the other hand, uses the randomly generated bits of a generator to produce numbers from a distribution with specific properties. Each instance of `Random` uses an instance of class `RNG` to provide the raw, uniform distribution used to produce the specific distribution. Several instances of `Random` classes can share the same instance of `RNG`, or each instance can use its own copy.

RNG

Random distributions are constructed from members of class `RNG`, the actual random number generators. The `RNG` class contains no data; it only serves to define the interface to random number generators. The `RNG::asLong` member returns an unsigned long (typically 32 bits) of random bits. Applications that require a number of random bits can use this directly. More often, these random bits are transformed to a uniform random number:

```
    //
    // Return random bits converted to either a float or a double
    //
    float asFloat();
    double asDouble();
};
```

using either `asFloat` or `asDouble`. It is intended that `asFloat` and `asDouble` return differing precisions; typically, `asDouble` will draw two random longwords and transform them into a legal `double`, while `asFloat` will draw a single longword and transform it into a legal `float`. These members are used by subclasses of the `Random` class to implement a variety of random number distributions.

ACG

Class `ACG` is a variant of a Linear Congruential Generator (Algorithm M) described in Knuth, *Art of Computer Programming, Vol III*. This result is permuted with a Fibonacci Additive Congruential Generator to get good independence between samples. This is a very high quality random number

generator, although it requires a fair amount of memory for each instance of the generator.

The `ACG::ACG` constructor takes two parameters: the seed and the size. The seed is any number to be used as an initial seed. The performance of the generator depends on having a distribution of bits through the seed. If you choose a number in the range of 0 to 31, a seed with more bits is chosen. Other values are deterministically modified to give a better distribution of bits. This provides a good random number generator while still allowing a sequence to be repeated given the same initial seed.

The `size` parameter determines the size of two tables used in the generator. The first table is used in the Additive Generator; see the algorithm in Knuth for more information. In general, this table is `size` longwords long. The default value, used in the algorithm in Knuth, gives a table of 220 bytes. The table size affects the period of the generators; smaller values give shorter periods and larger tables give longer periods. The smallest table size is 7 longwords, and the longest is 98 longwords. The `size` parameter also determines the size of the table used for the Linear Congruential Generator. This value is chosen implicitly based on the size of the Additive Congruential Generator table. It is two powers of two larger than the power of two that is larger than `size`. For example, if `size` is 7, the ACG table is 7 longwords and the LCG table is 128 longwords. Thus, the default size (55) requires 55 + 256 longwords, or 1244 bytes. The largest table requires 2440 bytes and the smallest table requires 100 bytes. Applications that require a large number of generators or applications that aren't so fussy about the quality of the generator may elect to use the `MLCG` generator.

MLCG

The `MLCG` class implements a *Multiplicative Linear Congruential Generator*. In particular, it is an implementation of the double MLCG described in *"Efficient and Portable Combined Random Number Generators"* by Pierre L'Ecuyer, appearing in *Communications of the ACM, Vol. 31. No. 6*. This generator has a fairly long period, and has been statistically analyzed to show that it gives good inter-sample independence.

The `MLCG::MLCG` constructor has two parameters, both of which are seeds for the generator. As in the `MLCG` generator, both seeds are modified to give a "better" distribution of seed digits. Thus, you can safely use values such as `0' or `1' for the seeds. The `MLCG` generator used much less state than the `ACG` generator; only two longwords (8 bytes) are needed for each generator.

Random

A random number generator may be declared by first declaring a `RNG` and then a `Random`. For example, `ACG gen(10, 20); NegativeExpntl rnd (1.0, &gen);` declares an additive congruential generator with seed 10 and table size 20, that is used to generate exponentially distributed values with mean of 1.0.

The virtual member `Random::operator()` is the common way of extracting a random number from a particular distribution. The base class, `Random` does not implement `operator()`. This is performed by each of the subclasses. Thus, given the above declaration of `rnd`, new random values may be obtained via, for example, `double next_exp_rand = rnd();` Currently, the following

subclasses are provided.

Binomial

The binomial distribution models successfully drawing items from a pool. The first parameter to the constructor, `n`, is the number of items in the pool, and the second parameter, `u`, is the probability of each item being successfully drawn. The member `asDouble` returns the number of samples drawn from the pool. Although it is not checked, it is assumed that `n>0` and `0 <= u <= 1`. The remaining members allow you to read and set the parameters.

Erlang

The `Erlang` class implements an Erlang distribution with mean `mean` and variance `variance`.

Geometric

The `Geometric` class implements a discrete geometric distribution. The first parameter to the constructor, `mean`, is the mean of the distribution. Although it is not checked, it is assumed that `0 <= mean <= 1`. `Geometric()` returns the number of uniform random samples that were drawn before the sample was larger than `mean`. This quantity is always greater than zero.

HyperGeometric

The `HyperGeometric` class implements the hypergeometric distribution. The first parameter to the constructor, `mean`, is the mean and the second, `variance`, is the variance. The remaining members allow you to inspect and change the mean and variance.

NegativeExpntl

The `NegativeExpntl` class implements the negative exponential distribution. The first parameter to the constructor is the mean. The remaining members allow you to inspect and change the mean.

Normal

The `Normal`class implements the normal distribution. The first parameter to the constructor, `mean`, is the mean and the second, `variance`, is the variance. The remaining members allow you to inspect and change the mean and variance. The `LogNormal` class is a subclass of `Normal`.

LogNormal

The `LogNormal`class implements the logarithmic normal distribution. The first parameter to the constructor, `mean`, is the mean and the second, `variance`, is the variance. The remaining members allow you to inspect and change the mean and variance. The `LogNormal` class is a subclass of `Normal`.

Poisson

The `Poisson` class implements the poisson distribution. The first parameter to the constructor is the mean. The remaining members allow you to inspect and change the mean.

DiscreteUniform

The `DiscreteUniform` class implements a uniform random variable over the closed interval ranging from `[low..high]`. The first parameter to the constructor is `low`, and the second is `high`, although the order of these may be reversed. The remaining members allow you to inspect and change `low` and `high`.

Uniform

The `Uniform` class implements a uniform random variable over the open interval ranging from `[low..high)`. The first parameter to the constructor is `low`, and the second is `high`, although the order of these may be reversed. The remaining members allow you to inspect and change `low` and `high`.

Weibull

The `Weibull` class implements a weibull distribution with parameters `alpha` and `beta`. The first parameter to the class constructor is `alpha`, and the second parameter is `beta`. The remaining members allow you to inspect and change `alpha` and `beta`.

RandomInteger

The `RandomInteger` class is *not* a subclass of Random, but a stand-alone integer-oriented class that is dependent on the RNG classes. RandomInteger returns random integers uniformly from the closed interval `[low..high]`. The first parameter to the constructor is `low`, and the second is `high`, although both are optional. The last argument is always a generator. Additional members allow you to inspect and change `low` and `high`. Random integers are generated using `asInt()` or `asLong()`. Operator syntax (`()`) is also available as a shorthand for `asLong()`. Because `RandomInteger` is often used in simulations for which uniform random integers are desired over a variety of ranges,

TOC

`asLong()` and `asInt` have `high` as an optional argument. Using this optional argument produces a single value from the new range, but does not change the default range.

Go to the previous, next section.

Go to the previous, next section.

Data Collection

Libg++ currently provides two classes for *data collection* and analysis of the collected data.

SampleStatistic

Class `SampleStatistic` provides a means of accumulating samples of `double` values and providing common sample statistics.

Assume declaration of `double x`.

`SampleStatistic a;`
: declares and initializes a.

`a.reset();`
: re-initializes a.

`a += x;`
: adds sample x.

`int n = a.samples();`
: returns the number of samples.

`x = a.mean;`
: returns the means of the samples.

`x = a.var()`
: returns the sample variance of the samples.

`x = a.stdDev()`
: returns the sample standard deviation of the samples.

`x = a.min()`
: returns the minimum encountered sample.

`x = a.max()`
: returns the maximum encountered sample.

`x = a.confidence(int p)`
: returns the p-percent (0 <= p < 100) confidence interval.

`x = a.confidence(double p)`
: returns the p-probability (0 <= p < 1) confidence interval.

SampleHistogram

Class `SampleHistogram` is a derived class of `SampleStatistic` that supports collection and display of samples in bucketed intervals. It supports the following in addition to `SampleStatisic` operations.

`SampleHistogram h(double lo, double hi, double width);`

declares and initializes h to have buckets of size width from lo to hi. If the optional argument width is not specified, 10 buckets are created. The first bucket and also holds samples less than lo, and the last one holds samples greater than hi.

`int n = h.similarSamples(x)`

returns the number of samples in the same bucket as x.

`int n = h.inBucket(int i)`

returns the number of samples in bucket i.

`int b = h.buckets()`

returns the number of buckets.

`h.printBuckets(ostream s)`

prints bucket counts on ostream s.

`double bound = h.bucketThreshold(int i)`

returns the upper bound of bucket i.

Go to the previous, next section.

Go to the previous, next section.

Curses-based classes

The `CursesWindow` class is a repackaging of standard curses library features into a class. It relies on `` `curses.h' ``.

The supplied `` `curses.h' `` is a fairly conservative declaration of curses library features, and does not include features like "screen" or X-window support. It is, for the most part, an adaptation, rather than an improvement of C-based `` `curses.h' `` files. The only substantive changes are the declarations of many functions as inline functions rather than macros, which was done solely to allow overloading.

The `CursesWindow` class encapsulates curses window functions within a class. Only those functions that control windows are included: Terminal control functions and macros like `cbreak` are not part of the class. All `CursesWindows` member functions have names identical to the corresponding curses library functions, except that the "w" prefix is generally dropped. Descriptions of these functions may be found in your local curses library documentation.

A `CursesWindow` may be declared via

`CursesWindow w(WINDOW* win)`

attaches w to the existing WINDOW* win. This is constructor is normally used only in the following special case.

`CursesWindow w(stdscr)`

attaches w to the default curses library standard screen window.

`CursesWindow w(int lines, int cols, int begin_y, int begin_x)`

attaches to an allocated curses window with the indicated size and screen position.

`CursesWindow sub(CursesWindow& w,int l,int c,int by,int bx,char ar='a')`

attaches to a subwindow of w created via the curses `subwin' command. If ar is sent as `r', the origin (by, bx) is relative to the parent window, else it is absolute.

The class maintains a static counter that is used in order to automatically call the curses library `initscr` and `endscr` functions at the proper times. These need not, and should not be called "manually".

`CursesWindows` maintain a tree of their subwindows. Upon destruction of a `CursesWindow`, all of their subwindows are also invalidated if they had not previously been destroyed.

It is possible to traverse trees of subwindows via the following member functions

`CursesWindow* w.parent()`

returns a pointer to the parent of the subwindow, or 0 if there is none.

`CursesWindow* w.child()`

returns the first child subwindow of the window, or 0 if there is none.

`CursesWindow* w.sibling()`

returns the next sibling of the subwindow, or 0 if there is none.

For example, to call some function `visit` for all subwindows of a window, you could write

```
void traverse(CursesWindow& w)
{
  visit(w);
  if (w.child() != 0) traverse(*w.child);
  if (w.sibling() != 0) traverse(*w.sibling);
}
```

Go to the previous, next section.

Go to the previous, next section.

List classes

The files `g++-include/List.hP' and `g++-include/List.ccP' provide pseudo-generic Lisp-type List classes. These lists are homogeneous lists, more similar to lists in statically typed functional languages like ML than Lisp, but support operations very similar to those found in Lisp. Any particular kind of list class may be generated via the `genclass` shell command. However, the implementation assumes that the base class supports an equality operator `==`. All equality tests use the `==` operator, and are thus equivalent to the use of `equal`, not `eq` in Lisp.

All list nodes are created dynamically, and managed via reference counts. `List` variables are actually pointers to these list nodes. Lists may also be traversed via Pixes, as described in the section describing Pixes. See section Pseudo-indexes

Supported operations are mirrored closely after those in Lisp. Generally, operations with functional forms are constructive, functional operations, while member forms (often with the same name) are sometimes procedural, possibly destructive operations.

As with Lisp, destructive operations are supported. Programmers are allowed to change head and tail fields in any fashion, creating circular structures and the like. However, again as with Lisp, some operations implicitly assume that they are operating on pure lists, and may enter infinite loops when presented with improper lists. Also, the reference-counting storage management facility may fail to reclaim unused circularly-linked nodes.

Several Lisp-like higher order functions are supported (e.g., `map`). Typedef declarations for the required functional forms are provided int the `.h' file.

For purposes of illustration, assume the specification of class `intList`. Common Lisp versions of supported operations are shown in brackets for comparison purposes.

Constructors and assignment

`intList a; [ (setq a nil) ]`

Declares a to be a nil intList.

`intList b(2); [ (setq b (cons 2 nil)) ]`

Declares b to be an intList with a head value of 2, and a nil tail.

`intList c(3, b); [ (setq c (cons 3 b)) ]`

Declares c to be an intList with a head value of 3, and b as its tail.

`b = a; [ (setq b a) ]`

Sets b to be the same list as a.

Assume the declarations of intLists a, b, and c in the following. See section Pseudo-indexes.

List status

`a.null(); OR !a; [ (null a) ]`

returns true if a is null.

`a.valid(); [ (listp a) ]`

returns true if a is non-null. Inside a conditional test, the `void*` coercion may also be used as in `if (a) ....`

`intList(); [ nil ]`

intList() may be used to null terminate a list, as in `intList f(int x) {if (x == 0) return intList(); ... }` .

`a.length(); [ (length a) ]`

returns the length of a.

`a.list_length(); [ (list-length a) ]`

returns the length of a, or -1 if a is circular.

heads and tails

`a.get(); OR a.head() [ (car a) ]`

returns a reference to the head field.

`a[2]; [ (elt a 2) ]`

returns a reference to the second (counting from zero) head field.

`a.tail(); [ (cdr a) ]`

returns the intList that is the tail of a.

`a.last(); [ (last a) ]`

returns the intList that is the last node of a.

`a.nth(2); [ (nth a 2) ]`

returns the intList that is the nth node of a.

`a.set_tail(b); [ (rplacd a b) ]`

sets a's tail to b.

`a.push(2); [ (push 2 a) ]`

equivalent to a = intList(2, a);

`int x = a.pop() [ (setq x (car a)) (pop a) ]`

returns the head of a, also setting a to its tail.

Constructive operations

`b = copy(a); [ (setq b (copy-seq a)) ]`
sets b to a copy of a.

`b = reverse(a); [ (setq b (reverse a)) ]`
Sets b to a reversed copy of a.

`c = concat(a, b); [ (setq c (concat a b)) ]`
Sets c to a concatenated copy of a and b.

`c = append(a, b); [ (setq c (append a b)) ]`
Sets c to a concatenated copy of a and b. All nodes of a are copied, with the last node pointing to b.

`b = map(f, a); [ (setq b (mapcar f a)) ]`
Sets b to a new list created by applying function f to each node of a.

`c = combine(f, a, b);`
Sets c to a new list created by applying function f to successive pairs of a and b. The resulting list has length the shorter of a and b.

`b = remove(x, a); [ (setq b (remove x a)) ]`
Sets b to a copy of a, omitting all occurrences of x.

`b = remove(f, a); [ (setq b (remove-if f a)) ]`
Sets b to a copy of a, omitting values causing function f to return true.

`b = select(f, a); [ (setq b (remove-if-not f a)) ]`
Sets b to a copy of a, omitting values causing function f to return false.

`c = merge(a, b, f); [ (setq c (merge a b f)) ]`
Sets c to a list containing the ordered elements (using the comparison function f) of the sorted lists a and b.

Destructive operations

`a.append(b); [ (rplacd (last a) b) ]`
appends b to the end of a. No new nodes are constructed.

`a.prepend(b); [ (setq a (append b a)) ]`
prepends b to the beginning of a.

`a.del(x); [ (delete x a) ]`
deletes all nodes with value x from a.

`a.del(f); [ (delete-if f a) ]`
deletes all nodes causing function f to return true.

`a.select(f); [ (delete-if-not f a) ]`
deletes all nodes causing function f to return false.

`a.reverse(); [ (nreverse a) ]`

reverses a in-place.

`a.sort(f); [ (sort a f) ]`

sorts a in-place using ordering (comparison) function f.

`a.apply(f); [ (mapc f a) ]`

Applies void function f (int x) to each element of a.

`a.subst(int old, int repl); [ (nsubst repl old a) ]`

substitutes repl for each occurrence of old in a. Note the different argument order than the Lisp version.

Other operations

`a.find(int x); [ (find x a) ]`

returns the intList at the first occurrence of x.

`a.find(b); [ (find b a) ]`

returns the intList at the first occurrence of sublist b.

`a.contains(int x); [ (member x a) ]`

returns true if a contains x.

`a.contains(b); [ (member b a) ]`

returns true if a contains sublist b.

`a.position(int x); [ (position x a) ]`

returns the zero-based index of x in a, or -1 if x does not occur.

`int x = a.reduce(f, int base); [ (reduce f a :initial-value base) ]`

Accumulates the result of applying int function f(int, int) to successive elements of a, starting with base.

Go to the previous, next section.

Go to the previous, next section.

Linked Lists

SLLists provide pseudo-generic singly linked lists. DLLists provide doubly linked lists. The lists are designed for the simple maintenance of elements in a linked structure, and do not provide the more extensive operations (or node-sharing) of class `List`. They behave similarly to the `slist` and similar classes described by Stroustrup.

All list nodes are created dynamically. Assignment is performed via copying.

Class `DLList` supports all `SLList` operations, plus additional operations described below.

For purposes of illustration, assume the specification of class `intSLList`. In addition to the operations listed here, SLLists support traversal via Pixes. See section Pseudo-indexes

`intSLList a;`
: Declares a to be an empty list.

`intSLList b = a;`
: Sets b to an element-by-element copy of a.

`a.empty()`
: returns true if a contains no elements

`a.length();`
: returns the number of elements in a.

`a.prepend(x);`
: places x at the front of the list.

`a.append(x);`
: places x at the end of the list.

`a.join(b)`
: places all nodes from b to the end of a, simultaneously destroying b.

`x = a.front()`
: returns a reference to the item stored at the head of the list, or triggers an error if the list is empty.

`a.rear()`
: returns a reference to the rear of the list, or triggers an error if the list is empty.

`x = a.remove_front()`
: deletes and returns the item stored at the head of the list.

`a.del_front()`
: deletes the first element, without returning it.

`a.clear()`
: deletes all items from the list.

`a.ins_after(Pix i, item);`

inserts item after position i. If i is null, insertion is at the front.

`a.del_after(Pix i);`

deletes the element following i. If i is 0, the first item is deleted.

Doubly linked lists

Class `DLList` supports the following additional operations, as well as backward traversal via Pixes.

`x = a.remove_rear();`

deletes and returns the item stored at the rear of the list.

`a.del_rear();`

deletes the last element, without returning it.

`a.ins_before(Pix i, x)`

inserts x before the i.

`a.del(Pix& iint dir = 1)`

deletes the item at the current position, then advances forward if dir is positive, else backward.

Go to the previous, next section.

Go to the previous, next section.

Vector classes

The files `g++-include/Vec.ccP' and `g++-include/AVec.ccP' provide pseudo-generic standard array-based vector operations. The corresponding header files are `g++-include/Vec.hP' and `g++-include/AVec.hP'. Class `Vec` provides operations suitable for any base class that includes an equality operator. Subclass `AVec` provides additional arithmetic operations suitable for base classes that include the full complement of arithmetic operators.

`Vecs` are constructed and assigned by copying. Thus, they should normally be passed by reference in applications programs.

Several mapping functions are provided that allow programmers to specify operations on vectors as a whole.

For illustrative purposes assume that classes `intVec` and `intAVec` have been generated via `genclass`.

Constructors and assignment

`intVec a;`
: declares a to be an empty vector. Its size may be changed via resize.

`intVec a(10);`
: declares a to be an uninitialized vector of ten elements (numbered 0-9).

`intVec b(6, 0);`
: declares b to be a vector of six elements, all initialized to zero. Any value can be used as the initial fill argument.

`a = b;`
: Copies b to a. a is resized to be the same as b.

`a = b.at(2, 4)`
: constructs a from the 4 elements of b starting at b[2].

Assume declarations of `intVec a, b, c` and `int i, x` in the following.

Status and access

`a.capacity();`
: returns the number of elements that can be held in a.

`a.resize(20);`
: sets a's length to 20. All elements are unchanged, except that if the new size is smaller than the

original, than trailing elements are deleted, and if greater, trailing elements are uninitialized.

`a[i];`

returns a reference to the i'th element of a, or produces an error if i is out of range.

`a.elem(i)`

returns a reference to the i'th element of a. Unlike the `[]` operator, i is not checked to ensure that it is within range.

`a == b;`

returns true if a and b contain the same elements in the same order.

`a != b;`

is the converse of a == b.

Constructive operations

`c = concat(a, b);`

sets c to the new vector constructed from all of the elements of a followed by all of b.

`c = map(f, a);`

sets c to the new vector constructed by applying int function f(int) to each element of a.

`c = merge(a, b, f);`

sets c to the new vector constructed by merging the elements of ordered vectors a and b using ordering (comparison) function f.

`c = combine(f, a, b);`

sets c to the new vector constructed by applying int function f(int, int) to successive pairs of a and b. The result has length the shorter of a and b.

`c = reverse(a)`

sets c to a, with elements in reverse order.

Destructive operations

`a.reverse();`

reverses a in-place.

`a.sort(f)`

sorts a in-place using comparison function f. The sorting method is a variation of the quicksort functions supplied with GNU emacs.

`a.fill(0, 4, 2)`

fills the 2 elements starting at a[4] with zero.

Other operations

`a.apply(f)`

applies function f to each element in a.

`x = a.reduce(f, base)`

accumulates the results of applying function f to successive elements of a starting with base.

`a.index(int targ);`

returns the index of the leftmost occurrence of the target, or -1, if it does not occur.

`a.error(char* msg)`

invokes the error handler. The default version prints the error message, then aborts.

AVec operations.

AVecs provide additional arithmetic operations. All vector-by-vector operators generate an error if the vectors are not the same length. The following operations are provided, for `AVecs a, b` and base element (scalar) `s`.

`a = b;`

Copies b to a. a and b must be the same size.

`a = s;`

fills all elements of a with the value s. a is not resized.

`a + s; a - s; a * s; a / s`

adds, subtracts, multiplies, or divides each element of a with the scalar.

`a += s; a -= s; a *= s; a /= s;`

adds, subtracts, multiplies, or divides the scalar into a.

`a + b; a - b; product(a, b), quotient(a, b)`

adds, subtracts, multiplies, or divides corresponding elements of a and b.

`a += b; a -= b; a.product(b); a.quotient(b);`

adds, subtracts, multiplies, or divides corresponding elements of b into a.

`s = a * b;`

returns the inner (dot) product of a and b.

`x = a.sum();`

returns the sum of elements of a.

`x = a.sumsq();`

returns the sum of squared elements of a.

`x = a.min();`

returns the minimum element of a.

`x = a.max();`

returns the maximum element of a.

`i = a.min_index();`

returns the index of the minimum element of a.

`i = a.max_index();`

returns the index of the maximum element of a.

Note that it is possible to apply vector versions other arithmetic operators via the mapping functions. For example, to set vector b to the cosines of doubleVec a, use `b = map(cos, a);`. This is often more efficient than performing the operations in an element-by-element fashion.

Go to the previous, next section.

Go to the previous, next section.

Plex classes

A "Plex" is a kind of array with the following properties:

- Plexes may have arbitrary upper and lower index bounds. For example a Plex may be declared to run from indices -10 .. 10.
- Plexes may be dynamically expanded at both the lower and upper bounds of the array in steps of one element.
- Only elements that have been specifically initialized or added may be accessed.
- Elements may be accessed via indices. Indices are always checked for validity at run time. Plexes may be traversed via simple variations of standard array indexing loops.
- Plex elements may be accessed and traversed via Pixes.
- Plex-to-Plex assignment and related operations on entire Plexes are supported.
- Plex classes contain methods to help programmers check the validity of indexing and pointer operations.
- Plexes form "natural" base classes for many restricted-access data structures relying on logically contiguous indices, such as array-based stacks and queues.
- Plexes are implemented as pseudo-generic classes, and must be generated via the `genclass` utility.

Four subclasses of Plexes are supported: A `FPlex` is a Plex that may only grow or shrink within declared bounds; an `XPlex` may dynamically grow or shrink without bounds; an `RPlex` is the same as an `XPlex` but better supports indexing with poor locality of reference; a `MPlex` may grow or shrink, and additionally allows the logical deletion and restoration of elements. Because these classes are virtual subclasses of the "abstract" class `Plex`, it is possible to write user code such as `void f(Plex& a) ...` that operates on any kind of Plex. However, as with nearly any virtual class, specifying the particular Plex class being used results in more efficient code.

Plexes are implemented as a linked list of `IChunks`. Each chunk contains a part of the array. Chunk sizes may be specified within Plex constructors. Default versions also exist, that use a `#define'd` default. Plexes grow by filling unused space in existing chunks, if possible, else, except for FPlexes, by adding another chunk. Whenever Plexes grow by a new chunk, the default element constructors (i.e., those which take no arguments) for all chunk elements are called at once. When Plexes shrink, destructors for the elements are not called until an entire chunk is freed. For this reason, Plexes (like C++ arrays) should only be used for elements with default constructors and destructors that have no side effects.

Plexes may be indexed and used like arrays, although traversal syntax is slightly different. Even though Plexes maintain elements in lists of chunks, they are implemented so that iteration and other constructs that maintain locality of reference require very little overhead over that for simple array traversal Pix-based traversal is also supported. For example, for a plex, p, of ints, the following traversal methods could be used.


```
for (int i = p.low(); i < p.fence(); p.next(i)) use(p[i]);
for (int i = p.high(); i > p.ecnef(); p.prev(i)) use(p[i]);
for (Pix t = p.first(); t != 0; p.next(t)) use(p(i));
for (Pix t = p.last(); t != 0; p.prev(t)) use(p(i));
```

Except for MPlexes, simply using `++i` and `--i` works just as well as `p.next(i)` and `p.prev(i)` when traversing by index. Index-based traversal is generally a bit faster than Pix-based traversal.

`XPlexes` and `MPlexes` are less than optimal for applications in which widely scattered elements are indexed, as might occur when using Plexes as hash tables or "manually" allocated linked lists. In such applications, `RPlexes` are often preferable. `RPlexes` use a secondary chunk index table that requires slightly greater, but entirely uniform overhead per index operation.

Even though they may grow in either direction, Plexes are normally constructed so that their "natural" growth direction is upwards, in that default chunk construction leaves free space, if present, at the end of the plex. However, if the chunksize arguments to constructors are negative, they leave space at the beginning.

All versions of Plexes support the following basic capabilities. (letting `Plex` stand for the type name constructed via the genclass utility (e.g., `intPlex`, `doublePlex`)). Assume declarations of `Plex p, q`, `int i, j`, base element `x`, and Pix `pix`.

`Plex p;`

Declares p to be an initially zero-sized Plex with low index of zero, and the default chunk size. For FPlexes, chunk sizes represent maximum sizes.

`Plex p(int size);`

Declares p to be an initially zero-sized Plex with low index of zero, and the indicated chunk size. If size is negative, then the Plex is created with free space at the beginning of the Plex, allowing more efficient add_low() operations. Otherwise, it leaves space at the end.

`Plex p(int low, int size);`

Declares p to be an initially zero-sized Plex with low index of low, and the indicated chunk size.

`Plex p(int low, int high, Base initval, int size = 0);`

Declares p to be a Plex with indices from low to high, initially filled with initval, and the indicated chunk size if specified, else the default or (high - low + 1), whichever is greater.

`Plex q(p);`

Declares q to be a copy of p.

`p = q;`

Copies Plex q into p, deleting its previous contents.

`p.length()`

Returns the number of elements in the Plex.

`p.empty()`

Returns true if Plex p contains no elements.

`p.full()`

Returns true if Plex p cannot be expanded. This always returns false for XPlexes and MPlexes.

`p[i]`

Returns a reference to the i'th element of p. An exception (error) occurs if i is not a valid index.

`p.valid(i)`

Returns true if i is a valid index into Plex p.

`p.low(); p.high();`

Return the minimum (maximum) valid index of the Plex, or the high (low) fence if the plex is empty.

`p.ecnef(); p.fence();`

Return the index one position past the minimum (maximum) valid index.

`p.next(i); i = p.prev(i);`

Set i to the next (previous) index. This index may not be within bounds.

`p(pix)`

returns a reference to the item at Pix pix.

`pix = p.first(); pix = p.last();`

Return the minimum (maximum) valid Pix of the Plex, or 0 if the plex is empty.

`p.next(pix); p.prev(pix);`

set pix to the next (previous) Pix, or 0 if there is none.

`p.owns(pix)`

Returns true if the Plex contains the element associated with pix.

`p.Pix_to_index(pix)`

If pix is a valid Pix to an element of the Plex, returns its corresponding index, else raises an exception.

`ptr = p.index_to_Pix(i)`

if i is a valid index, returns a the corresponding Pix.

`p.low_element(); p.high_element();`

Return a reference to the element at the minimum (maximum) valid index. An exception occurs if the Plex is empty.

`p.can_add_low(); p.can_add_high();`

Returns true if the plex can be extended one element downward (upward). These always return true for XPlex and MPlex.

`j = p.add_low(x); j = p.add_high(x);`

Extend the Plex by one element downward (upward). The new minimum (maximum) index is returned.

`j = p.del_low(); j = p.del_high()`

Shrink the Plex by one element on the low (high) end. The new minimum (maximum) element is returned. An exception occurs if the Plex is empty.

`p.append(q);`

Append all of Plex q to the high side of p.

`p.prepend(q);`

Prepend all of q to the low side of p.

`p.clear()`

Delete all elements, resetting p to a zero-sized Plex.

`p.reset_low(i);`

Resets p to be indexed starting at low() = i. For example. if p were initially declared via `Plex p(0, 10, 0)`, and then re-indexed via `p.reset_low(5)`, it could then be indexed from indices 5 .. 14.

`p.fill(x)`

sets all p[i] to x.

`p.fill(x, lo, hi)`

sets all of p[i] from lo to hi, inclusive, to x.

`p.reverse()`

reverses p in-place.

`p.chunk_size()`

returns the chunk size used for the plex.

`p.error(const char * msg)`

calls the resettable error handler.

MPlexes are plexes with bitmaps that allow items to be logically deleted and restored. They behave like other plexes, but also support the following additional and modified capabilities:

`p.del_index(i); p.del_Pix(pix)`

logically deletes p[i] (p(pix)). After deletion, attempts to access p[i] generate a error. Indexing via low(), high(), prev(), and next() skip the element. Deleting an element never changes the logical bounds of the plex.

`p.undel_index(i); p.undel_Pix(pix)`

logically undeletes p[i] (p(pix)).

`p.del_low(); p.del_high()`

Delete the lowest (highest) undeleted element, resetting the logical bounds of the plex to the next lowest (highest) undeleted index. Thus, MPlex del_low() and del_high() may shrink the bounds of the plex by more than one index.

`p.adjust_bounds()`

Resets the low and high bounds of the Plex to the indexes of the lowest and highest actual undeleted elements.

`int i = p.add(x)`

Adds x in an unused index, if possible, else performs add_high.

`p.count()`

returns the number of valid (undeleted) elements.

`p.available()`

returns the number of available (deleted) indices.

`int i = p.unused_index()`

returns the index of some deleted element, if one exists, else triggers an error. An unused element may be reused via undel.

`pix = p.unused_Pix()`

returns the pix of some deleted element, if one exists, else 0. An unused element may be reused via undel.

Go to the previous, next section.

Go to the previous, next section.

Stacks

Stacks are declared as an "abstract" class. They are currently implemented in any of three ways.

`VStack`
: implement fixed sized stacks via arrays.

`XPStack`
: implement dynamically-sized stacks via XPlexes.

`SLStack`
: implement dynamically-size stacks via linked lists.

All possess the same capabilities. They differ only in constructors. VStack constructors require a fixed maximum capacity argument. XPStack constructors optionally take a chunk size argument. SLStack constructors take no argument.

Assume the declaration of a base element `x`.

`Stack s; or Stack s(int capacity)`
: declares a Stack.

`s.empty()`
: returns true if stack s is empty.

`s.full()`
: returns true if stack s is full. XPStacks and SLStacks never become full.

`s.length()`
: returns the current number of elements in the stack.

`s.push(x)`
: pushes x on stack s.

`x = s.pop()`
: pops and returns the top of stack

`s.top()`
: returns a reference to the top of stack.

`s.del_top()`
: pops, but does not return the top of stack. When large items are held on the stack it is often a good idea to use `top()` to inspect and use the top of stack, followed by a `del_top()`

`s.clear()`
: removes all elements from the stack.

Go to the previous, next section.

Go to the previous, next section.

Queues

Queues are declared as an "abstract" class. They are currently implemented in any of three ways.

`VQueue`
: implement fixed sized Queues via arrays.

`XPQueue`
: implement dynamically-sized Queues via XPlexes.

`SLQueue`
: implement dynamically-size Queues via linked lists.

All possess the same capabilities; they differ only in constructors. `VQueue` constructors require a fixed maximum capacity argument. `XPQueue` constructors optionally take a chunk size argument. `SLQueue` constructors take no argument.

Assume the declaration of a base element `x`.

`Queue q; or Queue q(int capacity);`
: declares a queue.

`q.empty()`
: returns true if queue q is empty.

`q.full()`
: returns true if queue q is full. XPQueues and SLQueues are never full.

`q.length()`
: returns the current number of elements in the queue.

`q.enq(x)`
: enqueues x on queue q.

`x = q.deq()`
: dequeues and returns the front of queue

`q.front()`
: returns a reference to the front of queue.

`q.del_front()`
: dequeues, but does not return the front of queue

`q.clear()`
: removes all elements from the queue.

Go to the previous, next section.

Go to the previous, next section.

Double ended Queues

Deques are declared as an "abstract" class. They are currently implemented in two ways.

`XPDeque`
: implement dynamically-sized Deques via XPlexes.

`DLDeque`
: implement dynamically-size Deques via linked lists.

All possess the same capabilities. They differ only in constructors. XPDeque constructors optionally take a chunk size argument. DLDeque constructors take no argument.

Double-ended queues support both stack-like and queue-like capabilities:

Assume the declaration of a base element `x`.

`Deque d; or Deque d(int initial_capacity)`
: declares a deque.

`d.empty()`
: returns true if deque d is empty.

`d.full()`
: returns true if deque d is full. Always returns false in current implementations.

`d.length()`
: returns the current number of elements in the deque.

`d.enq(x)`
: inserts x at the rear of deque d.

`d.push(x)`
: inserts x at the front of deque d.

`x = d.deq()`
: dequeues and returns the front of deque

`d.front()`
: returns a reference to the front of deque.

`d.rear()`
: returns a reference to the rear of the deque.

`d.del_front()`
: deletes, but does not return the front of deque

`d.del_rear()`
: deletes, but does not return the rear of the deque.

`d.clear()`

removes all elements from the deque.

Go to the previous, next section.

Go to the previous, next section.

Priority Queue class prototypes.

Priority queues maintain collections of objects arranged for fast access to the least element.

Several prototype implementations of priority queues are supported.

`XPPQs`

implement 2-ary heaps via XPlexes.

`SplayPQs`

implement PQs via Sleator and Tarjan's (JACM 1985) splay trees. The algorithms use a version of "simple top-down splaying" (described on page 669 of the article). The simple-splay mechanism for priority queue functions is loosely based on the one used by D. Jones in the C splay tree functions available from volume 14 of the uunet.uu.net archives.

`PHPQs`

implement pairing heaps (as described by Fredman and Sedgewick in Algorithmica, Vol 1, p111-129). Storage for heap elements is managed via an internal freelist technique. The constructor allows an initial capacity estimate for freelist space. The storage is automatically expanded if necessary to hold new items. The deletion technique is a fast "lazy deletion" strategy that marks items as deleted, without reclaiming space until the items come to the top of the heap.

All PQ classes support the following operations, for some PQ class `Heap`, instance `h`, `Pix ind`, and base class variable `x`.

`h.empty()`

returns true if there are no elements in the PQ.

`h.length()`

returns the number of elements in h.

`ind = h.enq(x)`

Places x in the PQ, and returns its index.

`x = h.deq()`

Dequeues the minimum element of the PQ into x, or generates an error if the PQ is empty.

`h.front()`

returns a reference to the minimum element.

`h.del_front()`

deletes the minimum element.

`h.clear();`

deletes all elements from h;

`h.contains(x)`

returns true if x is in h.

`h(ind)`

returns a reference to the item indexed by ind.

`ind = h.first()`

returns the Pix of first item in the PQ or 0 if empty. This need not be the Pix of the least element.

`h.next(ind)`

advances ind to the Pix of next element, or 0 if there are no more.

`ind = h.seek(x)`

Sets ind to the Pix of x, or 0 if x is not in h.

`h.del(ind)`

deletes the item with Pix ind.

Go to the previous, next section.

Go to the previous, next section.

Set class prototypes

Set classes maintain unbounded collections of items containing no duplicate elements.

These are currently implemented in several ways, differing in representation strategy, algorithmic efficiency, and appropriateness for various tasks. (Listed next to each are average (followed by worst-case, if different) time complexities for [a] adding, [f] finding (via seek, contains), [d] deleting, elements, and [c] comparing (via ==, <=) and [m] merging (via |=, -=, &=) sets).

`XPSets`
: implement unordered sets via XPlexes. ([a O(n)], [f O(n)], [d O(n)], [c O(n^2)] [m O(n^2)]).

`OXPSets`
: implement ordered sets via XPlexes. ([a O(n)], [f O(log n)], [d O(n)], [c O(n)] [m O(n)]).

`SLSets`
: implement unordered sets via linked lists ([a O(n)], [f O(n)], [d O(n)], [c O(n^2)] [m O(n^2)]).

`OSLSets`
: implement ordered sets via linked lists ([a O(n)], [f O(n)], [d O(n)], [c O(n)] [m O(n)]).

`AVLSets`
: implement ordered sets via threaded AVL trees ([a O(log n)], [f O(log n)], [d O(log n)], [c O(n)] [m O(n)]).

`BSTSets`
: implement ordered sets via binary search trees. The trees may be manually rebalanced via the O(n) `balance()` member function. ([a O(log n)/O(n)], [f O(log n)/O(n)], [d O(log n)/O(n)], [c O(n)] [m O(n)]).

`SplaySets`
: implement ordered sets via Sleator and Tarjan's (JACM 1985) splay trees. The algorithms use a version of "simple top-down splaying" (described on page 669 of the article). (Amortized: [a O(log n)], [f O(log n)], [d O(log n)], [c O(n)] [m O(n log n)]).

`VHSets`
: implement unordered sets via hash tables. The tables are automatically resized when their capacity is exhausted. ([a O(1)/O(n)], [f O(1)/O(n)], [d O(1)/O(n)], [c O(n)/O(n^2)] [m O(n)/O(n^2)]).

`VOHSets`
: implement unordered sets via ordered hash tables The tables are automatically resized when their capacity is exhausted. ([a O(1)/O(n)], [f O(1)/O(n)], [d O(1)/O(n)], [c O(n)/O(n^2)] [m O(n)/O(n^2)]).

`CHSets`
: implement unordered sets via chained hash tables. ([a O(1)/O(n)], [f O(1)/O(n)], [d O(1)/O(n)], [c O(n)/O(n^2)] [m O(n)/O(n^2)]).

The different implementations differ in whether their constructors require an argument specifying the initial capacity. Initial capacities are required for plex and hash table based Sets. If none is given `DEFAULT_INITIAL_CAPACITY` (from `` `<T>defs.h' ``) is used.

Sets support the following operations, for some class `Set`, instances `a` and `b`, `Pix ind`, and base element `x`. Since all implementations are virtual derived classes of the `<T>Set` class, it is possible to mix and match operations across different implementations, although, as usual, operations are generally faster when the particular classes are specified in functions operating on Sets.

Pix-based operations are more fully described in the section on Pixes. See section Pseudo-indexes

`Set a; or Set a(int initial_size);`

Declares a to be an empty Set. The second version is allowed in set classes that require initial capacity or sizing specifications.

`a.empty()`

returns true if a is empty.

`a.length()`

returns the number of elements in a.

`Pix ind = a.add(x)`

inserts x into a, returning its index.

`a.del(x)`

deletes x from a.

`a.clear()`

deletes all elements from a;

`a.contains(x)`

returns true if x is in a.

`a(ind)`

returns a reference to the item indexed by ind.

`ind = a.first()`

returns the Pix of first item in the set or 0 if the Set is empty. For ordered Sets, this is the Pix of the least element.

`a.next(ind)`

advances ind to the Pix of next element, or 0 if there are no more.

`ind = a.seek(x)`

Sets ind to the Pix of x, or 0 if x is not in a.

`a == b`

returns true if a and b contain all the same elements.

`a != b`

returns true if a and b do not contain all the same elements.

`a <= b`

returns true if a is a subset of b.

`a |= b`

Adds all elements of b to a.

`a -= b`

Deletes all elements of b from a.

`a &= b`

Deletes all elements of a not occurring in b.

Go to the previous, next section.

Go to the previous, next section.

Bag class prototypes

Bag classes maintain unbounded collections of items potentially containing duplicate elements.

These are currently implemented in several ways, differing in representation strategy, algorithmic efficiency, and appropriateness for various tasks. (Listed next to each are average (followed by worst-case, if different) time complexities for [a] adding, [f] finding (via seek, contains), [d] deleting elements).

`XPBags`
: implement unordered Bags via XPlexes. ([a O(1)], [f O(n)], [d O(n)]).

`OXPBags`
: implement ordered Bags via XPlexes. ([a O(n)], [f O(log n)], [d O(n)]).

`SLBags`
: implement unordered Bags via linked lists ([a O(1)], [f O(n)], [d O(n)]).

`OSLBags`
: implement ordered Bags via linked lists ([a O(n)], [f O(n)], [d O(n)]).

`SplayBags`
: implement ordered Bags via Sleator and Tarjan's (JACM 1985) splay trees. The algorithms use a version of "simple top-down splaying" (described on page 669 of the article). (Amortized: [a O(log n)], [f O(log n)], [d O(log n)]).

`VHBags`
: implement unordered Bags via hash tables. The tables are automatically resized when their capacity is exhausted. ([a O(1)/O(n)], [f O(1)/O(n)], [d O(1)/O(n)]).

`CHBags`
: implement unordered Bags via chained hash tables. ([a O(1)/O(n)], [f O(1)/O(n)], [d O(1)/O(n)]).

The implementations differ in whether their constructors require an argument to specify their initial capacity. Initial capacities are required for plex and hash table based Bags. If none is given `DEFAULT_INITIAL_CAPACITY` (from `` `<T>defs.h' ``) is used.

Bags support the following operations, for some class `Bag`, instances `a` and `b`, `Pix ind`, and base element `x`. Since all implementations are virtual derived classes of the `<T>Bag` class, it is possible to mix and match operations across different implementations, although, as usual, operations are generally faster when the particular classes are specified in functions operating on Bags.

Pix-based operations are more fully described in the section on Pixes. See section Pseudo-indexes

`Bag a; or Bag a(int initial_size)`
: Declares a to be an empty Bag. The second version is allowed in Bag classes that require initial capacity or sizing specifications.

`a.empty()`
: returns true if a is empty.

`a.length()`
: returns the number of elements in a.

`ind = a.add(x)`
: inserts x into a, returning its index.

`a.del(x)`
: deletes one occurrence of x from a.

`a.remove(x)`
: deletes all occurrences of x from a.

`a.clear()`
: deletes all elements from a;

`a.contains(x)`
: returns true if x is in a.

`a.nof(x)`
: returns the number of occurrences of x in a.

`a(ind)`
: returns a reference to the item indexed by ind.

`int = a.first()`
: returns the Pix of first item in the Bag or 0 if the Bag is empty. For ordered Bags, this is the Pix of the least element.

`a.next(ind)`
: advances ind to the Pix of next element, or 0 if there are no more.

`ind = a.seek(x, Pix from = 0)`
: Sets ind to the Pix of the next occurrence x, or 0 if there are none. If from is 0, the first occurrence is returned, else the following from.

Go to the previous, next section.

Go to the previous, next section.

Map Class Prototypes

Maps support associative array operations (insertion, deletion, and membership of records based on an associated key). They require the specification of two types, the key type and the contents type.

These are currently implemented in several ways, differing in representation strategy, algorithmic efficiency, and appropriateness for various tasks. (Listed next to each are average (followed by worst-case, if different) time complexities for [a] accessing (via op [], contains), [d] deleting elements).

`AVLMaps`

implement ordered Maps via threaded AVL trees ([a O(log n)], [d O(log n)]).

`RAVLMaps`

Similar, but also maintain ranking information, used via `ranktoPix(int r)`, that returns the `Pix` of the item at rank r, and `rank(key)` that returns the rank of the corresponding item. ([a O(log n)], [d O(log n)]).

`SplayMaps`

implement ordered Maps via Sleator and Tarjan's (JACM 1985) splay trees. The algorithms use a version of "simple top-down splaying" (described on page 669 of the article). (Amortized: [a O(log n)], [d O(log n)]).

`VHMaps`

implement unordered Maps via hash tables. The tables are automatically resized when their capacity is exhausted. ([a O(1)/O(n)], [d O(1)/O(n)]).

`CHMaps`

implement unordered Maps via chained hash tables. ([a O(1)/O(n)], [d O(1)/O(n)]).

The different implementations differ in whether their constructors require an argument specifying their initial capacity. Initial capacities are required for hash table based Maps. If none is given `DEFAULT_INITIAL_CAPACITY` (from `` `<T>defs.h' ``) is used.

All Map classes share the following operations (for some Map class, `Map` instance `d`, `Pix ind` and key variable `k`, and contents variable `x`).

Pix-based operations are more fully described in the section on Pixes. See section Pseudo-indexes

`Map d(x); Map d(x, int initial_capacity)`

Declare d to be an empty Map. The required argument, x, specifies the default contents, i.e., the contents of an otherwise uninitialized location. The second version, specifying initial capacity is allowed for Maps with an initial capacity argument.

`d.empty()`

returns true if d contains no items.

`d.length()`

returns the number of items in d.

`d[k]`

returns a reference to the contents of item with key k. If no such item exists, it is installed with the default contents. Thus d[k] = x installs x, and x = d[k] retrieves it.

`d.contains(k)`

returns true if an item with key field k exists in d.

`d.del(k)`

deletes the item with key k.

`d.clear()`

deletes all items from the table.

`x = d.dflt()`

returns the default contents.

`k = d.key(ind)`

returns a reference to the key at Pix ind.

`x = d.contents(ind)`

returns a reference to the contents at Pix ind.

`ind = d.first()`

returns the Pix of the first element in d, or 0 if d is empty.

`d.next(ind)`

advances ind to the next element, or 0 if there are no more.

`ind = d.seek(k)`

returns the Pix of element with key k, or 0 if k is not in d.

Go to the previous, next section.

Go to the previous, next section.

C++ version of the GNU getopt function

The GetOpt class provides an efficient and structured mechanism for processing command-line options from an application program. The sample program fragment below illustrates a typical use of the GetOpt class for some hypothetical application program:

```
#include <stdio.h>
#include <GetOpt.h>
//...
int debug_flag, compile_flag, size_in_bytes;

int
main (int argc, char **argv)
{
  // Invokes ctor `GetOpt (int argc, char **argv,
  //                       char *optstring);'
  GetOpt getopt (argc, argv, "dcs:");
  int option_char;

  // Invokes member function `int operator ()(void);'
  while ((option_char = getopt ()) != EOF)
    switch (option_char)
      {
         case 'd': debug_flag = 1; break;
         case 'c': compile_flag = 1; break;
         case 's': size_in_bytes = atoi (getopt.optarg); break;
         case '?': fprintf (stderr,
                            "usage: %s [dcs<size>]\n", argv[0]);
      }
}
```

Unlike the C library version, the libg++ GetOpt class uses its constructor to initialize class data members containing the argument count, argument vector, and the option string. This simplifies the interface for each subsequent call to member function `int operator ()(void)`.

The C version, on the other hand, uses hidden static variables to retain the option string and argument list values between calls to `getopt`. This complicates the `getopt` interface since the argument count, argument vector, and option string must be passed as parameters for each invocation. For the C version, the loop in the previous example becomes:

```
  while ((option_char = getopt (argc, argv, "dcs:")) != EOF)
    // ...
```

which requires extra overhead to pass the parameters for every call.

Along with the GetOpt constructor and `int operator ()(void)`, the other relevant elements of class GetOpt are:

`char *optarg`

Used for communication from `operator ()(void)` to the caller. When `operator ()(void)` finds an option that takes an argument, the argument value is stored here.

`int optind`

Index in `argv` of the next element to be scanned. This is used for communication to and from the caller and for communication between successive calls to `operator ()(void)`. When `operator ()(void)` returns EOF, this is the index of the first of the non-option elements that the caller should itself scan. Otherwise, `optind` communicates from one call to the next how much of `argv` has been scanned so far.

The libg++ version of GetOpt acts like standard UNIX `getopt` for the calling routine, but it behaves differently for the user, since it allows the user to intersperse the options with the other arguments. As GetOpt works, it permutes the elements of `argv` so that, when it is done, all the options precede everything else. Thus all application programs are extended to handle flexible argument order.

Setting the environment variable _POSIX_OPTION_ORDER disables permutation. Then the behavior is completely standard.

Go to the previous, next section.

Go to the previous section.

Projects and other things left to do

Coming Attractions

Some things that will probably be available in libg++ in the near future:

- Revamped C-compatibility header files that will be compatible with the forthcoming (ANSI-based) GNU libc.a
- A revision of the File-based classes that will use the GNU stdio library, and also be 100% compatible (even at the streambuf level) with the AT&T 2.0 stream classes.
- Additional container class prototypes.
- generic Matrix class prototypes.
- A task package probably based on Dirk Grunwald's threads package.

Wish List

Some things that people have mentioned that they would like to see in libg++, but for which there have not been any offers:

- A method to automatically convert or incorporate libg++ classes so they can be used directly in Gorlen's OOPS environment.
- A class browser.
- A better general exception-handling strategy.
- Better documentation.

How to contribute

Programmers who have written C++ classes that they believe to be of general interest are encourage to write to dl at rocky.oswego.edu. Contributing code is not difficult. Here are some general guidelines:

- FSF must maintain the right to accept or reject potential contributions. Generally, the only reasons for rejecting contributions are cases where they duplicate existing or nearly-released code, contain unremovable specific machine dependencies, or are somehow incompatible with the rest of the library.
- Acceptance of contributions means that the code is accepted for adaptation into libg++. FSF must reserve the right to make various editorial changes in code. Very often, this merely entails formatting, maintenance of various conventions, etc. Contributors are always given authorship credit and shown the final version for approval.
- Contributors must assign their copyright to FSF via a form sent out upon acceptance. Assigning copyright to FSF ensures that the code may be freely distributed.

- Assistance in providing documentation, test files, and debugging support is strongly encourag

Extensions, comments, and suggested modifications of existing libg++ features are also very welcome.

Go to the previous section.

GNU Libtool

For version 1.2, 10 March 1998

Gordon Matzigkeit

This file documents GNU Libtool, a script that allows package developers to provide generic shared library support. This edition documents version 1.2.

See section Reporting bugs, for information on how to report problems with libtool.

This document was generated on 20 May 1999 using the texi2html translator version 1.51a.

Go to the first, previous, next, last section, table of contents.

Troubleshooting

Libtool is under constant development, changing to remain up-to-date with modern operating systems. If libtool doesn't work the way you think it should on your platform, you should read this chapter to help determine what the problem is, and how to resolve it.

The libtool test suite

Libtool comes with its own set of programs that test its capabilities, and report obvious bugs in the libtool program. These tests, too, are constantly evolving, based on past problems with libtool, and known deficiencies in other operating systems.

As described in the `INSTALL' file, you may run make check after you have built libtool (possibly before you install it) in order to make sure that it meets basic functional requirements.

Description of test suite

Here is a list of the current programs in the test suite, and what they test for:

`demo-conf.test`

`demo-exec.test`

`demo-inst.test`

`demo-make.test`

`demo-unst.test`

These programs check to see that the `demo' subdirectory of the libtool distribution can be configured, built, installed, and uninstalled correctly. The `demo' subdirectory contains a demonstration of a trivial package that uses libtool.

`hardcode.test`

On all systems with shared libraries, the location of the library can be encoded in executables that are linked against it see section Linking executables. This test checks the conditions under which your system linker hardcodes the library location, and guarantees that they correspond to libtool's own notion of how your linker behaves.

`link.test`

This test guarantees that linking directly against a non-libtool static library works properly.

`link-2.test`

This test makes sure that files ending in `.lo' are never linked directly into a program file.

`suffix.test`

When other programming languages are used with libtool (see section Using libtool with other languages), the source files may end in suffixes other than `.c'. This test validates that libtool can handle suffixes for all the file types that it supports, and that it fails when the suffix is invalid.

`test-e.test`

This program checks that the `test -e` construct is *never* used in the libtool scripts. Checking for the existence of a file can only be done in a portable way by using `test -f`.

When tests fail

Each of the above tests are designed to produce no output when they are run via make check. The exit status of each program tells the `Makefile' whether or not the test succeeded.

If a test fails, it means that there is either a programming error in libtool, or in the test program itself.

To investigate a particular test, you may run it directly, as you would a normal program. When the test is invoked in this way, it produces output which may be useful in determining what the problem is.

Another way to have the test programs produce output is to set the VERBOSE environment variable to `yes' before running them. For example, env VERBOSE=yes make check runs all the tests, and has each of them display debugging information.

Reporting bugs

If you think you have discovered a bug in libtool, you should think twice: the libtool maintainer is notorious for passing the buck (or maybe that should be "passing the bug"). Libtool was invented to fix known deficiencies in shared library implementations, so, in a way, most of the bugs in libtool are actually bugs in other operating systems. However, the libtool maintainer would definitely be happy to add support for somebody else's buggy operating system. [I wish there was a good way to do winking smiley-faces in texinfo.]

Genuine bugs in libtool include problems with shell script portability, documentation errors, and failures in the test suite (see section The libtool test suite).

First, check the documentation and help screens to make sure that the behaviour you think is a problem is not already mentioned as a feature.

Then, you should read the Emacs guide to reporting bugs (see section `Reporting Bugs' in The Emacs Manual). Some of the details listed there are specific to Emacs, but the principle behind them is a general one.

Finally, send a bug report to the libtool mailing list `<bug-libtool@gnu.org>` with any appropriate *facts*, such as test suite output (see section When tests fail), all the details needed to reproduce the bug, and a brief description of why you think the behaviour is a bug. Be sure to include the word "libtool" in the subject line, as well as the version number you are using (which can be found by typing ltconfig --version).

Please include the generated `libtool` script with your bug report, so that I can see what values `ltconfig` guessed for your system.

TOC

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Introduction

In the past, if a source code package developer wanted to take advantage of the power of shared libraries, he needed to write custom support code for each platform on which his package ran. He also had to design a configuration interface so that the package installer could choose what sort of libraries were built.

GNU Libtool simplifies the developer's job by encapsulating both the platform-specific dependencies, and the user interface, in a single script. GNU Libtool is designed so that the complete functionality of each host type is available via a generic interface, but nasty quirks are hidden from the programmer.

GNU Libtool's consistent interface is reassuring... users don't need to read obscure documentation in order to have their favorite source package build shared libraries. They just run your package `configure` script (or equivalent), and libtool does all the dirty work.

There are several examples throughout this document. All assume the same environment: we want to build a library, `` `libhello' ``, in a generic way.

`` `libhello' `` could be a shared library, a static library, or both... whatever is available on the host system, as long as libtool has been ported to it.

This chapter explains the original design philosophy of libtool. Feel free to skip to the next chapter, unless you are interested in history, or want to write code to extend libtool in a consistent way.

Motivation for writing libtool

Since early 1995, several different GNU developers have recognized the importance of having shared library support for their packages. The primary motivation for such a change is to encourage modularity and reuse of code (both conceptually and physically) in GNU programs.

Such a demand means that the way libraries are built in GNU packages needs to be general, to allow for any library type the package installer might want. The problem is compounded by the absence of a standard procedure for creating shared libraries on different platforms.

The following sections outline the major issues facing shared library support in GNU, and how I propose that shared library support could be standardized with libtool.

The following specifications were used in developing and evaluating this system:

1. The system must be as elegant as possible.
2. The system must be fully integrated with the GNU Autoconf and Automake utilities, so that it will be easy for GNU maintainers to use. However, the system must not require these tools, so that it can be used by non-GNU packages.

3. Portability to other (non-GNU) architectures and tools is desirable.

Implementation issues

The following issues need to be addressed in any reusable shared library system, specifically libtool:

1. The package installer should be able to control what sort of libraries are built.
2. It can be tricky to run dynamically linked programs whose libraries have not yet been installed. LD_LIBRARY_PATH must be set properly (if it is supported), or programs fail to run.
3. The system must operate consistently even on hosts which don't support shared libraries.
4. The commands required to build shared libraries may differ wildly from host to host. These need to be determined at configure time in a consistent way.
5. It is not always obvious with which suffix a shared library should be installed. This makes it difficult for `Makefile' rules, since they generally assume that file names are the same from host to host.
6. The system needs a simple library version number abstraction, so that shared libraries can be upgraded in place. The programmer should be informed how to design the interfaces to the library to maximize binary compatibility.
7. The install `Makefile' target should warn the package installer to set the proper environment variables (LD_LIBRARY_PATH or equivalent), or run `ldconfig(8)`.

Other implementations

I have investigated several different implementations of systems that build shared libraries as part of a free software package. At first, I made notes on the features of each of these packages for comparison purposes.

Now it is clear that none of these packages have documented the details of shared library systems that libtool requires. So, other packages have been more or less abandoned as influences.

A postmortem analysis of other implementations

In all fairness, each of the implementations that I examined do the job that they were intended to do, for a number of different host systems. However, none of these solutions seem to function well as a generalized, reusable component.

Most were too complex for me to use (much less modify) without understanding exactly what the implementation does, and they were generally not documented.

I think the main problem is that different vendors have different views of what libraries are, and none of the packages I examined seemed to be confident enough to settle on a single paradigm that just *works*.

Ideally, libtool would be a standard that would be implemented as series of extensions and modifications to existing library systems to make them work consistently. However, I don't have the time or power to

convince operating system developers to mend their evil ways, and I want to build shared libraries r now, even on buggy, broken, confused operating systems.

For this reason, I have designed libtool as an independent shell script. It isolates the problems and inconsistencies in library building that plague `Makefile' writers by wrapping the compiler suite on different platforms with a consistent, powerful interface.

I hope that libtool will be useful to and used by the GNU community, and that the lessons I've learned in writing it will be taken up and implemented by designers of library systems.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

TOC

The libtool paradigm

At first, libtool was designed to support an arbitrary number of library object types. After porting libtool to more platforms, I discovered a new paradigm for describing the relationship between libraries and programs.

In summary, "libraries are programs with multiple entry points, and more formally defined interfaces."

Version 0.7 of libtool was a complete redesign and rewrite of libtool to reflect this new paradigm. So far, it has proved to be successful: libtool is simpler and more useful than before.

The best way to introduce the libtool paradigm is to contrast it with the paradigm of existing library systems, with examples from each. It is a new way of thinking, so it may take a little time to absorb, but when you understand it, the world becomes simpler.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

TOC

Using libtool

It makes little sense to talk about using libtool in your own packages until you have seen how it makes your life simpler. The examples in this chapter introduce the main features of libtool by comparing the standard library building procedure to libtool's operation on two different platforms:

`a23'
: An Ultrix 4.2 platform with only static libraries.

`burger'
: A NetBSD/i386 1.2 platform with shared libraries.

You can follow these examples on your own platform, using the preconfigured libtool script that was installed with libtool (see section Configuring libtool).

Source files for the following examples are taken from the `demo' subdirectory of the libtool distribution. Assume that we are building a library, `libhello', out of the files `foo.c' and `hello.c'.

Note that the `foo.c' source file uses the `cos(3)` math library function, which is usually found in the standalone math library, and not the C library. So, we need to add -lm to the end of the link line whenever we link `foo.o' or `foo.lo' into an executable or a library (see section Inter-library dependencies).

The same rule applies whenever you use functions that don't appear in the standard C library... you need to add the appropriate -lname flag to the end of the link line when you link against those objects.

After we have built that library, we want to create a program by linking `main.o' against `libhello'.

Creating object files

To create an object file from a source file, the compiler is invoked with the `-c' flag (and any other desired flags):

```
burger$ gcc -g -O -c main.c
burger$
```

The above compiler command produces an object file, `main.o', from the source file `main.c'.

For most library systems, creating object files that become part of a static library is as simple as creating object files that are linked to form an executable:

```
burger$ gcc -g -O -c foo.c
burger$ gcc -g -O -c hello.c
burger$
```

Shared libraries, however, may only be built from **position-independent code** (PIC). So, special flags must be passed to the compiler to tell it to generate PIC rather than the standard position-dependent code.

Since this is a library implementation detail, libtool hides the complexity of PIC compiler flags by using separate library object files (which end in `.lo' instead of `.o'). On systems without shared libraries (or without special PIC compiler flags), these library object files are identical to "standard" object files.

To create library object files for `foo.c' and `hello.c', simply invoke libtool with the standard compilation command as arguments (see section Compile mode):

```
a23$ libtool gcc -g -O -c foo.c
gcc -g -O -c foo.c
ln -s foo.o foo.lo
a23$ libtool gcc -g -O -c hello.c
gcc -g -O -c hello.c
ln -s hello.o hello.lo
a23$
```

Note that libtool creates two object files for each invocation. The `.lo' file is a library object, and the `.o' file is a standard object file. On `a23', these files are identical, because only static libraries are supported.

On shared library systems, libtool automatically inserts the PIC generation flags into the compilation command, so that the library object and the standard object differ:

```
burger$ libtool gcc -g -O -c foo.c
gcc -g -O -c -fPIC -DPIC foo.c
mv -f foo.o foo.lo
gcc -g -O -c foo.c
burger$ libtool gcc -g -O -c hello.c
gcc -g -O -c -fPIC -DPIC hello.c
mv -f hello.o hello.lo
gcc -g -O -c hello.c
burger$
```

Linking libraries

Without libtool, the programmer would invoke the `ar` command to create a static library:

```
burger$ ar cru libhello.a hello.o foo.o
burger$
```

But of course, that would be too simple, so many systems require that you run the `ranlib` command on the resulting library (to give it better karma, or something):

```
burger$ ranlib libhello.a
burger$
```

It seems more natural to use the C compiler for this task, given libtool's "libraries are programs" approach. So, on platforms without shared libraries, libtool simply acts as a wrapper for the system `ar` (and possibly `ranlib`) commands.

Again, the libtool library name differs from the standard name (it has a `.la' suffix instead of a `.a' suffix). The arguments to libtool are the same ones you would use to produce an executable named `libhello.la' with your compiler (see section Link mode):

```
burger$ libtool gcc -g -O -o libhello.la foo.o hello.o
libtool: cannot build libtool library `libhello.la' from non-libtool \
                objects
```


```
burger$
```

Aha! Libtool caught a common error... trying to build a library from standard objects instead of library objects. This doesn't matter for static libraries, but on shared library systems, it is of great importance.

So, let's try again, this time with the library object files:(1)

```
a23$ libtool gcc -g -O -o libhello.la foo.lo hello.lo -lm
libtool: you must specify an installation directory with `-rpath'
a23$
```

Argh. Another complication in building shared libraries is that we need to specify the path to the directory in which they (eventually) will be installed. So, we try again, with an `rpath` setting of `` `/usr/local/lib' ``:

```
a23$ libtool gcc -g -O -o libhello.la foo.lo hello.lo \
                -rpath /usr/local/lib -lm
mkdir .libs
ar cru .libs/libhello.a foo.o hello.o
ranlib .libs/libhello.a
creating libhello.la
a23$
```

Now, let's try the same trick on the shared library platform:

```
burger$ libtool gcc -g -O -o libhello.la foo.lo hello.lo \
                -rpath /usr/local/lib -lm
mkdir .libs
ld -Bshareable -o .libs/libhello.so.0.0 foo.lo hello.lo -lm
ar cru .libs/libhello.a foo.o hello.o
ranlib .libs/libhello.a
creating libhello.la
burger$
```

Now that's significantly cooler... libtool just ran an obscure `ld` command to create a shared library, as well as the static library.

Note how libtool creates extra files in the `` `.libs' `` subdirectory, rather than the current directory. This feature is to make it easier to clean up the build directory, and to help ensure that other programs fail horribly if you accidentally forget to use libtool when you should.

Linking executables

If you choose at this point to **install** the library (put it in a permanent location) before linking executables against it, then you don't need to use libtool to do the linking. Simply use the appropriate `-L' and `-l' flags to specify the library's location.

Some system linkers insist on encoding the full directory name of each shared library in the resulting executable. Libtool has to work around this misfeature by special magic to ensure that only permanent directory names are put into installed executables.

The importance of this bug must not be overlooked: it won't cause programs to crash in obvious ways. It creates a security hole, and possibly even worse, if you are modifying the library source code after you have installed the

package, you will change the behaviour of the installed programs!

So, if you want to link programs against the library before you install it, you must use libtool to do the linking.

Here's the old way of linking against an uninstalled library:

```
burger$ gcc -g -O -o hell.old main.o libhello.a -lm
burger$
```

Libtool's way is almost the same(2) (see section Link mode):

```
a23$ libtool gcc -g -O -o hell main.o libhello.la -lm
gcc -g -O -o hell main.o ./.libs/libhello.a -lm
a23$
```

That looks too simple to be true. All libtool did was transform `libhello.la' to `./.libs/libhello.a', but remember that `a23' has no shared libraries.

On `burger' the situation is different:

```
burger$ libtool gcc -g -O -o hell main.o libhello.la -lm
gcc -g -O -o .libs/hell main.o -L./.libs -R/usr/local/lib -lhello -lm
creating hell
burger$
```

Notice that the executable, `hell`, was actually created in the `.libs' subdirectory. Then, a wrapper script was created in the current directory.

On NetBSD 1.2, libtool encodes the installation directory of `libhello', by using the `-R/usr/local/lib' compiler flag. Then, the wrapper script guarantees that the executable finds the correct shared library (the one in `./.libs') until it is properly installed.

Let's compare the two different programs:

```
burger$ time ./hell.old
Welcome to GNU Hell!
** This is not GNU Hello.  There is no built-in mail reader. **
        0.21 real         0.02 user         0.08 sys
burger$ time ./hell
Welcome to GNU Hell!
** This is not GNU Hello.  There is no built-in mail reader. **
        0.63 real         0.09 user         0.59 sys
burger$
```

The wrapper script takes significantly longer to execute, but at least the results are correct, even though the shared library hasn't been installed yet.

So, what about all the space savings that shared libraries are supposed to yield?

```
burger$ ls -l hell.old libhello.a
-rwxr-xr-x  1 gord  gord  15481 Nov 14 12:11 hell.old
-rw-r--r--  1 gord  gord   4274 Nov 13 18:02 libhello.a
burger$ ls -l .libs/hell .libs/libhello.*
```


```
-rwxr-xr-x  1 gord  gord  11647 Nov 14 12:10 .libs/hell
-rw-r--r--  1 gord  gord   4274 Nov 13 18:44 .libs/libhello.a
-rwxr-xr-x  1 gord  gord  12205 Nov 13 18:44 .libs/libhello.so.0.0
burger$
```

Well, that sucks. Maybe I should just scrap this project and take up basket weaving.

Actually, it just proves an important point: shared libraries incur overhead because of their (relative) complexity. In this situation, the price of being dynamic is eight kilobytes, and the payoff is about four kilobytes. So, having a shared `libhello' won't be an advantage until we link it against at least a few more programs.

Debugging executables

If `hell' was a complicated program, you would certainly want to test and debug it before installing it on your system. In the above section, you saw how it the libtool wrapper script makes it possible to run the program directly, but unfortunately, it interferes with the debugger:

```
burger$ gdb hell
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (i386-unknown-netbsd), Copyright 1996 Free Software Foundation, Inc...

"hell": not in executable format: File format not recognized

(gdb) quit
burger$
```

Sad. It doesn't work because GDB isn't doesn't know where the executable lives. So, let's try again, by invoking GDB directly on the executable:

```
burger$ gdb .libs/hell
trick:/home/src/libtool/demo$ gdb .libs/hell
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (i386-unknown-netbsd), Copyright 1996 Free Software Foundation, Inc...
(gdb) break main
Breakpoint 1 at 0x8048547: file main.c, line 29.
(gdb) run
Starting program: /home/src/libtool/demo/.libs/hell
/home/src/libtool/demo/.libs/hell: can't load library 'libhello.so.2'

Program exited with code 020.
(gdb) quit
burger$
```

Argh. Now GDB complains because it cannot find the shared library that `hell' is linked against. So, we must use libtool in order to properly set the library path and run the debugger. Fortunately, we can forget all about the `.libs' directory, and just run it on the executable wrapper (see section Execute mode):


```
burger$ libtool gdb hell
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (i386-unknown-netbsd), Copyright 1996 Free Software Foundation, Inc...
(gdb) break main
Breakpoint 1 at 0x8048547: file main.c, line 29.
(gdb) run
Starting program: /home/src/libtool/demo/.libs/hell

Breakpoint 1, main (argc=1, argv=0xbffffc40) at main.c:29
29        printf ("Welcome to GNU Hell!\n");
(gdb) quit
The program is running.  Quit anyway (and kill it)? (y or n) y
burger$
```

Installing libraries

Installing libraries on a non-libtool system is quite straightforward... just copy them into place:(3)

```
burger$ su
Password: ********
burger# cp libhello.a /usr/local/lib/libhello.a
burger#
```

Oops, don't forget the `ranlib` command:

```
burger# ranlib /usr/local/lib/libhello.a
burger#
```

Libtool installation is quite simple, as well. Just use the `install` or `cp` command that you normally would (see section Install mode):

```
a23# libtool cp libhello.la /usr/local/lib/libhello.la
cp libhello.la /usr/local/lib/libhello.la
cp .libs/libhello.a /usr/local/lib/libhello.a
ranlib /usr/local/lib/libhello.a
a23#
```

Note that the libtool library `` `libhello.la' `` is also installed, for informational purposes, and to help libtool with uninstallation (see section Uninstall mode).

Here is the shared library example:

```
burger# libtool install -c libhello.la /usr/local/lib/libhello.la
install -c .libs/libhello.so.0.0 /usr/local/lib/libhello.so.0.0
install -c libhello.la /usr/local/lib/libhello.la
install -c .libs/libhello.a /usr/local/lib/libhello.a
ranlib /usr/local/lib/libhello.a
burger#
```

TOC

It is safe to specify the `-s' (strip symbols) flag if you use a BSD-compatible install program when installing libraries. Libtool will either ignore the `-s' flag, or will run a program that will strip only debugging and compiler symbols from the library.

Once the libraries have been put in place, there may be some additional configuration that you need to do before using them. First, you must make sure that where the library is installed actually agrees with the `-rpath' flag you used to build it.

Then, running `libtool -n --finish libdir' can give you further hints on what to do (see section Finish mode):

```
burger# libtool -n --finish /usr/local/lib
ldconfig -m /usr/local/lib
To link against installed libraries in LIBDIR, users may have to:
   - add LIBDIR to their `LD_LIBRARY_PATH' environment variable
   - use the `-LLIBDIR' linker flag
burger#
```

After you have completed these steps, you can go on to begin using the installed libraries. You may also install any executables that depend on libraries you created.

Installing executables

If you used libtool to link any executables against uninstalled libtool libraries (see section Linking executables), you need to use libtool to install the executables after the libraries have been installed (see section Installing libraries).

So, for our Ultrix example, we would run:

```
a23# libtool install -c hell /usr/local/bin/hell
install -c hell /usr/local/bin/hell
a23#
```

On shared library systems, libtool just ignores the wrapper script and installs the correct binary:

```
burger# libtool install -c hell /usr/local/bin/hell
install -c .libs/hell /usr/local/bin/hell
burger#
```

Linking static libraries

Why return to `ar` and `ranlib` silliness when you've had a taste of libtool? Well, sometimes it is desirable to create a static archive that can never be shared. The most frequent case is when you have a "convenience library" that is a collection of related object files without a really nice interface.

To do this, you should ignore libtool entirely, and just use the old `ar` and `ranlib` commands to create a static library.

If you want to install the library (but you probably don't), then you may use libtool if you want:

```
burger$ libtool ./install-sh -c libhello.a /local/lib/libhello.a
./install-sh -c libhello.a /local/lib/libhello.a
ranlib /local/lib/libhello.a
```


```
burger$
```

Using libtool for static library installation protects your library from being accidentally stripped (if the installer used the `-s' flag), as well as automatically running the correct `ranlib` command.

Another common situation where static linking is desirable is in creating a standalone binary. Use libtool to do the linking and add the `-all-static' flag.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Invoking `libtool`

The `libtool` program has the following synopsis:

```
libtool [option]... [mode-arg]...
```

and accepts the following options:

`-n'

`--dry-run'

Don't create, modify, or delete any files, just show what commands would be executed by libtool.

`--features'

Display libtool configuration information and exit. This provides a way for packages to determine whether shared or static libraries will be built.

`--finish'

Same as `--mode=finish'.

`--help'

Display a help message and exit. If `--mode=mode' is specified, then detailed help for mode is displayed.

`--mode=mode'

Use mode as the operation mode. By default, the operation mode is inferred from the contents of mode-args. If mode is specified, it must be one of the following:

`compile'

Compile a source file into a libtool object.

`execute'

Automatically set the library path so that another program can use uninstalled libtool-generated programs or libraries.

`finish'

Complete the installation of libtool libraries on the system.

`install'

Install libraries or executables.

`link'

Create a library or an executable.

`uninstall'

Delete libraries or executables.

`--version'

Print libtool version information and exit.

Compile mode

For `compile' mode, mode-args is a compiler command to be used in creating a `standard' object file. These arguments should begin with the name of the C compiler, and contain the `-c' compiler flag so that only an object file is created.

Libtool determines the name of the output file by removing the directory component from the source file name, then substituting the C source code suffix `.c' with the library object suffix, `.lo'.

If shared libraries are being built, any necessary PIC generation flags are substituted into the compilation command.

Note that the `-o' option is not supported for compile mode, because it cannot be implemented properly for all platforms. It is far easier just to change your Makefiles to create all the output files in the current working directory.

Link mode

`link' mode links together object files (including library objects) to form another library or to create an executable program.

mode-args consist of a command using the C compiler to create an output file (with the `-o' flag) from several object files.

The following components of mode-args are treated specially:

`-all-static'

If output-file is a program, then do not link it against any shared libraries at all. If output-file is a library, then only create a static library.

`-dlopen file'

Same as `-dlpreopen file', if native dlopening is not supported on the host platform (see section Dlopened modules). Otherwise, no effect.

`-dlpreopen file'

Link file into the output program, and add its symbols to dld_preloaded_symbols (see section Dlpreopening).

`-export-dynamic'

Allow symbols from output-file to be resolved with `dlsym(3)` (see section Dlopened modules).

`-Llibdir'

Search libdir for required libraries that have already been installed.

`-lname'

output-file requires the installed library `libname'. This option is required even when

output-file is not an executable.

`-no-undefined'

Declare that output-file does not depend on any other libraries. Some platforms cannot create shared libraries that depend on other libraries (see section Inter-library dependencies).

`-o output-file'

Create output-file from the specified objects and libraries.

`-release release'

Specify that the library was generated by release release of your package, so that users can easily tell which versions are newer than others. Be warned that no two releases of your package will be binary compatible if you use this flag. If you want binary compatibility, use the `-version-info' flag (see section Library interface versions).

`-rpath libdir'

If output-file is a library, it will eventually be installed in libdir.

`-static'

If output-file is a program, then do not link it against any uninstalled shared libtool libraries. If output-file is a library, then only create a static library.

`-version-info current[:revision[:age]]'

If output-file is a libtool library, use interface version information current, revision, and age to build it (see section Library interface versions). Do **not** use this flag to specify package release information, rather see the `-release' flag.

If the output-file ends in `.la', then a libtool library is created, which must be built only from library objects (`.lo' files). The `-rpath' option is required. In the current implementation, libtool libraries may not depend on other uninstalled libtool libraries (see section Inter-library dependencies).

If the output-file ends in `.a', then a standard library is created using `ar` and possibly `ranlib`.

If output-file ends in `.o' or `.lo', then a reloadable object file is created from the input files (generally using `ld -r'). This method is often called **partial linking**.

Otherwise, an executable program is created.

Execute mode

For `execute' mode, the library path is automatically set, then a program is executed.

The first of the mode-args is treated as a program name, with the rest as arguments to that program.

The following components of mode-args are treated specially:

`-dlopen file'

Add the directory containing file to the library path.

This mode sets the library path environment variable according to any `-dlopen' flags.

If any of the args are libtool executable wrappers, then they are translated into the name of their corresponding uninstalled binary, and any of their required library directories are added to the library path.

Install mode

In `install' mode, libtool interprets mode-args as an installation command beginning with `cp`, or a BSD-compatible `install` program.

The rest of the mode-args are interpreted as arguments to that command.

The command is run, and any necessary unprivileged post-installation commands are also completed.

Finish mode

`finish' mode helps system administrators install libtool libraries so that they can be located and linked into user programs.

Each mode-arg is interpreted as the name of a library directory. Running this command may require superuser privileges, so the `--dry-run' option may be useful.

Uninstall mode

This mode deletes installed libraries (and other files).

The first mode-arg is the name of the program to use to delete files (typically `` `/bin/rm' ``).

The remaining mode-args are either flags for the deletion program (beginning with a `-'), or the names of files to delete.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Integrating libtool with your own packages

This chapter describes how to integrate libtool with your packages so that your users can install hassle-free shared libraries.

Writing `Makefile' rules for libtool

Libtool is fully integrated with Automake (see section `Introduction' in The Automake Manual), starting with Automake version 1.2.

If you want to use libtool in a regular `Makefile' (or `Makefile.in'), you are on your own. If you're not using Automake 1.2, and you don't know how to incorporate libtool into your package you need to do one of the following:

1. Download Automake (version 1.2 or later) from your nearest GNU mirror, install it, and start using it.
2. Learn how to write `Makefile' rules by hand. They're sometimes complex, but if you're clever enough to write rules for compiling your old libraries, then you should be able to figure out new rules for libtool libraries (hint: examine the `Makefile.in' in the `demo' subdirectory of the libtool distribution... note especially that it was automatically generated from the `Makefile.am' by Automake).

Using Automake with libtool

Libtool library support is implemented under the `LTLIBRARIES' primary.

Here are some samples from the Automake `Makefile.am' in the libtool distribution's `demo' subdirectory.

First, to link a program against a libtool library, just use the `program_LDADD' variable:

```
bin_PROGRAMS = hell hell.debug

# Build hell from main.c and libhello.la
hell_SOURCES = main.c
hell_LDADD = libhello.la

# Create an easier-to-debug version of hell.
```


```
hell_debug_SOURCES = main.c
hell_debug_LDADD = libhello.la
hell_debug_LDFLAGS = -static
```

You may use the `program_LDFLAGS' variable to stuff in any flags you want to pass to libtool while linking `program' (such as `-static' to avoid linking uninstalled shared libtool libraries).

Building a libtool library is almost as trivial... note the use of `libhello_la_LDFLAGS' to pass the `-version-info' (see section Library interface versions) option to libtool:

```
# Build a libtool library, libhello.la for installation in libdir.
lib_LTLIBRARIES = libhello.la
libhello_la_SOURCES = hello.c foo.c
libhello_la_LDFLAGS = -version-info 3:12:1
```

The `-rpath' option is passed automatically by Automake, so you should not specify it.

See section `The Automake Manual' in The Automake Manual, for more information.

Configuring libtool

Libtool requires intimate knowledge of your compiler suite and operating system in order to be able to create shared libraries and link against them properly. When you install the libtool distribution, a system-specific libtool script is installed into your binary directory.

However, when you distribute libtool with your own packages (see section Including libtool with your package), you do not always know which compiler suite and operating system are used to compile your package.

For this reason, libtool must be **configured** before it can be used. This idea should be familiar to anybody who has used a GNU `configure` script. `configure` runs a number of tests for system features, then generates the `Makefiles' (and possibly a `config.h' header file), after which you can run `make` and build the package.

Libtool has its own equivalent to the `configure` script, `ltconfig`.

Invoking `ltconfig`

`ltconfig` runs a series of configuration tests, then creates a system-specific `libtool` in the current directory. The `ltconfig` program has the following synopsis:

```
ltconfig [option]... ltmain [host]
```

and accepts the following options:

`--disable-shared'

Create a `libtool` that only builds static libraries.

`--disable-static'

Create a `libtool` that builds only shared libraries if they are available. If only static libraries can be built, then this flag has no effect.

`--help'

Display a help message and exit.

`--no-verify'

Do not use `config.sub` to verify that host is a valid canonical host system name.

`--quiet'

`--silent'

Do not print informational messages when running configuration tests.

`--srcdir=dir'

Look for `config.guess` and `config.sub` in dir.

`--version'

Print `ltconfig` version information and exit.

`--with-gcc'

Assume that the GNU C compiler will be used when invoking the created `libtool` to compile and link object files.

ltmain is the `ltmain.sh` shell script fragment that provides the basic libtool functionality (see section Including libtool with your package).

host is the canonical host system name, which by default is guessed by running `config.guess`.

`ltconfig` also recognizes the following environment variables:

Variable: **CC**

The C compiler that will be used by the generated `libtool`.

Variable: **CFLAGS**

Compiler flags used to generate standard object files.

Variable: **CPPFLAGS**

C preprocessor flags.

Variable: **LD**

The system linker to use (if the generated `libtool` requires one).

Variable: **RANLIB**

Program to use rather than checking for `ranlib`.

Using `ltconfig`

Here is a simple example of using `ltconfig` to configure libtool on my NetBSD/i386 1.2 system:

```
burger$ ./ltconfig ltmain.sh
```


```
checking host system type... i386-unknown-netbsd1.2
checking for ranlib... ranlib
checking for gcc... gcc
checking whether we are using GNU C... yes
checking for gcc option to produce PIC... -fPIC -DPIC
checking for gcc option to statically link programs... -static
checking if ld is GNU ld... no
checking if ld supports shared libraries... yes
checking dynamic linker characteristics... netbsd1.2 ld.so
checking if libtool supports shared libraries... yes
checking whether to build shared libraries... yes
creating libtool
burger$
```

This example shows how to configure `libtool` for cross-compiling to a i486 GNU/Hurd 0.1 system (assuming compiler tools reside in `` `/local/i486-gnu/bin' ``):

```
burger$ export PATH=/local/i486-gnu/bin:$PATH
burger$ ./ltconfig ltmain.sh i486-gnu0.1
checking host system type... i486-unknown-gnu0.1
checking for ranlib... ranlib
checking for gcc... gcc
checking whether we are using GNU C... yes
checking for gcc option to produce PIC... -fPIC -DPIC
checking for gcc option to statically link programs... -static
checking if ld is GNU ld... yes
checking if GNU ld supports shared libraries... yes
checking dynamic linker characteristics... gnu0.1 ld.so
checking if libtool supports shared libraries... yes
checking whether to build shared libraries... yes
creating libtool
burger$
```

The `AM_PROG_LIBTOOL` macro

If you are using GNU Autoconf (or Automake), you should add a call to `AM_PROG_LIBTOOL` to your `` `configure.in' `` file. This macro offers seamless integration between the `configure` script and `ltconfig`:

Macro: **AM_PROG_LIBTOOL**

> Add support for the `--enable-shared' and `--disable-shared' `configure` flags. Invoke `ltconfig` with the correct arguments to configure the package (see section Invoking `ltconfig`).(4)

By default, this macro turns on shared libraries if they are available, and also enables static libraries if they don't conflict with the shared libraries. You can modify these defaults by setting calling either the

AM_DISABLE_SHARED or AM_DISABLE_STATIC macros:

```
# Turn off shared libraries during beta-testing, since they make the
# build process take too long.
AM_DISABLE_SHARED
AM_PROG_LIBTOOL
```

The user may specify a modified form of `--enable-shared' and `--enable-static' to choose whether shared or static libraries are built based on the name of the package. For example, to have shared `bfd' and `gdb' libraries built, but not shared `libg++', you can run all three `configure` scripts as follows:

```
trick$ ./configure --enable-shared=bfd,gdb
```

In general, specifying `--enable-shared=pkgs' is the same as specifying `--enable-shared' to every package named in the pkgs list, and `--disable-shared' to every other package. The `--enable-static=pkgs' flag behaves similarly, except it translates into `--enable-static' and `--disable-static'.

The package name `default' matches any packages which have not set their name in the PACKAGE environment variable.

Macro: **AM_DISABLE_SHARED**

Change the default behaviour for AM_PROG_LIBTOOL to disable shared libraries. The user may still override this default by specifying `--enable-shared'.

Macro: **AM_DISABLE_STATIC**

Change the default behaviour for AM_PROG_LIBTOOL to disable static libraries. The user may still override this default by specifying `--enable-static'.

When you invoke the `libtoolize` program (see section Invoking `libtoolize`), it will tell you where to find a definition of AM_PROG_LIBTOOL. If you use Automake, the `aclocal` program will automatically add AM_PROG_LIBTOOL support to your `configure` script.

Including libtool with your package

In order to use libtool, you need to include the following files with your package:

`config.guess'

Attempt to guess a canonical system name.

`config.sub'

Canonical system name validation subroutine script.

`ltconfig'

Generate a libtool script for a given system.

`ltmain.sh'

A generic script implementing basic libtool functionality.

Note that the libtool script itself should *not* be included with your package. See section Configuring

libtool.

You should use the `libtoolize` program, rather than manually copying these files into your package.

Invoking `libtoolize`

The `libtoolize` program provides a standard way to add libtool support to your package. In the future, it may implement better usage checking, or other features to make libtool even easier to use.

The `libtoolize` program has the following synopsis:

```
libtoolize [option]...
```

and accepts the following options:

`--automake'

Work silently, and assume that Automake libtool support is used. `libtoolize --automake' is used by Automake to add libtool files to your package, when `AM_PROG_LIBTOOL` appears in your `` `configure.in' ``.

`--copy'

`-c'

Copy files from the libtool data directory rather than creating symlinks.

`--dry-run'

`-n'

Don't run any commands that modify the file system, just print them out.

`--force'

`-f'

Replace existing libtool files. By default, `libtoolize` won't overwrite existing files.

`--help'

Display a help message and exit.

`--version'

Print `libtoolize` version information and exit.

If `libtoolize` detects an explicit call to `AC_CONFIG_AUX_DIR` (see section `The Autoconf Manual' in The Autoconf Manual) in your `` `configure.in' ``, it will put the files in the specified directory.

`libtoolize` displays hints for adding libtool support to your package, as well.

Autoconf `.o' macros

The Autoconf package comes with a few macros that run tests, then set a variable corresponding to the name of an object file. Sometimes it is necessary to use corresponding names for libtool objects.

Here are the names of variables that list libtool objects:

Variable: **LTALLOCA**

Substituted by `AC_FUNC_ALLOCA` (see section `The Autoconf Manual' in The Autoconf Manual). Is either empty, or contains `alloca.lo'.

Variable: **LTLIBOBJS**

Substituted by `AC_REPLACE_FUNCS` (see section `The Autoconf Manual' in The Autoconf Manual), and a few other functions.

Unfortunately, the most recent version of Autoconf (2.12, at the time of this writing) does not have any way for libtool to provide support for these variables. So, if you depend on them, use the following code immediately before the call to `AC_OUTPUT` in your `` `configure.in' ``:

```
LTLIBOBJS=`echo "$LIBOBJS" | sed 's/\.o/.lo/g'`
AC_SUBST(LTLIBOBJS)
LTALLOCA=`echo "$ALLOCA" | sed 's/\.o/.lo/g'`
AC_SUBST(LTALLOCA)
AC_OUTPUT(...)
```

Static-only libraries

When you are developing a package, it is often worthwhile to configure your package with the `--disable-shared' flag, or to override the defaults for `AM_PROG_LIBTOOL` by using the `AM_DISABLE_SHARED` Autoconf macro (see section The `AM_PROG_LIBTOOL` macro). This prevents libtool from building shared libraries, which has several advantages:

- compilation is twice as fast, which can speed up your development cycle
- debugging is easier because you don't need to deal with any complexities added by shared libraries
- you can see how libtool behaves on static-only platforms

You may want to put a small note in your package `` `README' `` to let other developers know that `--disable-shared' can save them time. The following example note is taken from the GIMP(5) distribution `` `README' ``:

```
The GIMP uses GNU Libtool in order to build shared libraries on a
variety of systems. While this is very nice for making usable
binaries, it can be a pain when trying to debug a program. For that
reason, compilation of shared libraries can be turned off by
specifying the `--disable-shared' option to `configure'.
```

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Library interface versions

The most difficult issue introduced by shared libraries is that of creating and resolving runtime dependencies. Dependencies on programs and libraries are often described in terms of a single name, such as `sed`. So, I may say "libtool depends on sed," and that is good enough for most purposes.

However, when an interface changes regularly, we need to be more specific: "Gnus 5.1 requires Emacs 19.28 or above." Here, the description of an interface consists of a name, and a "version number."

Even that sort of description is not accurate enough for some purposes. What if Emacs 20 changes enough to break Gnus 5.1?

The same problem exists in shared libraries: we require a formal version system to describe the sorts of dependencies that programs have on shared libraries, so that the dynamic linker can guarantee that programs are linked only against libraries that provide the interface they require.

What are library interfaces?

Interfaces for libraries may be any of the following (and more):

- global variables: both names and types
- global functions: argument types and number, return types, and function names
- standard input, standard output, standard error, and file formats
- sockets, pipes, and other inter-process communication protocol formats

Note that static functions do not count as interfaces, because they are not directly available to the user of the library.

Libtool's versioning system

Libtool has its own formal versioning system. It is not as flexible as some, but it is definitely the simplest of the more powerful versioning systems.

Think of a library as exporting several sets of interfaces, arbitrarily represented by integers. When a program is linked against a library, it may use any subset of those interfaces.

Libtool's description of the interfaces that a program uses is simple: it encodes the least and the greatest interface numbers in the resulting binary (first-interface, last-interface).

The dynamic linker is guaranteed that if a library supports *every* interface number between first-interface and last-interface, then the program can be relinked against that library.

Note that this can cause problems because libtool's compatibility requirements are actually stricter th necessary.

Say `libhello' supports interfaces 5, 16, 17, 18, and 19, and that libtool is used to link `test' against `libhello'.

Libtool encodes the numbers 5 and 19 in `test', and the dynamic linker will only link `test' against libraries that support *every* interface between 5 and 19. So, the dynamic linker refuses to link `test' against `libhello'!

In order to eliminate this problem, libtool only allows libraries to declare consecutive interface numbers. So, `libhello' can declare at most that it supports interfaces 16 through 19. Then, the dynamic linker will link `test' against `libhello'.

So, libtool library versions are described by three integers:

current
: The most recent interface number that this library implements.

revision
: The implementation number of the current interface.

age
: The difference between the newest and oldest interfaces that this library implements. In other words, the library implements all the interface numbers in the range from number `current - age` to `current`.

If two libraries have identical current and age numbers, then the dynamic linker chooses the library with the greater revision number.

Updating library version information

If you want to use libtool's versioning system, then you must specify the version information to libtool using the `-version-info' flag during link mode (see section Link mode).

This flag accepts an argument of the form `current[:revision[:age]]'. So, passing `-version-info 3:12:1' sets current to 3, revision to 12, and age to 1.

If either revision or age are omitted, they default to 0. Also note that age must be less than or equal to the current interface number.

Here are a set of rules to help you update your library version information:

1. Start with version information of `0:0:0' for each libtool library.
2. Update the version information only immediately before a public release of your software. More frequent updates are unnecessary, and only guarantee that the current interface number gets larger faster.
3. If the library source code has changed at all since the last update, then increment revision (`c:r:a' becomes `c:r+1:a').

4. If any interfaces have been added, removed, or changed since the last update, increment current, and set revision to 0.
5. If any interfaces have been added since the last public release, then increment age.
6. If any interfaces have been removed since the last public release, then set age to 0.

Never try to set the interface numbers so that they correspond to the release number of your package. This is an abuse that only fosters misunderstanding of the purpose of library versions. Instead, use the `-release' flag (see section Managing release information), but be warned that every release of your package will not be binary compatibility with any other release.

Managing release information

Often, people want to encode the name of the package release into the shared library so that it is obvious to the user which package their programs are linked against. This convention is used especially on Linux:

```
trick$ ls /usr/lib/libbfd*
/usr/lib/libbfd.a           /usr/lib/libbfd.so.2.7.0.2
/usr/lib/libbfd.so
trick$
```

On `trick', `/usr/lib/libbfd.so' is just a symbolic link to `/usr/lib/libbfd.so.2.7.0.2', which was distributed as a part of `binutils-2.7.0.2'.

Unfortunately, this convention conflicts directly with libtool's idea of library interface versions, because the library interface rarely changes at the same time that the release number does, and the library suffix is never the same across all platforms.

So, in order to accomodate both views, you can use the `-release' flag in order to set release information for libraries which you do not want to use `-version-info'. For the `libbfd' example, the next release which uses libtool should be built with `-release 2.9.0', which will produce the following files on Linux:

```
trick$ ls /usr/lib/libbfd*
/usr/lib/libbfd-2.9.0.so.0          /usr/lib/libbfd.so
/usr/lib/libbfd-2.9.0.so.0.0.0      /usr/lib/libbfd.a
trick$
```

In this case, `/usr/lib/libbfd.so' is a symbolic link to `/usr/lib/libbfd-2.9.0.so.0.0.0'. This makes it obvious that the user is dealing with `binutils-2.9.0', without compromising libtool's idea of interface versions.

Note that this option actually causes a modification of the library name, so do not use it if unless you want to break binary compatibility with any past library releases. In general, you should only use `-release' for libraries whose interfaces change very frequently.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Tips for interface design

Writing a good library interface takes a lot of practice and thorough understanding of the problem that the library is intended to solve.

If you design a good interface, it won't have to change often, you won't have to keep updating documentation, and users won't have to keep relearning how to use the library.

Here is a brief list of tips for library interface design, which may help you in your exploits:

Plan ahead

> Try to make every interface truly minimal, so that you won't need to delete entry points very often.

Avoid interface changes

> Some people love redesigning and changing entry points just for the heck of it (note: *renaming* a function is considered changing an entry point). Don't be one of those people. If you must redesign an interface, then try to leave compatibility functions behind so that users don't need to rewrite their existing code.

Use opaque data types

> The fewer data type definitions a library user has access to, the better. If possible, design your functions to accept a generic pointer (which you can cast to an internal data type), and provide access functions rather than allowing the library user to directly manipulate the data. That way, you have the freedom to change the data structures without changing the interface. This is essentially the same thing as using abstract data types and inheritance in an object-oriented system.

Use header files

> If you are careful to document each of your library's global functions and variables in header files, and include them in your library source files, then the compiler will let you know if you make any interface changes by accident (see section Writing C header files).

Use the `static` keyword (or equivalent) whenever possible

> The fewer global functions your library has, the more flexibility you'll have in changing them. Static functions and variables may change forms as often as you like... your users cannot access them, so they aren't interface changes.

Writing C header files

Writing portable C header files can be difficult, since they may be read by different types of compilers:

C++ compilers

> C++ compilers require that functions be declared with full prototypes, since C++ is more strongly typed than C. C functions and variables also need to be declared with the `extern "C"` directive,

so that the names aren't mangled. See section Writing libraries for C++, for other issues relevant to using C++ with libtool.

ANSI C compilers

ANSI C compilers are not as strict as C++ compilers, but functions should be prototyped to avoid unnecessary warnings when the header file is `#included`.

non-ANSI C compilers

Non-ANSI compilers will report errors if functions are prototyped.

These complications mean that your library interface headers must use some C preprocessor magic in order to be usable by each of the above compilers.

`foo.h' in the `demo' subdirectory of the libtool distribution serves as an example for how to write a header file that can be safely installed in a system directory.

Here are the relevant portions of that file:

```
/* __BEGIN_DECLS should be used at the beginning of your declarations,
   so that C++ compilers don't mangle their names.  Use __END_DECLS at
   the end of C declarations. */
#undef __BEGIN_DECLS
#undef __END_DECLS
#ifdef __cplusplus
# define __BEGIN_DECLS extern "C" {
# define __END_DECLS }
#else
# define __BEGIN_DECLS /* empty */
# define __END_DECLS /* empty */
#endif

/* __P is a macro used to wrap function prototypes, so that compilers
   that don't understand ANSI C prototypes still work, and ANSI C
   compilers can issue warnings about type mismatches. */
#undef __P
#if defined (__STDC__) || defined (_AIX) \
        || (defined (__mips) && defined (_SYSTYPE_SVR4)) \
        || defined(WIN32) || defined(__cplusplus)
# define __P(protos) protos
#else
# define __P(protos) ()
#endif
```

These macros are used in `foo.h' as follows:

```
#ifndef _FOO_H_
#define _FOO_H_ 1
```


```
/* The above macro definitions. */
...

__BEGIN_DECLS
int foo __P((void));
int hello __P((void));
__END_DECLS

#endif /* !_FOO_H_ */
```

Note that the `#ifndef _FOO_H_' prevents the body of `foo.h' from being read more than once in a given compilation.

Feel free to copy the definitions of __P, __BEGIN_DECLS, and __END_DECLS into your own headers. Then, you may use them to create header files that are valid for C++, ANSI, and non-ANSI compilers.

Do not be naive about writing portable code. Following the tips given above will help you miss the most obvious problems, but there are definitely other subtle portability issues. You may need to cope with some of the following issues:

- Pre-ANSI compilers do not always support the `void *` generic pointer type, and so need to use `char *` in its place.
- The `const` and `signed` keywords are not supported by some compilers, especially pre-ANSI compilers.
- The `long double` type is not supported by many compilers.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

TOC

Inter-library dependencies

By definition, every shared library system provides a way for executables to depend on libraries, so that symbol resolution is deferred until runtime.

An **inter-library dependency** is one in which a library depends on other libraries. For example, if the libtool library `libhello' uses the `cos(3)` function, then it has an inter-library dependency on `libm', the math library that implements `cos(3)`.

Some shared library systems provide this feature in an internally-consistent way: these systems allow chains of dependencies of potentially infinite length.

However, most shared library systems are restricted in that they only allow a single level of dependencies. In these systems, programs may depend on shared libraries, but shared libraries may not depend on other shared libraries.

In any event, libtool provides a simple mechanism for you to declare inter-library dependencies: for every library `libname' that your own library depends on, simply add a corresponding `-lname` option to the link line when you create your library.(6) To make an example of our `libhello' that depends on `libm':

```
burger$ libtool gcc -g -O -o libhello.la foo.lo hello.lo \
                -rpath /usr/local/lib -lm
burger$
```

In order to link a program against `libhello', you need to specify the same `-l' options, in order to guarantee that all the required libraries are found. This restriction is only necessary to preserve compatibility with static library systems and simple dynamic library systems.

Some platforms, such as AIX, do not even allow you this flexibility. In order to build a shared library, it must be entirely self-contained (that is, have no references to external symbols), and you need to specify the -no-undefined flag to allow a shared library to be built. By default, libtool builds only static libraries on these kinds of platforms.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Dlopened modules

It can sometimes be confusing to discuss **dynamic linking**, because the term is used to refer to two different concepts:

1. Compiling and linking a program against a shared library, which is resolved automatically at run time by the dynamic linker. In this process, dynamic linking is transparent to the application.
2. The application calling functions such as `dlopen(3)`,(7) which load arbitrary, user-specified modules at runtime. This type of dynamic linking is explicitly controlled by the application.

To mitigate confusion, this manual refers to the second type of dynamic linking as **dlopening** a module.

The main benefit to dlopening object modules is the ability to access compiled object code to extend your program, rather than using an interpreted language. In fact, dlopen calls are frequently used in language interpreters to provide an efficient way to extend the language.

As of version 1.2, libtool provides experimental support for dlopened modules, which does not radically simplify the development of dlopening applications. However, this support is designed to be a portable foundation for generic, higher-level dlopen functions.

This chapter discusses the preliminary support that libtool offers, and how you as a dlopen application developer might use libtool to generate dlopen-accessible modules. It is important to remember that these are experimental features, and not to rely on them for easy answers to the problems associated with dlopened modules.

Building modules to dlopen

On some operating systems, a program symbol must be specially declared in order to be dynamically resolved with the `dlsym(3)` (or equivalent) function.

Libtool provides the `-export-dynamic' link flag (see section Link mode), which does this declaration. You need to use this flag if you are linking an application program that dlopens other modules or a libtool library that will also be dlopened.

For example, if we wanted to build a shared library, `libhello', that would later be dlopened by an application, we would add `-export-dynamic' to the other link flags:

```
burger$ libtool gcc -export-dynamic -o libhello.la foo.lo \
                hello.lo -rpath /usr/local/lib -lm
burger$
```

Another situation where you would use `-export-dynamic' is if symbols from your *executable* are needed to satisfy unresolved references in a library you want to dlopen. In this case, you should use

`-export-dynamic' while linking the executable that calls dlopen:

```
burger$ libtool gcc -export-dynamic -o hell-dlopener main.o
burger$
```

Dlpreopening

Libtool provides special support for dlopening libtool object and libtool library files, so that their symbols can be resolved *even on platforms without any* `dlopen(3)` *and* `dlsym(3)` *functions.*.

Consider the following alternative ways of loading code into your program, in order of increasing "laziness":

1. Linking against object files that become part of the program executable, whether or not they are referenced. If an object file cannot be found, then the linker refuses to create the executable.
2. Declaring a static library to the linker, so that it is searched at link time in order to satisfy any undefined references in the above object files. If the static library cannot be found, then the linker refuses to link the executable.
3. Declaring a shared library to the runtime linker, so that it is searched at runtime in order to satisfy any undefined references in the above files. If the shared library cannot be found, then the dynamic linker aborts the program before it runs.
4. Dlopening a module, so that the application can resolve its own, dynamically-computed references. If there is an error opening the module, or the module is not found, then the application can recover without crashing.

Libtool emulates `-export-dynamic' on static platforms by linking objects into the program at compile time, and creating data structures that represent the program's symbol table.

In order to use this feature, you must declare the objects you want your application to dlopen by using the `-dlopen' or `-dlpreopen' flags when you link your program (see section Link mode).

Structure: **dld_symbol** *name address*

The name attribute is a 0-terminated character string of the symbol name, such as `"fprintf"`. The address attribute is a generic pointer to the appropriate object, which is `&fprintf` in this example.

Variable: dld_symbol * **dld_preloaded_symbols**

An array of dld_symbol structures, representing all the preloaded symbols linked into the program. The last element has a name of `0`.

Variable: int **dld_preloaded_symbol_count**

The number of elements in dld_preloaded_symbols, if it is sorted in ascending order by name. Otherwise, `-1`, to indicate that the application needs to sort and count dld_preloaded_symbols itself, or search it linearly.

Some compilers may allow identifiers which are not valid in ANSI C, such as dollar signs. Libtool only recognizes valid ANSI C symbols (an initial ASCII letter or underscore, followed by zero or more ASCII

letters, digits, and underscores), so non-ANSI symbols will not appear in dld_preloaded_symbols.

Finding the correct name to dlopen

After a library has been linked with `-export-dynamic', it can be dlopened. Unfortunately, because of the variation in library names, your package needs to determine the correct file to dlopen.

The most straightforward and flexible implementation is to determine the name at runtime, by finding the installed `.la' file, and searching it for the following lines:

```
# The name that we can dlopen(3).
dlname='dlname'
```

If dlname is empty, then the library cannot be dlopened. Otherwise, it gives the dlname of the library. So, if the library was installed as `/usr/local/lib/libhello.la', and the dlname was `libhello.so.3', then `/usr/local/lib/libhello.so.3' should be dlopened.

If your program uses this approach, then it should search the directories listed in the LD_LIBRARY_PATH(8) environment variable, as well as the directory where libraries will eventually be installed. Searching this variable (or equivalent) will guarantee that your program can find its dlopened modules, even before installation, provided you have linked them using libtool.

Unresolved dlopen issues

The following problems are not solved by using libtool's dlopen support:

- Dlopen functions are generally only available on shared library platforms. If you want your package to be portable to static platforms, you have to develop your own alternatives to dlopening dynamic code. Most reasonable solutions involve writing wrapper functions for the `dlopen(3)` family, which do package-specific tricks when dlopening is unsupported or not available on a given platform.
- There are major differences in implementations of the `dlopen(3)` family of functions. Some platforms do not even use the same function names (notably HP-UX, with its `shl_load(3)' family).
- The application developer must write a custom search function in order to discover the correct module filename to supply to `dlopen(3)`.

Each of these limitations will be addressed in GNU DLD 4.(9)

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Using libtool with other languages

Libtool was first implemented in order to add support for writing shared libraries in the C language. However, over time, libtool is being integrated with other languages, so that programmers are free to reap the benefits of shared libraries in their favorite programming language.

This chapter describes how libtool interacts with other languages, and what special considerations you need to make if you do not use C.

Writing libraries for C++

Creating libraries of C++ code is a fairly straightforward process, and differs from C code in only two ways:

1. Because of name mangling, C++ libraries are only usable by the C++ compiler that created them. This decision was made by the designers of C++ in order to protect users from conflicting implementations of features such as constructors, exception handling, and RTTI.
2. On some systems, notably SunOS 4, the dynamic linker does not call non-constant initializers. This can lead to hard-to-pinpoint bugs in your library. GCC 2.7 and later versions work around this problem, but previous versions and other compilers do not.

This second issue is complex. Basically, you should avoid any global or static variable initializations that would cause an "initializer element is not constant" error if you compiled them with a standard C compiler.

There are other ways of working around this problem, but they are beyond the scope of this manual.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Maintenance notes for libtool

This chapter contains information that the libtool maintainer finds important. It will be of no use to you unless you are considering porting libtool to new systems, or writing your own libtool.

Porting libtool to new systems

To port libtool to a new system, you'll generally need the following information:

man pages for `ld(1)` and `cc(1)`

: These generally describe what flags are used to generate PIC, to create shared libraries, and to link against only static libraries. You may need to follow some cross references to find the information that is required.

man pages for `ld.so(8)`, `rtld(8)`, or equivalent

: These are a valuable resource for understanding how shared libraries are loaded on the system.

man page for `ldconfig(8)`, or equivalent

: This page usually describes how to install shared libraries.

output from ls -l /lib /usr/lib

: This shows the naming convention for shared libraries on the system, including which names should be symbolic links.

any additional documentation

: Some systems have special documentation on how to build and install shared libraries.

Tested platforms

This table describes when libtool was last known to be tested on platforms where it claims to support shared libraries:

canonical host name	compiler	libtool release	results
alpha-dec-osf3.2	cc	0.8	ok
alpha-dec-osf3.2	gcc	0.8	ok
alpha-dec-osf4.0	cc	1.0f	ok
alpha-dec-osf4.0	gcc	1.0f	ok
alpha-unknown-linux-gnu	gcc	0.9h	ok
hppa1.1-hp-hpux9.07	cc	1.0f	ok


```
hppa1.1-hp-hpux9.07           gcc       1.0f      ok
hppa1.1-hp-hpux10.10          cc        0.9h      ok
hppa1.1-hp-hpux10.10          gcc       0.9h      ok
i386-unknown-freebsd2.1.5     gcc       0.5       ok
i386-unknown-gnu0.0           gcc       0.5       ok
i386-unknown-netbsd1.2        gcc       0.9g      ok
i586-pc-linux-gnulibc1        gcc       1.0i      ok
i586-pc-linux-gnu             gcc       1.0i      ok
mips-sgi-irix5.2              gcc       1.0i      ok
mips-sgi-irix5.3              cc        0.8       ok
mips-sgi-irix5.3              gcc       0.8       ok
mips-sgi-irix6.2              cc        0.9       ok
mips-sgi-irix6.3              cc        1.0f      ok
mips-sgi-irix6.3              gcc       1.0i      ok
mips-sgi-irix6.3              irix5-gcc 1.0f      ok
mipsel-unknown-openbsd2.1     gcc       1.0       ok
powerpc-ibm-aix4.1.4.0        xlc       1.0i      ok
powerpc-ibm-aix4.1.4.0        gcc       1.0       ok
rs6000-ibm-aix3.2.5           xlc       1.0i      ok
rs6000-ibm-aix3.2.5           gcc       1.0i      ok*
sparc-sun-linux-gnu2.1.23     gcc       0.9h      ok
sparc-sun-sunos4.1.3          gcc       1.0i      ok
sparc-sun-sunos4.1.4          cc        1.0f      ok
sparc-sun-sunos4.1.4          gcc       1.0f      ok
sparc-sun-solaris2.4          cc        1.0a      ok
sparc-sun-solaris2.4          gcc       1.0a      ok
sparc-sun-solaris2.5          cc        1.0f      ok
sparc-sun-solaris2.5          gcc       1.0i      ok
sparc-sun-solaris2.6          gcc       1.0i      ok
-------------------------------------------------------

* Some versions of GCC's collect2 linker program cannot link trivial
static binaries on AIX 3.  For these configurations, libtool's `-static'
flag has no effect.
```

Platform quirks

This section is dedicated to the sanity of the libtool maintainer. It describes the programs that libtool uses, how they vary from system to system, and how to test for them.

Because libtool is a shell script, it is difficult to understand just by reading it from top to bottom. This section helps show why libtool does things a certain way. After reading it, then reading the scripts themselves, you should have a better sense of how to improve libtool, or write your own.

References

The following is a list of valuable documentation references:

- SGI's IRIX Manual Pages, `http://techpubs.sgi.com/cgi-bin/infosrch.cgi?cmd=browse&db=man`.
- Sun's free service area (`http://www.sun.com/service/online/free.html`) and documentation server (`http://docs.sun.com/`).

Compilers

The only compiler characteristics that affect libtool are the flags needed (if any) to generate PIC objects. In general, if a C compiler supports certain PIC flags, then any derivative compilers support the same flags. Until there are some noteworthy exceptions to this rule, this section will document only C compilers.

The following C compilers have standard command line options, regardless of the platform:

`gcc`

This is the GNU C compiler, which is also the system compiler for many free operating systems (FreeBSD, GNU/Hurd, Linux/GNU, Lites, NetBSD, and OpenBSD, to name a few). The `-fpic' or `-fPIC' flags can be used to generate position-independent code. `-fPIC' is guaranteed to generate working code, but the code is slower on m68k, m88k, and Sparc chips. However, using `-fpic' on those chips imposes arbitrary size limits on the shared libraries.

The rest of this subsection lists compilers by the operating system that they are bundled with:

`aix3*`

`aix4*`

AIX compilers have no PIC flags, since AIX has been ported only to PowerPC and RS/6000 chips. (10)

`hpux10*`

Use `+Z' to generate PIC.

`osf3*`

Digital/UNIX 3.x does not have PIC flags, at least not on the PowerPC platform.

`solaris2*`

Use `-KPIC' to generate PIC.

`sunos4*`

Use `-PIC' to generate PIC.

Reloadable objects

On all known systems, a reloadable object can be created by running ld -r -o output.o input1.o input2.o. This reloadable object may be treated as exactly equivalent to other objects.

Archivers

On all known systems, building a static library can be accomplished by running ar cru libname.a obj1.o obj2.o ..., where the `.a' file is the output library, and each `.o' file is an object file.

On all known systems, if there is a program named `ranlib`, then it must be used to "bless" the created library before linking against it, with the ranlib libname.a command.

`libtool` script contents

The `libtool` script is generated by `ltconfig` (see section Configuring libtool). Ever since libtool version 0.7, this script simply sets shell variables, then sources the libtool backend, `ltmain.sh`.

Here is a listing of each of these variables, and how they are used within `ltmain.sh`:

Variable: **AR**

The name of the system library archiver.

Variable: **CC**

The name of the C compiler used to configure libtool.

Variable: **LD**

The name of the linker that libtool should use internally for reloadable linking and possibly shared libraries.

Variable: **LTCONFIG_VERSION**

This is set to the version number of the `ltconfig` script, to prevent mismatches between the configuration information in `libtool`, and how that information is used in `ltmain.sh`.

Variable: **NM**

The name of a BSD-compatible `nm` program, which produces listings of global symbols in one the following formats:

```
address C global-variable-name
address D global-variable-name
address T global-function-name
```

Variable: **RANLIB**

Set to the name of the ranlib program, if any.

Variable: **allow_undefined_flag**

The flag that is used by `archive_cmds' in order to declare that there will be unresolved symbols in the resulting shared library. Empty, if no such flag is required. Set to `unsupported' if there is no way to generate a shared library with references to symbols that aren't defined in that library.

Variable: **archive_cmds**

Variable: **old_archive_cmds**

Commands used to create shared and static libraries, respectively.

Variable: **build_libtool_libs**

Whether libtool should build shared libraries on this system. Set to `yes' or `no'.

Variable: **build_old_libs**

Whether libtool should build static libraries on this system. Set to `yes' or `no'.

Variable: **echo**

An `echo(1)` program which does not interpret backslashes as an escape character.

Variable: **export_dynamic_flag_spec**

Compiler link flag that allows a dlopened shared library to reference symbols that are defined in the program.

Variable: **finish_cmds**

Commands to tell the dynamic linker how to find shared libraries in a specific directory.

Variable: **finish_eval**

Same as finish_cmds, except the commands are not displayed.

Variable: **global_symbol_pipe**

A pipeline that takes the output of NM, and produces a listing of raw symbols followed by their C names. For example:

```
$ $NM | $global_symbol_pipe
symbol1 C-symbol1
symbol2 C-symbol2
symbol3 C-symbol3
...
$
```

Variable: **hardcode_action**

Either `immediate' or `relink', depending on whether shared library paths can be hardcoded into executables before they are installed, or if they need to be relinked.

Variable: **hardcode_direct**

Set to `yes' or `no', depending on whether the linker hardcodes directories if a library is directly specified on the command line (such as `dir/libname.a').

Variable: **hardcode_libdir_flag_spec**

Flag to hardcode a libdir variable into a binary, so that the dynamic linker searches libdir for shared libraries at runtime.

Variable: **hardcode_libdir_separator**

If the compiler only accepts a single hardcode_libdir_flag, then this variable contains the string that should separate multiple arguments to that flag.

Variable: **hardcode_minus_L**

Set to `yes' or `no', depending on whether the linker hardcodes directories specified by `-L' flags into the resulting executable.

Variable: **hardcode_shlibpath_var**

Set to `yes' or `no', depending on whether the linker hardcodes directories by writing the contents of `$shlibpath_var' into the resulting executable. Set to `unsupported' if directories specified by

`$shlibpath_var' are searched at run time, but not at link time.

Variable: **host**

Variable: **host_alias**

For information purposes, set to the specified and canonical names of the system that libtool was configured for.

Variable: **libname_spec**

The format of a library name prefix. On all Unix systems, static libraries are called `libname.a', but on some systems (such as OS/2 or MS-DOS), the library is just called `name.a'.

Variable: **library_names_spec**

A list of shared library names. The first is the name of the file, the rest are symbolic links to the file. The name in the list is the file name that the linker finds when given `-lname'.

Variable: **link_static_flag**

Linker flag (passed through the C compiler) used to prevent dynamic linking.

Variable: **no_builtin_flag**

Compiler flag to disable builtin functions that conflict with declaring external global symbols as `char`.

Variable: **no_undefined_flag**

The flag that is used by `archive_cmds' in order to declare that there will be no unresolved symbols in the resulting shared library. Empty, if no such flag is required.

Variable: **pic_flag**

Any additional compiler flags for building library object files.

Variable: **postinstall_cmds**

Variable: **old_postinstall_cmds**

Commands run after installing a shared or static library, respectively.

Variable: **reload_cmds**

Variable: **reload_flag**

Commands to create a reloadable object.

Variable: **runpath_var**

The environment variable that tells the linker which directories to hardcode in the resulting executable.

Variable: **shlibpath_var**

The environment variable that tells the dynamic linker where to find shared libraries.

Variable: **soname_spec**

The name coded into shared libraries, if different from the real name of the file.

Variable: **version_type**

The library version numbering type. One of `libtool', `linux', `osf', `sunos', or `none'.

Variable: **wl**

The C compiler flag that allows libtool to pass a flag directly to the linker. Used as: `${wl}some-flag'.

Variables ending in `_cmds' or `_eval' contain a semicolon-separated list of commands that are `eval'ed one after another. If any of the commands return a nonzero exit status, libtool generally exits with an error message.

Variables ending in `_spec' are `eval'ed before being used by libtool.

Go to the first, previous, next, last section, table of contents.

Go to the first, previous, next, last section, table of contents.

Index

.

a

b

c

d

e

f

g

h

i

l

m

n

o

p

r

s

t

u

v

w

Go to the first, previous, next, last section, table of contents.

GNU Libtool

For version 1.2, 10 March 1998

Gordon Matzigkeit

---

(1)

Remember that we need to add -lm to the link command line because `foo.c' uses the cos(3) math library function. See section Using libtool.

(2)

However, you should never use `-L' or `-l' flags to link against an uninstalled libtool library. Just specify the relative path to the `.la' file, such as `../intl/libintl.la'. This is a design decision to eliminate any ambiguity when linking against uninstalled shared libraries.

(3)

Don't accidentally strip the libraries, though, or they will be unusable.

(4)

AM_PROG_LIBTOOL requires that you define the `Makefile' variable top_builddir in your `Makefile.in'. Automake does this automatically, but Autoconf users should set it to the relative path to the top of your build directory (`../..', for example).

(5)

GNU Image Manipulation Program, for those who haven't taken the plunge. See http://www.gimp.org/.

(6)

Unfortunately, as of libtool version 1.2, there is no way to specify inter-library dependencies on libtool libraries that have not yet been installed.

(7)

HP-UX, to be different, uses a function named shl_load(3).

(8)

LIBPATH on AIX, and SHLIB_PATH on HP-UX.

(9)

Unfortunately, the DLD maintainer is also the libtool maintainer, so time spent on one of these projects takes time away from the other. When libtool is reasonably stable, DLD 4 development will proceed.

(10)

All code compiled for the PowerPC and RS/6000 chips (`powerpc-*-*`, `powerpcle-*-*`, and `rs6000-*-*`) is position-independent, regardless of the operating system or compiler suite. So, "regular objects" can be used to build shared libraries on these systems and no special PIC compiler flags are required.

This document was generated on 20 May 1999 using the texi2html translator version 1.51a.

NAME

perl - Practical Extraction and Report Language

SYNOPSIS

For ease of access, the Perl manual has been split up into a number of sections:

- perl Perl overview (this section)
- perldata Perl data structures
- perlsyn Perl syntax
- perlop Perl operators and precedence
- perlre Perl regular expressions
- perlrun Perl execution and options
- perlfunc Perl builtin functions
- perlvar Perl predefined variables
- perlsub Perl subroutines
- perlmod Perl modules
- perlref Perl references and nested data structures
- perlobj Perl objects
- perlbot Perl OO tricks and examples
- perldebug Perl debugging
- perldiag Perl diagnostic messages
- perlform Perl formats
- perlipc Perl interprocess communication
- perlsec Perl security
- perltrap Perl traps for the unwary
- perlstyle Perl style guide
- perlapi Perl application programming interface
- perlguts Perl internal functions for those doing extensions
- perlcall Perl calling conventions from C
- perlovl Perl overloading semantics

- perlbook Perl book information

(If you're intending to read these straight through for the first time, the suggested order will tend to reduce the number of forward references.)

If something strange has gone wrong with your program and you're not sure where you should look for help, try the **-w** switch first. It will often point out exactly where the trouble is.

DESCRIPTION

Perl is an interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many system management tasks. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). It combines (in the author's opinion, anyway) some of the best features of C, **sed** , **awk** , and **sh** , so people familiar with those languages should have little difficulty with it. (Language historians will also note some vestiges of **csh** , Pascal, and even BASIC-PLUS.) Expression syntax corresponds quite closely to C expression syntax. Unlike most Unix utilities, Perl does not arbitrarily limit the size of your data--if you've got the memory, Perl can slurp in your whole file as a single string. Recursion is of unlimited depth. And the hash tables used by associative arrays grow as necessary to prevent degraded performance. Perl uses sophisticated pattern matching techniques to scan large amounts of data very quickly. Although optimized for scanning text, Perl can also deal with binary data, and can make dbm files look like associative arrays (where dbm is available). Setuid Perl scripts are safer than C programs through a dataflow tracing mechanism which prevents many stupid security holes. If you have a problem that would ordinarily use **sed** or **awk** or **sh** , but it exceeds their capabilities or must run a little faster, and you don't want to write the silly thing in C, then Perl may be for you. There are also translators to turn your **sed** and **awk** scripts into Perl scripts.

But wait, there's more...

Perl version 5 is nearly a complete rewrite, and provides the following additional benefits:

*** Many usability enhancements**

It is now possible to write much more readable Perl code (even within regular expressions). Formerly cryptic variable names can be replaced by mnemonic identifiers. Error messages are more informative, and the optional warnings will catch many of the mistakes a novice might make. This cannot be stressed enough. Whenever you get mysterious behavior, try the **-w** switch!!! Whenever you don't get mysterious behavior, try using **-w** anyway.

*** Simplified grammar**

The new yacc grammar is one half the size of the old one. Many of the arbitrary grammar rules have been regularized. The number of reserved words has been cut by 2/3. Despite this, nearly all old Perl scripts will continue to work unchanged.

*** Lexical scoping**

Perl variables may now be declared within a lexical scope, like "auto" variables in C. Not only is this more efficient, but it contributes to better privacy for "programming in the large".

*** Arbitrarily nested data structures**

Any scalar value, including any array element, may now contain a reference to any other variable or subroutine. You can easily create anonymous variables and subroutines. Perl manages your reference counts for you.

*** Modularity and reusability**

The Perl library is now defined in terms of modules which can be easily shared among various packages. A package may choose to import all or a portion of a module's published interface. Pragmas (that is, compiler directives) are defined and used by the same mechanism.

**** Object-oriented programming***

A package can function as a class. Dynamic multiple inheritance and virtual methods are supported in a straightforward manner and with very little new syntax. Filehandles may now be treated as objects.

*** Embeddible and Extensible**

Perl may now be embedded easily in your C or C++ application, and can either call or be called by your routines through a documented interface. The XS preprocessor is provided to make it easy to glue your C or C++ routines into Perl. Dynamic loading of modules is supported.

*** POSIX compliant**

A major new module is the POSIX module, which provides access to all available POSIX routines and definitions, via object classes where appropriate.

*** Package constructors and destructors**

The new BEGIN and END blocks provide means to capture control as a package is being compiled, and after the program exits. As a degenerate case they work just like awk's BEGIN and END when you use the **-p** or **-n** switches.

*** Multiple simultaneous DBM implementations**

A Perl program may now access DBM, NDBM, SDBM, GDBM, and Berkeley DB files from the same script simultaneously. In fact, the old dbmopen interface has been generalized to allow any variable to be tied to an object class which defines its access methods.

*** Subroutine definitions may now be autoloaded**

In fact, the AUTOLOAD mechanism also allows you to define any arbitrary semantics for undefined subroutine calls. It's not just for autoloading.

*** Regular expression enhancements**

You can now specify non-greedy quantifiers. You can now do grouping without creating a backreference. You can now write regular expressions with embedded whitespace and comments for readability. A consistent extensibility mechanism has been added that is upwardly compatible with all old regular expressions.

Ok, that's *definitely* enough hype..

ENVIRONMENT

HOME

Used if chdir has no argument.

LOGDIR

Used if chdir has no argument and HOME is not set.

PATH

Used in executing subprocesses, and in finding the script if **-S** is used.

PERL5LIB

A colon-separated list of directories in which to look for Perl library files before looking in the standard library and the current directory. If PERL5LIB is not defined, PERLLIB is used.

PERL5DB

The command used to get the debugger code. If unset, uses

BEGIN { require 'perl5db.pl' }

PERLLIB

A colon-separated list of directories in which to look for Perl library files before looking in the standard library and the current directory. If PERL5LIB is defined, PERLLIB is not used.

Apart from these, Perl uses no other environment variables, except.to make them available to the script being executed, and to child processes. However, scripts running setuid would do well to execute the following lines before doing anything else, just to keep people honest:

$ENV{'PATH'} = '/bin:/usr/bin'; # or whatever you need $ENV{'SHELL'} = '/bin/sh' if defined $ENV{'SHELL'}; $ENV{'IFS'} = '' if defined $ENV{'IFS'};

AUTHOR

Larry Wall< *lwall* ***@netlabs*** *.com.* , with the help of oodles of other folks.

FILES

"/tmp/perl-e$$" temporary file for -e commands " **@INC** " locations of perl 5 libraries

---

SEE ALSO

a2p awk to perl translator s2p sed to perl translator

DIAGNOSTICS

The **-w** switch produces some lovely diagnostics.

See the *perldiag* manpage for explanations of all Perl's diagnostics.

Compilation errors will tell you the line number of the error, with an indication of the next token or token type that was to be examined. (In the case of a script passed to Perl via **-e** switches, each **-e** is counted as one line.)

Setuid scripts have additional constraints that can produce error messages such as "Insecure dependency". See the *perlsec* manpage .

Did we mention that you should definitely consider using the **-w** switch?

BUGS

The **-w** switch is not mandatory.

Perl is at the mercy of your machine's definitions of various operations such as type casting, *atof()* and *sprintf()* .

If your stdio requires a seek or eof between reads and writes on a particular stream, so does Perl. (This doesn't apply to *sysread()* and *syswrite()* .)

While none of the built-in data types have any arbitrary size limits (apart from memory size), there are still a few arbitrary limits: a given identifier may not be longer than 255 characters, and no component of your PATH may be longer than 255 if you use **-S** . A regular expression may not compile to more than 32767 bytes internally.

Perl actually stands for Pathologically Eclectic Rubbish Lister, but don't tell anyone I said that.

NOTES

TOC

The Perl motto is "There's more than one way to do it." Divining how many more is left as an exercise to the reader.

The three principle virtues of a programmer are Laziness, Impatience, and Hubris. See the Camel Book for why.

NAME

perldata - Perl data structures

DESCRIPTION

Variable names

Perl has three data structures: scalars, arrays of scalars, and associative arrays of scalars, known as "hashes". Normal arrays are indexed by number, starting with 0. (Negative subscripts count from the end.) Hash arrays are indexed by string.

Scalar values are always named with '$', even when referring to a scalar that is part of an array. It works like the English word "the". Thus we have:

$days # the simple scalar value "days" $days[28] # the 29th element of array @days $days{'Feb'} # the 'Feb' value from hash %days $#days # the last index of array @days

but entire arrays or array slices are denoted by '@', which works much like the word "these" or "those":

@days # ($days[0], $days[1],... $days[n]) @days[3,4,5] # same as @days[3..5] @days{'a','c'} # same as ($days{'a'},$days{'c'})

and entire hashes are denoted by '%':

%days # (key1, val1, key2, val2 ...)

In addition, subroutines are named with an initial '&', though this is optional when it's otherwise unambiguous (just as "do" is often redundant in English). Symbol table entries can be named with an initial '*', but you don't really care about that yet.

Every variable type has its own namespace. You can, without fear of conflict, use the same name for a scalar variable, an array, or a hash (or, for that matter, a filehandle, a subroutine name, or a label). This means that **$foo** and **@foo** are two different variables. It also means that **$foo [1]** is a part of **@foo** , not a part of **$foo** . This may seem a bit weird, but that's okay, because it is weird.

Since variable and array references always start with '$', '@', or '%', the "reserved" words aren't in fact reserved with respect to variable names. (They ARE reserved with respect to labels and filehandles, however, which don't have an initial special character. You can't have a filehandle named "log", for instance. Hint: you could say **open(LOG,'logfile')** rather than **open(log,'logfile')** . Using uppercase filehandles also improves readability and protects you from conflict with future reserved words.) Case *IS* significant--"FOO", "Foo" and "foo" are all different names. Names that start with a letter or underscore may also contain digits and underscores.

It is possible to replace such an alphanumeric name with an expression that returns a reference to an object of that type. For a description of this, see the *perlref* manpage .

Names that start with a digit may only contain more digits. Names which do not start with a letter, underscore, or digit are limited to one character, e.g. "$%" or "$$". (Most of these one character names have a predefined significance to Perl. For instance, $ **$** is the current process id.)

Context

The interpretation of operations and values in Perl sometimes depends on the requirements of the context around the operation or value. There are two major contexts: scalar and list. Certain operations return list values in contexts wanting a list, and scalar values otherwise. (If this is true of an operation it will be mentioned in the documentation for that operation.) In other words, Perl overloads certain operations based on whether the expected return value is singular or plural. (Some words in English work this way, like "fish" and "sheep".)

In a reciprocal fashion, an operation provides either a scalar or a list context to each of its arguments. For example, if you say

int(<STDIN>)

the integer operation provides a scalar context for the<STDIN> operator, which responds by reading one line from STDIN and passing it back to the integer operation, which will then find the integer value of that line and return that. If, on the other hand, you say

sort(<STDIN>)

then the sort operation provides a list context for<STDIN>, which will proceed to read every line available up to the end of file, and pass that list of lines back to the sort routine, which will then sort those lines and return them as a list to whatever the context of the sort was.

Assignment is a little bit special in that it uses its left argument to determine the context for the right argument. Assignment to a scalar evaluates the righthand side in a scalar context, while assignment to an array or array slice evaluates the righthand side in a list context. Assignment to a list also evaluates the righthand side in a list context.

User defined subroutines may choose to care whether they are being called in a scalar or list context, but most subroutines do not need to care, because scalars are automatically interpolated into lists. See *wantarray* .

Scalar values

Scalar variables may contain various kinds of singular data, such as numbers, strings and references. In general, conversion from one form to another is transparent. (A scalar may not contain multiple values, but may contain a reference to an array or hash containing multiple values.) Because of the automatic conversion of scalars, operations and functions that return scalars don't need to care (and, in fact, can't care) whether the context is looking for a string or a number.

A scalar value is interpreted as TRUE in the Boolean sense if it is not the null string or the number 0 (or its string equivalent, "0"). The Boolean context is just a special kind of scalar context.

There are actually two varieties of null scalars: defined and undefined. Undefined null scalars are returned when there is no real value for something, such as when there was an error, or at end of file, or when you refer to an uninitialized variable or element of an array. An undefined null scalar may become defined the first time you use it as if it were defined, but prior to that you can use the *defined()* operator to determine whether the value is defined or not.

The length of an array is a scalar value. You may find the length of array **@days** by evaluating **$# days** , as in **csh** . (Actually, it's not the length of the array, it's the subscript of the last element, since there is (ordinarily) a 0th element.) Assigning to **$# days** changes the length of the array. Shortening an array by this method destroys intervening values. Lengthening an array that was previously shortened *NO LONGER* recovers the values that were in those elements. (It used to in Perl 4, but we had to break this make to make sure destructors were called when expected.) You can also gain some measure of efficiency by preextending an array that is going to get big. (You can also extend an array by assigning to an element that is off the end of the array.) You can truncate an array down to nothing by assigning the null list () to it. The following are equivalent:

@whatever = (); $#whatever = $[- 1;

If you evaluate a named array in a scalar context, it returns the length of the array. (Note that this is not true of lists, which return the last value, like the C comma operator.) The following is always true:

scalar(@whatever) == $#whatever - $[+ 1;

Version 5 of Perl changed the semantics of $[: files that don't set the value of $[no longer need to worry about whether another file changed its value. (In other words, use of $[is deprecated.) So in general you can just assume that

scalar(@whatever) == $#whatever + 1;

If you evaluate a hash in a scalar context, it returns a value which is true if and only if the hash contains any key/value pairs. (If there are any key/value pairs, the value returned is a string consisting of the number of used buckets and the number of allocated buckets, separated by a slash. This is pretty much only useful to find out whether Perl's (compiled in) hashing algorithm is performing poorly on your data set. For example, you stick 10,000 things in a hash, but evaluating **%HASH** in scalar context reveals "1/16", which means only one out of sixteen buckets has been touched, and presumably contains all 10,000 of your items. This isn't supposed to happen.)

Scalar value constructors

Numeric literals are specified in any of the customary floating point or integer formats:

12345 12345.67 .23E-10 0xffff # hex 0377 # octal 4_294_967_296 # underline for legibility

String literals are delimited by either single or double quotes. They work much like shell quotes: double-quoted string literals are subject to backslash and variable substitution; single-quoted strings are

not (except for " \' " and " \\ "). The usual Unix backslash rules apply for making characters such as newline, tab, etc., as well as some more exotic forms. See *qq* for a list.

You can also embed newlines directly in your strings, i.e. they can end on a different line than they begin. This is nice, but if you forget your trailing quote, the error will not be reported until Perl finds another line containing the quote character, which may be much further on in the script. Variable substitution inside strings is limited to scalar variables, arrays, and array slices. (In other words, identifiers beginning with **$** or @, followed by an optional bracketed expression as a subscript.) The following code segment prints out "The price is **$100** ."

$Price = '$100'; # not interpreted print "The price is $Price.\n"; # interpreted

As in some shells, you can put curly brackets around the identifier to delimit it from following alphanumerics. In fact, an identifier within such curlies is forced to be a string, as is any single identifier within a hash subscript. Our earlier example,

$days{'Feb'}

can be written as

$days{Feb}

and the quotes will be assumed automatically. But anything more complicated in the subscript will be interpreted as an expression.

Note that a single-quoted string must be separated from a preceding word by a space, since single quote is a valid (though deprecated) character in an identifier (see *Packages*).

Two special literals are __LINE__ and __FILE__, which represent the current line number and filename at that point in your program. They may only be used as separate tokens; they will not be interpolated into strings. In addition, the token __END__ may be used to indicate the logical end of the script before the actual end of file. Any following text is ignored, but may be read via the DATA filehandle. (The DATA filehandle may read data only from the main script, but not from any required file or evaluated string.) The two control characters ^D and ^Z are synonyms for __END__.

A word that has no other interpretation in the grammar will be treated as if it were a quoted string. These are known as "barewords". As with filehandles and labels, a bareword that consists entirely of lowercase letters risks conflict with future reserved words, and if you use the **-w** switch, Perl will warn you about any such words. Some people may wish to outlaw barewords entirely. If you say

use strict 'subs';

then any bareword that would NOT be interpreted as a subroutine call produces a compile-time error instead. The restriction lasts to the end of the enclosing block. An inner block may countermand this by saying **no strict 'subs'** .

Array variables are interpolated into double-quoted strings by joining all the elements of the array with the delimiter specified in the **$"** variable, space by default. The following are equivalent:

$temp = join($",@ARGV); system "echo $temp"; system "echo @ARGV";

Within search patterns (which also undergo double-quotish substitution) there is a bad ambiguity: Is /**$foo [bar]**/ to be interpreted as /**${ foo}[bar]**/ (where **[bar]** is a character class for the regular expression) or as /**${ foo[bar]}**/ (where **[bar]** is the subscript to array **@foo**)? If **@foo** doesn't otherwise exist, then it's obviously a character class. If **@foo** exists, Perl takes a good guess about **[bar]** , and is almost always right. If it does guess wrong, or if you're just plain paranoid, you can force the correct interpretation with curly brackets as above.

A line-oriented form of quoting is based on the shell "here-doc" syntax. Following a << you specify a string to terminate the quoted material, and all lines following the current line down to the terminating string are the value of the item. The terminating string may be either an identifier (a word), or some quoted text. If quoted, the type of quotes you use determines the treatment of the text, just as in regular quoting. An unquoted identifier works like double quotes. There must be no space between the << and the identifier. (If you put a space it will be treated as a null identifier, which is valid, and matches the first blank line--see the Merry Christmas example below.) The terminating string must appear by itself (unquoted and with no surrounding whitespace) on the terminating line.

```
    print <<EOF;      # same as above
The price is  $Price .
EOF
    print <<"EOF";    # same as above
The price is  $Price .
EOF
    print << x 10;    # Legal but discouraged.  Use <<"".
Merry Christmas!
    print <<`EOC`;    # execute commands
echo hi there
echo lo there
EOC
    print <<"foo", <<"bar";     # you can stack them
I said foo.
foo
I said bar.
bar
    myfunc(<<"THIS", 23, <<'THAT');
Here's a line
or two.
THIS
and here another.
THAT
```

Just don't forget that you have to put a semicolon on the end to finish the statement, as Perl doesn't know you're not going to try to do this:

```
    print <<ABC
179231
```

```
    ABC
        + 20;
```

List value constructors

List values are denoted by separating individual values by commas (and enclosing the list in parentheses where precedence requires it):

(LIST)

In a context not requiring a list value, the value of the list literal is the value of the final element, as with the C comma operator. For example,

@foo = ('cc', '-E', $bar);

assigns the entire list value to array foo, but

$foo = ('cc', '-E', $bar);

assigns the value of variable bar to variable foo. Note that the value of an actual array in a scalar context is the length of the array; the following assigns to **$foo** the value 3:

@foo = ('cc', '-E', $bar); $foo = @foo; # $foo gets 3

You may have an optional comma before the closing parenthesis of an list literal, so that you can say:

@foo = (1, 2, 3,);

LISTs do automatic interpolation of sublists. That is, when a LIST is evaluated, each element of the list is evaluated in a list context, and the resulting list value is interpolated into LIST just as if each individual element were a member of LIST. Thus arrays lose their identity in a LIST--the list

(@foo,@bar,&SomeSub)

contains all the elements of **@foo** followed by all the elements of **@bar** , followed by all the elements returned by the subroutine named SomeSub. To make a list reference that does *NOT* interpolate, see the *perlref* manpage .

The null list is represented by (). Interpolating it in a list has no effect. Thus ((),(),()) is equivalent to (). Similarly, interpolating an array with no elements is the same as if no array had been interpolated at that point.

A list value may also be subscripted like a normal array. You must put the list in parentheses to avoid ambiguity. Examples:

Stat returns list value. $time = (stat($file))[8]; # Find a hex digit. $hexdigit = ('a','b','c','d','e','f')[$digit-10]; # A "reverse comma operator". return (pop(@foo),pop(@foo))[0];

Lists may be assigned to if and only if each element of the list is legal to assign to:

TOC

($a, $b, $c) = (1, 2, 3); ($map{'red'}, $map{'blue'}, $map{'green'}) = (0x00f, 0x0f0, 0xf00);

The final element may be an array or a hash:

($a, $b, @rest) = split; local($a, $b, %rest) = @_;

You can actually put an array anywhere in the list, but the first array in the list will soak up all the values, and anything after it will get a null value. This may be useful in a *local()* or *my()* .

A hash literal contains pairs of values to be interpreted as a key and a value:

same as map assignment above %map = ('red',0x00f,'blue',0x0f0,'green',0xf00);

It is often more readable to use the => operator between key/value pairs (the => operator is actually nothing more than a more visually distinctive synonym for a comma):

%map = ('red' => 0x00f, 'blue' => 0x0f0, 'green' => 0xf00,);

Array assignment in a scalar context returns the number of elements produced by the expression on the right side of the assignment:

$x = (($foo,$bar) = (3,2,1)); # set $x to 3, not 2

This is very handy when you want to do a list assignment in a Boolean context, since most list functions return a null list when finished, which when assigned produces a 0, which is interpreted as FALSE.

TOC

NAME

perlsyn - Perl syntax

DESCRIPTION

A Perl script consists of a sequence of declarations and statements. The only things that need to be declared in Perl are report formats and subroutines. See the sections below for more information on those declarations. All uninitialized user-created objects are assumed to start with a null or 0 value until they are defined by some explicit operation such as assignment. (Though you can get warnings about the use of undefined values if you like.) The sequence of statements is executed just once, unlike in **sed** and **awk** scripts, where the sequence of statements is executed for each input line. While this means that you must explicitly loop over the lines of your input file (or files), it also means you have much more control over which files and which lines you look at. (Actually, I'm lying--it is possible to do an implicit loop with either the **-n** or **-p** switch. It's just not the mandatory default like it is in **sed** and **awk** .)

Perl is, for the most part, a free-form language. (The only exception to this is format declarations, for obvious reasons.) Comments are indicated by the "#" character, and extend to the end of the line. If you attempt to use **/* */** C-style comments, it will be interpreted either as division or pattern matching, depending on the context, and C++ // comments just look like a null regular expression, So don't do that.

A declaration can be put anywhere a statement can, but has no effect on the execution of the primary sequence of statements--declarations all take effect at compile time. Typically all the declarations are put at the beginning or the end of the script.

As of Perl 5, declaring a subroutine allows a subroutine name to be used as if it were a list operator from that point forward in the program. You can declare a subroutine without defining it by saying just

sub myname; $me = myname $0 or die "can't get myname";

Note that it functions as a list operator though, not a unary operator, so be careful to use **or** instead of **||** there.

Subroutines declarations can also be imported by a **use** statement.

Also as of Perl 5, a statement sequence may contain declarations of lexically scoped variables, but apart from declaring a variable name, the declaration acts like an ordinary statement, and is elaborated within the sequence of statements as if it were an ordinary statement.

Simple statements

The only kind of simple statement is an expression evaluated for its side effects. Every simple statement must be terminated with a semicolon, unless it is the final statement in a block, in which case the semicolon is optional. (A semicolon is still encouraged there if the block takes up more than one line, since you may eventually add another line.) Note that there are some operators like **eval {}** and **do {}** that look like compound statements, but aren't (they're just TERMs in an expression), and thus need an explicit termination if used as the last item in a statement.

Any simple statement may optionally be followed by a *SINGLE* modifier, just before the terminating semicolon (or block ending). The possible modifiers are:

if EXPR unless EXPR while EXPR until EXPR

The **if** and **unless** modifiers have the expected semantics, presuming you're a speaker of English. The **while** and **until** modifiers also have the usual "while loop" semantics (conditional evaluated first), except when applied to a do-BLOCK (or to the now-deprecated do-SUBROUTINE statement), in which case the block executes once before the conditional is evaluated. This is so that you can write loops like:

do { $_ = <STDIN>; ... } until $_ eq ".\n";

See *do* . Note also that the loop control statements described later will *NOT* work in this construct, since modifiers don't take loop labels. Sorry. You can always wrap another block around it to do that sort of thing.)

Compound statements

In Perl, a sequence of statements that defines a scope is called a block. Sometimes a block is delimited by the file containing it (in the case of a required file, or the program as a whole), and sometimes a block is delimited by the extent of a string (in the case of an eval).

But generally, a block is delimited by curly brackets, also known as braces. We will call this syntactic construct a BLOCK.

The following compound statements may be used to control flow:

if (EXPR) BLOCK if (EXPR) BLOCK else BLOCK if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK LABEL while (EXPR) BLOCK LABEL while (EXPR) BLOCK continue BLOCK LABEL for (EXPR; EXPR; EXPR) BLOCK LABEL foreach VAR (LIST) BLOCK LABEL BLOCK continue BLOCK

Note that, unlike C and Pascal, these are defined in terms of BLOCKs, not statements. This means that the curly brackets are *required* --no dangling statements allowed. If you want to write conditionals without curly brackets there are several other ways to do it. The following all do the same thing:

if (!open(FOO)) { die "Can't open $FOO: $!"; } die "Can't open $FOO: $!" unless open(FOO); open(FOO) or die "Can't open $FOO: $!"; # FOO or bust! open(FOO) ? 'hi mom' : die "Can't open $FOO: $!"; # a bit exotic, that last one

The **if** statement is straightforward. Since BLOCKs are always bounded by curly brackets, there is never any ambiguity about which **if** an **else** goes with. If you use **unless** in place of **if** , the sense of the test is reversed.

The **while** statement executes the block as long as the expression is true (does not evaluate to the null string or 0 or "0"). The LABEL is optional, and if present, consists of an identifier followed by a colon. The LABEL identifies the loop for the loop control statements **next** , **last** , and **redo** (see below). If there is a **continue** BLOCK, it is always executed just before the conditional is about to be evaluated again, just like the third part of a **for** loop in C. Thus it can be used to increment a loop variable, even when the loop has been continued via the **next** statement (which is similar to the C **continue** statement).

If the word **while** is replaced by the word **until** , the sense of the test is reversed, but the conditional is still tested before the first iteration.

In either the **if** or the **while** statement, you may replace "(EXPR)" with a BLOCK, and the conditional is true if the value of the last statement in that block is true. (This feature continues to work in Perl 5 but is deprecated. Please change any occurrences of "if BLOCK" to "if (do BLOCK)".)

The C-style **for** loop works exactly like the corresponding **while** loop:

for ($i = 1; $i < 10; $i++) { ... }

is the same as

$i = 1; while ($i < 10) { ... } continue { $i++; }

The foreach loop iterates over a normal list value and sets the variable VAR to be each element of the list in turn. The variable is implicitly local to the loop and regains its former value upon exiting the loop. (If the variable was previously declared with **my** , it uses that variable instead of the global one, but it's still localized to the loop.) The **foreach** keyword is actually a synonym for the **for** keyword, so you can use **foreach** for readability or **for** for brevity. If VAR is omitted, **$_** is set to each value. If LIST is an actual array (as opposed to an expression returning a list value), you can modify each element of the array by modifying VAR inside the loop. Examples:

for (@ary) { s/foo/bar/; } foreach $elem (@elements) { $elem *= 2; } for ((10,9,8,7,6,5,4,3,2,1,'BOOM')) { print $_, "\n"; sleep(1); } for (1..15) { print "Merry Christmas\n"; } foreach $item (split(/:[\\\n:]*/, $ENV{'TERMCAP'})) { print "Item: $item\n"; }

A BLOCK by itself (labeled or not) is semantically equivalent to a loop that executes once. Thus you can use any of the loop control statements in it to leave or restart the block. The **continue** block is optional. This construct is particularly nice for doing case structures.

SWITCH: { if (/^abc/) { $abc = 1; last SWITCH; } if (/^def/) { $def = 1; last SWITCH; } if (/^xyz/) { $xyz = 1; last SWITCH; } $nothing = 1; }

There is no official switch statement in Perl, because there are already several ways to write the equivalent. In addition to the above, you could write

SWITCH: { $abc = 1, last SWITCH if /^abc/; $def = 1, last SWITCH if /^def/; $xyz = 1, last SWITCH if /^xyz/; $nothing = 1; }

(That's actually not as strange as it looks one you realize that you can use loop control "operators" wi an expression, That's just the normal C comma operator.)

or

SWITCH: { /^abc/ && do { $abc = 1; last SWITCH; }; /^def/ && do { $def = 1; last SWITCH; }; /^xyz/ && do { $xyz = 1; last SWITCH; }; $nothing = 1; }

or formatted so it stands out more as a "proper" switch statement:

SWITCH: { /^abc/ && do { $abc = 1; last SWITCH; }; /^def/ && do { $def = 1; last SWITCH; }; /^xyz/ && do { $xyz = 1; last SWITCH; }; $nothing = 1; }

or

SWITCH: { /^abc/ and $abc = 1, last SWITCH; /^def/ and $def = 1, last SWITCH; /^xyz/ and $xyz = 1, last SWITCH; $nothing = 1; }

or even, horrors,

if (/^abc/) { $abc = 1 } elsif (/^def/) { $def = 1 } elsif (/^xyz/) { $xyz = 1 } else { $nothing = 1 }

NAME

perlop - Perl operators and precedence

SYNOPSIS

Perl operators have the following associativity and precedence, listed from highest precedence to lowest. Note that all operators borrowed from C keep the same precedence relationship with each other, even where C's precedence is slightly screwy. (This makes learning Perl easier for C folks.)

- left terms and list operators (leftward)
- left ->
- nonassoc ++ --
- right **
- right ! ~ \ and unary + and -
- left =~ !~
- left * / % x
- left + - .
- left << >>
- nonassoc named unary operators
- nonassoc < > <= >= lt gt le ge
- nonassoc == != <=> eq ne cmp
- left &
- left | ^
- left &&
- left ||
- nonassoc ..
- right ?:
- right = += -= *= etc.
- left , =>
- nonassoc list operators (rightward)
- left not
- left and
- left or xor

In the following sections, these operators are covered in precedence order.

DESCRIPTIONS

Terms and List Operators (Leftward)

Any TERM is of highest precedence of Perl. These includes variables, quote and quotelike operators, any expression in parentheses, and any function whose arguments are parenthesized. Actually, there aren't really functions in this sense, just list operators and unary operators behaving as functions because you put parentheses around the arguments. These are all documented in the *perlfunc* manpage .

If any list operator (*print()* , etc.) or any unary operator (*chdir()* , etc.) is followed by a left parenthesis as the next token, the operator and arguments within parentheses are taken to be of highest precedence, just like a normal function call.

In the absence of parentheses, the precedence of list operators such as **print** , **sort** , or **chmod** is either very high or very low depending on whether you look at the left side of operator or the right side of it. For example, in

@ary = (1, 3, sort 4, 2); print @ary; # prints 1324

the commas on the right of the sort are evaluated before the sort, but the commas on the left are evaluated after. In other words, list operators tend to gobble up all the arguments that follow them, and then act like a simple TERM with regard to the preceding expression. Note that you have to be careful with parens:

These evaluate exit before doing the print: print($foo, exit); # Obviously not what you want. print $foo, exit; # Nor is this. # These do the print before evaluating exit: (print $foo), exit; # This is what you want. print($foo), exit; # Or this. print ($foo), exit; # Or even this.

Also note that

print ($foo & 255) + 1, "\n";

probably doesn't do what you expect at first glance. See *Named Unary Operators* for more discussion of this.

Also parsed as terms are the **do {}** and **eval {}** constructs, as well as subroutine and method calls, and the anonymous constructors **[]** and **{}** .

See also *Quote and Quotelike Operators* toward the end of this section, as well as *I/O Operators* .

The Arrow Operator

Just as in C and C++, " **->** " is an infix dereference operator. If the right side is either a **[...]** or **{...}** subscript, then the left side must be either a hard or symbolic reference to an array or hash (or a location capable of holding a hard reference, if it's an lvalue (assignable)). See the *perlref* manpage .

Otherwise, the right side is a method name or a simple scalar variable containing the method name, a the left side must either be an object (a blessed reference) or a class name (that is, a package name). the *perlobj* manpage .

Autoincrement and Autodecrement

"++" and "--" work as in C. That is, if placed before a variable, they increment or decrement the variable before returning the value, and if placed after, increment or decrement the variable after returning the value.

The autoincrement operator has a little extra built-in magic to it. If you increment a variable that is numeric, or that has ever been used in a numeric context, you get a normal increment. If, however, the variable has only been used in string contexts since it was set, and has a value that is not null and matches the pattern /^**[a-zA-Z]*[0-9]*$/** , the increment is done as a string, preserving each character within its range, with carry:

print ++($foo = '99'); # prints '100' print ++($foo = 'a0'); # prints 'a1' print ++($foo = 'Az'); # prints 'Ba' print ++($foo = 'zz'); # prints 'aaa'

The autodecrement operator is not magical.

Exponentiation

Binary "**" is the exponentiation operator. Note that it binds even more tightly than unary minus, so -2**4 is -(2**4), not (-2)**4.

Symbolic Unary Operators

Unary "!" performs logical negation, i.e. "not". See also **not** for a lower precedence version of this.

Unary "-" performs arithmetic negation if the operand is numeric. If the operand is an identifier, a string consisting of a minus sign concatenated with the identifier is returned. Otherwise, if the string starts with a plus or minus, a string starting with the opposite sign is returned. One effect of these rules is that **-bareword** is equivalent to **"-bareword"** .

Unary "~" performs bitwise negation, i.e. 1's complement.

Unary "+" has no effect whatsoever, even on strings. It is useful syntactically for separating a function name from a parenthesized expression that would otherwise be interpreted as the complete list of function arguments. (See examples above under *List Operators* .)

Unary "\" creates a reference to whatever follows it. See the *perlref* manpage . Do not confuse this behavior with the behavior of backslash within a string, although both forms do convey the notion of protecting the next thing from interpretation.

Binding Operators

Binary "=~" binds an expression to a pattern match. Certain operations search or modify the string **$_** by default. This operator makes that kind of operation work on some other string. The right argument is a search pattern, substitution, or translation. The left argument is what is supposed to be searched, substituted, or translated instead of the default **$_** . The return value indicates the success of the operation. (If the right argument is an expression rather than a search pattern, substitution, or translation, it is interpreted as a search pattern at run time. This is less efficient than an explicit search, since the pattern must be compiled every time the expression is evaluated--unless you've used **/o** .)

Binary "!~" is just like "=~" except the return value is negated in the logical sense.

Multiplicative Operators

Binary "*" multiplies two numbers.

Binary "/" divides two numbers.

Binary "%" computes the modulus of the two numbers.

Binary "x" is the repetition operator. In a scalar context, it returns a string consisting of the left operand repeated the number of times specified by the right operand. In a list context, if the left operand is a list in parens, it repeats the list.

print '-' x 80; # print row of dashes print "\t" x ($tab/8), ' ' x ($tab%8); # tab over @ones = (1) x 80; # a list of 80 1's @ones = (5) x @ones; # set all elements to 5

Additive Operators

Binary "+" returns the sum of two numbers.

Binary "-" returns the difference of two numbers.

Binary "." concatenates two strings.

Shift Operators

Binary "<<" returns the value of its left argument shifted left by the number of bits specified by the right argument. Arguments should be integers.

Binary >>" returns the value of its left argument shifted right by the number of bits specified by the right argument. Arguments should be integers.

TOC

Named Unary Operators

The various named unary operators are treated as functions with one argument, with optional parentheses. These include the filetest operators, like **-f** , **-M** , etc. See the *perlfunc* manpage .

If any list operator (*print()* , etc.) or any unary operator (*chdir()* , etc.) is followed by a left parenthesis as the next token, the operator and arguments within parentheses are taken to be of highest precedence, just like a normal function call. Examples:

chdir $foo || die; # (chdir $foo) || die chdir($foo) || die; # (chdir $foo) || die chdir ($foo) || die; # (chdir $foo) || die chdir +($foo) || die; # (chdir $foo) || die

but, because * is higher precedence than ||:

chdir $foo * 20; # chdir ($foo * 20) chdir($foo) * 20; # (chdir $foo) * 20 chdir ($foo) * 20; # (chdir $foo) * 20 chdir +($foo) * 20; # chdir ($foo * 20) rand 10 * 20; # rand (10 * 20) rand(10) * 20; # (rand 10) * 20 rand (10) * 20; # (rand 10) * 20 rand +(10) * 20; # rand (10 * 20)

See also *List Operators* .

Relational Operators

Binary <" returns true if the left argument is numerically less than the right argument.

Binary ">" returns true if the left argument is numerically greater than the right argument.

Binary <=" returns true if the left argument is numerically less than or equal to the right argument.

Binary ">=" returns true if the left argument is numerically greater than or equal to the right argument.

Binary "lt" returns true if the left argument is stringwise less than the right argument.

Binary "gt" returns true if the left argument is stringwise greater than the right argument.

Binary "le" returns true if the left argument is stringwise less than or equal to the right argument.

Binary "ge" returns true if the left argument is stringwise greater than or equal to the right argument.

Equality Operators

Binary "==" returns true if the left argument is numerically equal to the right argument.

Binary "!=" returns true if the left argument is numerically not equal to the right argument.

Binary <=>" returns -1, 0, or 1 depending on whether the left argument is numerically less than, equal to, or greater than the right argument.

Binary "eq" returns true if the left argument is stringwise equal to the right argument.

Binary "ne" returns true if the left argument is stringwise not equal to the right argument.

Binary "cmp" returns -1, 0, or 1 depending on whether the left argument is stringwise less than, equa or greater than the right argument.

Bitwise And

Binary "&" returns its operators ANDed together bit by bit.

Bitwise Or and Exclusive Or

Binary "|" returns its operators ORed together bit by bit.

Binary "^" returns its operators XORed together bit by bit.

C-style Logical And

Binary "&&" performs a short-circuit logical AND operation. That is, if the left operand is false, the right operand is not even evaluated. Scalar or list context propagates down to the right operand if it is evaluated.

C-style Logical Or

Binary "||" performs a short-circuit logical OR operation. That is, if the left operand is true, the right operand is not even evaluated. Scalar or list context propagates down to the right operand if it is evaluated.

The || and **&&** operators differ from C's in that, rather than returning 0 or 1, they return the last value evaluated. Thus, a reasonably portable way to find out the home directory (assuming it's not "0") might be:

$home = $ENV{'HOME'} || $ENV{'LOGDIR'} || (getpwuid($<))[7] || die "You're homeless!\n";

As more readable alternatives to **&&** and || , Perl provides "and" and "or" operators (see below). The short-circuit behavior is identical. The precedence of "and" and "or" is much lower, however, so that you can safely use them after a list operator without the need for parentheses:

unlink "alpha", "beta", "gamma" or gripe(), next LINE;

With the C-style operators that would have been written like this:

unlink("alpha", "beta", "gamma") || (gripe(), next LINE);

Range Operator

Binary ".." is the range operator, which is really two different operators depending on the context. In a list context, it returns an array of values counting (by ones) from the left value to the right value. This is useful for writing **for (1..10)** loops and for doing slice operations on arrays. Be aware that under the current implementation, a temporary array is created, so you'll burn a lot of memory if you write something like this:

for (1 .. 1_000_000) { # code }

In a scalar context, ".." returns a boolean value. The operator is bistable, like a flip-flop, and emulates the line-range (comma) operator of **sed** , **awk** , and various editors. Each ".." operator maintains its own boolean state. It is false as long as its left operand is false. Once the left operand is true, the range operator stays true until the right operand is true, *AFTER* which the range operator becomes false again. (It doesn't become false till the next time the range operator is evaluated. It can test the right operand and become false on the same evaluation it became true (as in **awk**), but it still returns true once. If you don't want it to test the right operand till the next evaluation (as in **sed**), use three dots ("...") instead of two.) The right operand is not evaluated while the operator is in the "false" state, and the left operand is not evaluated while the operator is in the "true" state. The precedence is a little lower than || and &&. The value returned is either the null string for false, or a sequence number (beginning with 1) for true. The sequence number is reset for each range encountered. The final sequence number in a range has the string "E0" appended to it, which doesn't affect its numeric value, but gives you something to search for if you want to exclude the endpoint. You can exclude the beginning point by waiting for the sequence number to be greater than 1. If either operand of scalar ".." is a numeric literal, that operand is implicitly compared to the **$.** variable, the current line number. Examples:

As a scalar operator:

if (101 .. 200) { print; } # print 2nd hundred lines next line if (1 .. /^$/); # skip header lines s/^/> / if (/^$/ .. eof()); # quote body

As a list operator:

for (101 .. 200) { print; } # print $_ 100 times @foo = @foo[$[.. $#foo]; # an expensive no-op @foo = @foo[$#foo-4 .. $#foo]; # slice last 5 items

The range operator (in a list context) makes use of the magical autoincrement algorithm if the operaands are strings. You can say

@alphabet = ('A' .. 'Z');

to get all the letters of the alphabet, or

$hexdigit = (0 .. 9, 'a' .. 'f')[$num & 15];

to get a hexadecimal digit, or

@z2 = ('01' .. '31'); print $z2[$mday];

to get dates with leading zeros. If the final value specified is not in the sequence that the magical

increment would produce, the sequence goes until the next value would be longer than the final value specified.

Conditional Operator

Ternary "?:" is the conditional operator, just as in C. It works much like an if-then-else. If the argument before the ? is true, the argument before the : is returned, otherwise the argument after the : is returned. Scalar or list context propagates downward into the 2nd or 3rd argument, whichever is selected. The operator may be assigned to if both the 2nd and 3rd arguments are legal lvalues (meaning that you can assign to them):

($a_or_b ? $a : $b) = $c;

Note that this is not guaranteed to contribute to the readability of your program.

Assigment Operators

"=" is the ordinary assignment operator.

Assignment operators work as in C. That is,

$a += 2;

is equivalent to

$a = $a + 2;

although without duplicating any side effects that dereferencing the lvalue might trigger, such as from *tie()* . Other assignment operators work similarly. The following are recognized:

```
**=     +=      *=      &=      <<=     &&=
        -=      /=      |=     >>=      ||=
        .=      %=      ^=
                x=
```

Note that while these are grouped by family, they all have the precedence of assignment.

Unlike in C, the assignment operator produces a valid lvalue. Modifying an assignment is equivalent to doing the assignment and then modifying the variable that was assigned to. This is useful for modifying a copy of something, like this:

($tmp = $global) =~ tr [A-Z] [a-z];

Likewise,

($a += 2) *= 3;

is equivalent to

$a += 2; $a *= 3;

Comma Operator

Binary "," is the comma operator. In a scalar context it evaluates its left argument, throws that value away, then evaluates its right argument and returns that value. This is just like C's comma operator.

In a list context, it's just the list argument separator, and inserts both its arguments into the list.

The => digraph is simply a synonym for the comma operator. It's useful for documenting arguments that come in pairs.

List Operators (Rightward)

On the right side of a list operator, it has very low precedence, such that it controls all comma-separated expressions found there. The only operators with lower precedence are the logical operators "and", "or", and "not", which may be used to evaluate calls to list operators without the need for extra parentheses:

open HANDLE, "filename" or die "Can't open: $!\n";

See also discussion of list operators in *List Operators (Leftward)* .

Logical Not

Unary "not" returns the logical negation of the expression to its right. It's the equivalent of "!" except for the very low precedence.

Logical And

Binary "and" returns the logical conjunction of the two surrounding expressions. It's equivalent to && except for the very low precedence. This means that it short-circuits: i.e. the right expression is evaluated only if the left expression is true.

Logical or and Exclusive Or

Binary "or" returns the logical disjunction of the two surrounding expressions. It's equivalent to || except for the very low precedence. This means that it short-circuits: i.e. the right expression is evaluated only if the left expression is false.

Binary "xor" returns the exclusive-OR of the two surrounding expressions. It cannot short circuit, of course.

C Operators Missing From Perl

Here is what C has that Perl doesn't:

unary &

Address-of operator. (But see the "\" operator for taking a reference.)

unary *

Dereference-address operator. (Perl's prefix dereferencing operators are typed: $, @, %, and &.)

(TYPE)

Type casting operator.

.

Quote and Quotelike Operators

While we usually think of quotes as literal values, in Perl they function as operators, providing various kinds of interpolating and pattern matching capabilities. Perl provides customary quote characters for these behaviors, but also provides a way for you to choose your quote character for any of them. In the following table, a **{}** represents any pair of delimiters you choose. Non-bracketing delimiters use the same character fore and aft, but the 4 sorts of brackets (round, angle, square, curly) will all nest.

Customary Generic Meaning Interpolates " q{} Literal no "" qq{} Literal yes `` qx{} Command yes qw{} Word list no // m{} Pattern match yes s{}{} Substitution yes tr{}{} Translation no

For constructs that do interpolation, variables beginning with " **$** or " **@** " are interpolated, as are the following sequences:

\t tab \n newline \r return \f form feed \v vertical tab, whatever that is \b backspace \a alarm (bell) \e escape \033 octal char \x1b hex char \c[control char \l lowercase next char \u uppercase next char \L lowercase till \E \U uppercase till \E \E end case modification \Q quote regexp metacharacters till \E

Patterns are subject to an additional level of interpretation as a regular expression. This is done as a second pass, after variables are interpolated, so that regular expressions may be incorporated into the pattern from the variables. If this is not what you want, use **\Q** to interpolate a variable literally.

Apart from the above, there are no multiple levels of interpolation. In particular, contrary to the expectations of shell programmers, backquotes do *NOT* interpolate within double quotes, nor do single quotes impede evaluation of variables when used within double quotes.

?PATTERN?

This is just like the **/pattern/** search, except that it matches only once between calls to the *reset()* operator. This is a useful optimization when you only want to see the first occurrence of something in each file of a set of files, for instance. Only **??** patterns local to the current package are reset.

This usage is vaguely deprecated, and may be removed in some future version of Perl.

m/PATTERN/gimosx

/PATTERN/gimosx

Searches a string for a pattern match, and in a scalar context returns true (1) or false ('). If no string is specified via the =~ or !~ operator, the **$_** string is searched. (The string specified with need not be an lvalue--it may be the result of an expression evaluation, but remember the =~ binds rather tightly.) See also the *perlre* manpage .

Options are:

- g Match globally, i.e. find all occurrences.
- i Do case-insensitive pattern matching.
- m Treat string as multiple lines.
- o Only compile pattern once.
- s Treat string as single line.
- x Use extended regular expressions.

If "/" is the delimiter then the initial **m** is optional. With the **m** you can use any pair of non-alphanumeric, non-whitespace characters as delimiters. This is particularly useful for matching Unix path names that contain "/", to avoid LTS (leaning toothpick syndrome).

PATTERN may contain variables, which will be interpolated (and the pattern recompiled) every time the pattern search is evaluated. (Note that **$)** and **$|** might not be interpolated because they look like end-of-string tests.) If you want such a pattern to be compiled only once, add a **/o** after the trailing delimiter. This avoids expensive run-time recompilations, and is useful when the value you are interpolating won't change over the life of the script. However, mentioning **/o** constitutes a promise that you won't change the variables in the pattern. If you change them, Perl won't even notice.

If the PATTERN evaluates to a null string, the most recently executed (and successfully compiled) regular expression is used instead.

If used in a context that requires a list value, a pattern match returns a list consisting of the subexpressions matched by the parentheses in the pattern, i.e. (**$1** , **$2** , **$3** ...). (Note that here **$1** etc. are also set, and that this differs from Perl 4's behavior.) If the match fails, a null array is returned. If the match succeeds, but there were no parentheses, a list value of (1) is returned.

Examples:

open(TTY, '/dev/tty'); <TTY> =~ /^y/i && foo(); # do foo if desired if (/Version: *([0-9.]*)/) { $version = $1; } next if m#^/usr/spool/uucp#; # poor man's grep $arg = shift; while (<>) { print if /$arg/o; # compile only once } if (($F1, $F2, $Etc) = ($foo =~ /^(\S+)\s+(\S+)\s*(.*)/))

This last example splits **$foo** into the first two words and the remainder of the line, and assigns those three fields to **$F1** , **$F2** and **$Etc** . The conditional is true if any variables were assigned, i.e. if the pattern matched.

The **/g** modifier specifies global pattern matching--that is, matching as many times as possible within the string. How it behaves depends on the context. In a list context, it returns a list of all the substrings matched by all the parentheses in the regular expression. If there are no parentheses, it returns a list of all the matched strings, as if there were parentheses around the whole pattern.

In a scalar context, **m//g** iterates through the string, returning TRUE each time it matches, and FALSE when it eventually runs out of matches. (In other words, it remembers where it left off last time and restarts the search at that point. You can actually find the current match position of a string using the *pos()* function--see the *perlfunc* manpage .) If you modify the string in any way, the match position is reset to the beginning. Examples:

list context ($one,$five,$fifteen) = (`uptime` =~ /(\d+\.\d+)/g); # scalar context $/ = ""; $* = 1; # $* deprecated in Perl 5 while ($paragraph = <>) { while ($paragraph =~ /[a-z]['")]*[.!?]+['")]*\s/g) { $sentences++; } } print "$sentences\n";

q/STRING/

'STRING'

A single-quoted, literal string. Backslashes are ignored, unless followed by the delimiter or another backslash, in which case the delimiter or backslash is interpolated.

$foo = q!I said, "You said, 'She said it.'"!; $bar = q('This is it.');

qq/STRING/

"STRING"

A double-quoted, interpolated string.

$_ .= qq (*** The previous line contains the naughty word "$1".\n) if /(tcl|rexx|python)/; # :-)

qx/STRING/

`STRING`

A string which is interpolated and then executed as a system command. The collected standard output of the command is returned. In scalar context, it comes back as a single (potentially multi-line) string. In list context, returns a list of lines (however you've defined lines with $/ or **$INPUT_RECORD_SEPARATOR**).

$today = qx{ date };

See *I/O Operators* for more discussion.

qw/STRING/

Returns a list of the words extracted out of STRING, using embedded whitespace as the word delimiters. It is exactly equivalent to

split(' ', q/STRING/);

Some frequently seen examples:

use POSIX qw(setlocale localeconv) @EXPORT = qw(foo bar baz);

s/PATTERN/REPLACEMENT/egimosx

Searches a string for a pattern, and if found, replaces that pattern with the replacement text and returns the number of substitutions made. Otherwise it returns false (0).

If no string is specified via the **=~** or **!~** operator, the **$_** variable is searched and modified. (The string specified with **=~** must be a scalar variable, an array element, a hash element, or an assignment to one of those, i.e. an lvalue.)

If the delimiter chosen is single quote, no variable interpolation is done on either the PATTERN or the REPLACEMENT. Otherwise, if the PATTERN contains a **$** that looks like a variable rather than an end-of-string test, the variable will be interpolated into the pattern at run-time. If you only want the pattern compiled once the first time the variable is interpolated, use the **/o** option. If the pattern evaluates to a null string, the most recently executed (and successfully compiled) regular expression is used instead. See the *perlre* manpage for further explanation on these.

Options are:

- e Evaluate the right side as an expression.
- g Replace globally, i.e. all occurrences.
- i Do case-insensitive pattern matching.
- m Treat string as multiple lines.
- o Only compile pattern once.
- s Treat string as single line.
- x Use extended regular expressions.

Any non-alphanumeric, non-whitespace delimiter may replace the slashes. If single quotes are used, no interpretation is done on the replacement string (the **/e** modifier overrides this, however). If backquotes are used, the replacement string is a command to execute whose output will be used as the actual replacement text. If the PATTERN is delimited by bracketing quotes, the REPLACEMENT has its own pair of quotes, which may or may not be bracketing quotes, e.g. **s(foo)(bar)** or **s<foo>/bar/** . A **/e** will cause the replacement portion to be interpreter as a full-fledged Perl expression and *eval()* ed right then and there. It is, however, syntax checked at compile-time.

Examples:

s/\bgreen\b/mauve/g; # don't change wintergreen $path =~ s|/usr/bin|/usr/local/bin|; s/Login: $foo/Login: $bar/; # run-time pattern ($foo = $bar) =~ s/this/that/; $count = ($paragraph =~ s/Mister\b/Mr./g); $_ = 'abc123xyz'; s/\d+/$&*2/e; # yields 'abc246xyz' s/\d+/sprintf("%5d",$&)/e; # yields 'abc 246xyz' s/\w/$& x 2/eg; # yields 'aabbcc 224466xxyyzz' s/%(.)/$percent{$1}/g; # change percent escapes; no /e s/%(.)/$percent{$1} || $&/ge; # expr now, so /e s/^=(\w+)/&pod($1)/ge; # use function call # /e's can even nest; this will expand # simple embedded variables in $_ s/(\$\w+)/$1/eeg; # Delete C comments. $program =~ s { /* (?# Match the opening delimiter.) .*? (?# Match a minimal number of characters.) */ (?# Match the closing delimiter.) } []gsx; s/^\s*(.*?)\s*$/$1/; # trim white space s/([^]*) *([^]*)/$2 $1/; # reverse 1st two fields

Note the use of **$** instead of \ in the last example. Unlike **sed** , we only use the < *digit* > form in the left hand side. Anywhere else it's < *digit* >.

Occasionally, you can't just use a **/g** to get all the changes to occur. Here are two common cases:

put commas in the right places in an integer 1 while s/(.*\d)(\d\d\d)/$1,$2/g; # perl4 1 while s/(\d)(\d\d\d)(?!\d)/$1,$2/g; # perl5 # expand tabs to 8-column spacing 1 while s/\t+/' ' x (length($&)*8 - length($`)%8)/e;

tr/SEARCHLIST/REPLACEMENTLIST/cds

y/SEARCHLIST/REPLACEMENTLIST/cds

Translates all occurrences of the characters found in the search list with the corresponding character in the replacement list. It returns the number of characters replaced or deleted. If no string is specified via the =~ or !~ operator, the **$_** string is translated. (The string specified with =~ must be a scalar variable, an array element, or an assignment to one of those, i.e. an lvalue.) For **sed** devotees, **y** is provided as a synonym for **tr** . If the SEARCHLIST is delimited by bracketing quotes, the REPLACEMENTLIST has its own pair of quotes, which may or may not be bracketing quotes, e.g. **tr[A-Z][a-z]** or **tr(+-*/)/ABCD/** .

Options:

- c Complement the SEARCHLIST.
- d Delete found but unreplaced characters.
- s Squash duplicate replaced characters.

If the **/c** modifier is specified, the SEARCHLIST character set is complemented. If the **/d** modifier is specified, any characters specified by SEARCHLIST not found in REPLACEMENTLIST are deleted. (Note that this is slightly more flexible than the behavior of some **tr** programs, which delete anything they find in the SEARCHLIST, period.) If the **/s** modifier is specified, sequences of characters that were translated to the same character are squashed down to a single instance of the character.

If the **/d** modifier is used, the REPLACEMENTLIST is always interpreted exactly as specified. Otherwise, if the REPLACEMENTLIST is shorter than the SEARCHLIST, the final character is replicated till it is long enough. If the REPLACEMENTLIST is null, the SEARCHLIST is replicated. This latter is useful for counting characters in a class or for squashing character sequences in a class.

Examples:

$ARGV[1] =~ tr/A-Z/a-z/; # canonicalize to lower case $cnt = tr/*/*/; # count the stars in $_ $cnt = $sky =~ tr/*/*/; # count the stars in $sky $cnt = tr/0-9//; # count the digits in $_ tr/a-zA-Z//s; # bookkeeper -> bokeper ($HOST = $host) =~ tr/a-z/A-Z/; tr/a-zA-Z/ /cs; # change non-alphas to single space tr [\200-\377] [\000-\177]; # delete 8th bit

If multiple translations are given for a character, only the first one is used:

tr/AAA/XYZ/

will translate any A to X.

Note that because the translation table is built at compile time, neither the SEARCHLIST nor the REPLACEMENTLIST are subjected to double quote interpolation. That means that if you want to use variables, you must use an *eval()* :

eval "tr/$oldlist/$newlist/"; die $@ if $@; eval "tr/$oldlist/$newlist/, 1" or die $@;

.

I/O Operators

There are several I/O operators you should know about. A string is enclosed by backticks (grave accents) first undergoes variable substitution just like a double quoted string. It is then interpreted as a command, and the output of that command is the value of the pseudo-literal, like in a shell. In a scalar context, a single string consisting of all the output is returned. In a list context, a list of values is returned, one for each line of output. (You can set **$/** to use a different line terminator.) The command is executed each time the pseudo-literal is evaluated. The status value of the command is returned in **$?** (see the *perlvar* manpage for the interpretation of **$?**). Unlike in **csh** , no translation is done on the return data--newlines remain newlines. Unlike in any of the shells, single quotes do not hide variable names in the command from interpretation. To pass a **$** through to the shell you need to hide it with a backslash. The generalized form of backticks is qx// .

Evaluating a filehandle in angle brackets yields the next line from that file (newline included, so it's never false until end of file, at which time an undefined value is returned). Ordinarily you must assign that value to a variable, but there is one situation where an automatic assignment happens. *If and ONLY if* the input symbol is the only thing inside the conditional of a **while** loop, the value is automatically assigned to the variable **$_** . The assigned value is then tested to see if it is defined. (This may seem like an odd thing to you, but you'll use the construct in almost every Perl script you write.) Anyway, the following lines are equivalent to each other:

while (defined($_ = <STDIN>)) { print; } while (<STDIN>) { print; } for (;<STDIN>;) { print; } print while defined($_ = <STDIN>); print while <STDIN>;

The filehandles STDIN, STDOUT and STDERR are predefined. (The filehandles **stdin** , **stdout** and **stderr** will also work except in packages, where they would be interpreted as local identifiers rather than global.) Additional filehandles may be created with the *open()* function.

If a<FILEHANDLE> is used in a context that is looking for a list, a list consisting of all the input lines is returned, one line per list element. It's easy to make a *LARGE* data space this way, so use with care.

The null filehandle<> is special and can be used to emulate the behavior of **sed** and **awk** . Input from<> comes either from standard input, or from each file listed on the command line. Here's how it works: the first time<> is evaluated, the **@ARGV** array is checked, and if it is null, **$ARGV [0]** is set to "-", which when opened gives you standard input. The **@ARGV** array is then processed as a list of filenames. The loop

while (<>) { ... # code for each line }

is equivalent to the following Perl-like pseudo code:

unshift(@ARGV, '-') if $#ARGV < $[; while ($ARGV = shift) { open(ARGV, $ARGV); while (<ARGV>) { ... # code for each line } }

except that it isn't so cumbersome to say, and will actually work. It really does shift array **@ARGV** and put the current filename into variable **$ARGV** . It also uses filehandle *ARGV* internally-<> is just a synonym for<ARGV>, which is magical. (The pseudo code above doesn't work because it treats<ARGV> as non-magical.)

You can modify **@ARGV** before the first<> as long as the array ends up containing the list of filena you really want. Line numbers (**$.**) continue as if the input were one big happy file. (But see examp under *eof()* for how to reset line numbers on each file.)

If you want to set **@ARGV** to your own list of files, go right ahead. If you want to pass switches into your script, you can use one of the Getopts modules or put a loop on the front like this:

while ($_ = $ARGV[0], /^-/) { shift; last if /^--$/; if (/^-D(.*)/) { $debug = $1 } if (/^-v/) { $verbose++ } ... # other switches } while (<>) { ... # code for each line }

The<> symbol will return FALSE only once. If you call it again after this it will assume you are processing another **@ARGV** list, and if you haven't set **@ARGV** , will input from STDIN.

If the string inside the angle brackets is a reference to a scalar variable (e.g.< **$foo** >), then that variable contains the name of the filehandle to input from.

If the string inside angle brackets is not a filehandle, it is interpreted as a filename pattern to be globbed, and either a list of filenames or the next filename in the list is returned, depending on context. One level of **$** interpretation is done first, but you can't say **< $foo >** because that's an indirect filehandle as explained in the previous paragraph. You could insert curly brackets to force interpretation as a filename glob: **< ${ foo}>** . (Alternately, you can call the internal function directly as **glob($foo)** , which is probably the right way to have done it in the first place.) Example:

while (<*.c>) { chmod 0644, $_; }

is equivalent to

open(FOO, "echo *.c | tr -s ' \t\r\f' '\\012\\012\\012\\012'|"); while (<FOO>) { chop; chmod 0644, $_; }

In fact, it's currently implemented that way. (Which means it will not work on filenames with spaces in them unless you have csh(1) on your machine.) Of course, the shortest way to do the above is:

chmod 0644, <*.c>;

Because globbing invokes a shell, it's often faster to call *readdir()* yourself and just do your own *grep()* on the filenames. Furthermore, due to its current implementation of using a shell, the *glob()* routine may get "Arg list too long" errors (unless you've installed tcsh(1L) as */bin/csh*).

Constant Folding

Like C, Perl does a certain amount of expression evaluation at compile time, whenever it determines that all of the arguments to an operator are static and have no side effects. In particular, string concatenation happens at compile time between literals that don't do variable substitution. Backslash interpretation also happens at compile time. You can say

'Now is the time for all' . "\n" . 'good men to come to.'

and this all reduces to one string internally. Likewise, if you say

foreach $file (@filenames) { if (-s $file > 5 + 100 * 2**16) { ... } }

the compiler will pre-compute the number that expression represents so that the interpreter won't hav

Integer arithmetic

By default Perl assumes that it must do most of its arithmetic in floating point. But by saying

use integer;

you may tell the compiler that it's okay to use integer operations from here to the end of the enclosing BLOCK. An inner BLOCK may countermand this by saying

no integer;

which lasts until the end of that BLOCK.

NAME

perlre - Perl regular expressions

DESCRIPTION

For a description of how to use regular expressions in matching operations, see **m//** and **s///** in the *perlop* manpage . The matching operations can have various modifiers, some of which relate to the interpretation of the regular expression inside. These are:

i Do case-insensitive pattern matching. m Treat string as multiple lines. s Treat string as single line. x Use extended regular expressions.

These are usually written as "the **/x** modifier", even though the delimiter in question might not actually be a slash. In fact, any of these modifiers may also be embedded within the regular expression itself using the new **(?...)** construct. See below.

The **/x** modifier itself needs a little more explanation. It tells the regular expression parser to ignore whitespace that is not backslashed or within a character class. You can use this to break up your regular expression into (slightly) more readable parts. Together with the capability of embedding comments described later, this goes a long way towards making Perl 5 a readable language. See the C comment deletion code in the *perlop* manpage .

Regular Expressions

The patterns used in pattern matching are regular expressions such as those supplied in the Version 8 regexp routines. (In fact, the routines are derived (distantly) from Henry Spencer's freely redistributable reimplementation of the V8 routines.) See *Version 8 Regular Expressions* for details.

In particular the following metacharacters have their standard *egrep* -ish meanings:

\ Quote the next metacharacter ^ Match the beginning of the line . Match any character (except newline) $ Match the end of the line | Alternation () Grouping [] Character class

By default, the "^" character is guaranteed to match only at the beginning of the string, the "$" character only at the end (or before the newline at the end) and Perl does certain optimizations with the assumption that the string contains only one line. Embedded newlines will not be matched by "^" or "$". You may, however, wish to treat a string as a multi-line buffer, such that the "^" will match after any newline within the string, and "$" will match before any newline. At the cost of a little more overhead, you can do this by using the /m modifier on the pattern match operator. (Older programs did this by setting **$*** , but this practice is deprecated in Perl 5.)

To facilitate multi-line substitutions, the "." character never matches a newline unless you use the **/s** modifier, which tells Perl to pretend the string is a single line--even if it isn't. The **/s** modifier also overrides the setting of **$*** , in case you have some (badly behaved) older code that sets it in another module.

The following standard quantifiers are recognized:

* Match 0 or more times + Match 1 or more times ? Match 1 or 0 times {n} Match exactly n times {n,} Match at least n times {n,m} Match at least n but not more than m times

(If a curly bracket occurs in any other context, it is treated as a regular character.) The "*" modifier is equivalent to **{0,}** , the "+" modifier to **{1,}** , and the "?" modifier to **{0,1}** . There is no limit to the size of n or m, but large numbers will chew up more memory.

By default, a quantified subpattern is "greedy", that is, it will match as many times as possible without causing the rest pattern not to match. The standard quantifiers are all "greedy", in that they match as many occurrences as possible (given a particular starting location) without causing the pattern to fail. If you want it to match the minimum number of times possible, follow the quantifier with a "?" after any of them. Note that the meanings don't change, just the "gravity":

*? Match 0 or more times +? Match 1 or more times ?? Match 0 or 1 time {n}? Match exactly n times {n,}? Match at least n times {n,m}? Match at least n but not more than m times

Since patterns are processed as double quoted strings, the following also work:

\t tab \n newline \r return \f form feed \v vertical tab, whatever that is \a alarm (bell) \e escape \033 octal char \x1b hex char \c[control char \l lowercase next char \u uppercase next char \L lowercase till \E \U uppercase till \E \E end case modification \Q quote regexp metacharacters till \E

In addition, Perl defines the following:

\w Match a "word" character (alphanumeric plus "_") \W Match a non-word character \s Match a whitespace character \S Match a non-whitespace character \d Match a digit character \D Match a non-digit character

Note that **\w** matches a single alphanumeric character, not a whole word. To match a word you'd need to say **\w+** . You may use **\w** , **\W** , **\s** , **\S** , **\d** and **\D** within character classes (though not as either end of a range).

Perl defines the following zero-width assertions:

\b Match a word boundary \B Match a non-(word boundary) \A Match only at beginning of string \Z Match only at end of string \G Match only where previous m//g left off

A word boundary (**\b**) is defined as a spot between two characters that has a **\w** on one side of it and and a **\W** on the other side of it (in either order), counting the imaginary characters off the beginning and end of the string as matching a **\W** . (Within character classes **\b** represents backspace rather than a word boundary.) The **\A** and **\Z** are just like "^" and "$" except that they won't match multiple times when the **/m** modifier is used, while "^" and "$" will match at every internal line boundary.

When the bracketing construct **(...)** is used, <digit> matches the digit'th substring. (Outside of the

pattern, always use "$" instead of "\" in front of the digit. The scope of <digit> (and **$`** , **$&** , and **$'**) extends to the end of the enclosing BLOCK or eval string, or to the next pattern match with subexpressions. If you want to use parentheses to delimit subpattern (e.g. a set of alternatives) without saving it as a subpattern, follow the (with a ?. The <digit> notation sometimes works outside the current pattern, but should not be relied upon.) You may have as many parentheses as you wish. If you have more than 9 substrings, the variables **$10** , **$11** , ... refer to the corresponding substring. Within the pattern, \10, \11, etc. refer back to substrings if there have been at least that many left parens before the backreference. Otherwise (for backward compatibilty) \10 is the same as \010, a backspace, and \11 the same as \011, a tab. And so on. (\1 through \9 are always backreferences.)

$+** returns whatever the last bracket match matched. **$& returns the entire matched string. (**$0** used to return the same thing, but not any more.) **$`** returns everything before the matched string. **$'** returns everything after the matched string. Examples:

s/^([^]*) *([^]*)/$2 $1/; # swap first two words if (/Time: (..):(..):(..)/) { $hours = $1; $minutes = $2; $seconds = $3; }

You will note that all backslashed metacharacters in Perl are alphanumeric, such as **\b** , **\w** , **\n** . Unlike some other regular expression languages, there are no backslashed symbols that aren't alphanumeric. So anything that looks like \\, \(, \), <, \>, \{, or \} is always interpreted as a literal character, not a metacharacter. This makes it simple to quote a string that you want to use for a pattern but that you are afraid might contain metacharacters. Simply quote all the non-alphanumeric characters:

$pattern =~ s/(\W)/\\$1/g;

You can also use the built-in *quotemeta()* function to do this. An even easier way to quote metacharacters right in the match operator is to say

/$unquoted\Q$quoted\E$unquoted/

Perl 5 defines a consistent extension syntax for regular expressions. The syntax is a pair of parens with a question mark as the first thing within the parens (this was a syntax error in Perl 4). The character after the question mark gives the function of the extension. Several extensions are already supported:

(?#text)

A comment. The text is ignored.

(?:regexp)

This groups things like "()" but doesn't make backrefences like "()" does. So

split(/\b(?:a|b|c)\b/)

is like

split(/\b(a|b|c)\b/)

but doesn't spit out extra fields.

(?=regexp)

A zero-width positive lookahead assertion. For example, **/\w+(?=\t)/** matches a word followed by a tab, without including the tab in **$&** .

(?!regexp)

A zero-width negative lookahead assertion. For example **/foo(?!bar)/** matches any occurrence of "foo" that isn't followed by "bar". Note however that lookahead and lookbehind are NOT the same thing. You cannot use this for lookbehind: **/(?!foo)bar/** will not find an occurrence of "bar" that is preceded by something which is not "foo". That's because the **(?!foo)** is just saying that the next thing cannot be "foo"--and it's not, it's a "bar", so "foobar" will match. You would have to do something like **/(?foo)...bar/** for that. We say "like" because there's the case of your "bar" not having three characters before it. You could cover that this way: **/(?:(?!foo)...|^..?)bar/** . Sometimes it's still easier just to say:

if (/foo/ && $` =~ /bar$/)

(?imsx)

One or more embedded pattern-match modifiers. This is particularly useful for patterns that are specified in a table somewhere, some of which want to be case sensitive, and some of which don't. The case insensitive ones merely need to include **(?i)** at the front of the pattern. For example:

$pattern = "foobar"; if (/$pattern/i) # more flexible: $pattern = "(?i)foobar"; if (/$pattern/)

The specific choice of question mark for this and the new minimal.matching construct was because 1) question mark is pretty rare in older regular expressions, and 2) whenever you see one, you should stop and "question" exactly what is going on. That's psychology...

Version 8 Regular Expressions

In case you're not familiar with the "regular" Version 8 regexp routines, here are the pattern-matching rules not described above.

Any single character matches itself, unless it is a *metacharacter* with a special meaning described here or above. You can cause characters which normally function as metacharacters to be interpreted literally by prefixing them with a "\" (e.g. "\." matches a ".", not any character; "\\" matches a "\"). A series of characters matches that series of characters in the target string, so the pattern **blurfl** would match "blurfl" in the target string.

You can specify a character class, by enclosing a list of characters in **[]** , which will match any one of the characters in the list. If the first character after the "[" is "^", the class matches any character not in the list. Within a list, the "-" character is used to specify a range, so that **a-z** represents all the characters between "a" and "z", inclusive.

Characters may be specified using a metacharacter syntax much like that used in C: "\n" matches a newline, "\t" a tab, "\r" a carriage return, "\f" a form feed, etc. More generally, \ *nnn* , where *nnn* is a string of octal digits, matches the character whose ASCII value is *nnn* . Similarly, \x *nn* , where *nn* are hexidecimal digits, matches the character whose ASCII value is *nn* . The expression \c *x* matches the ASCII character control- *x* . Finally, the "." metacharacter matches any character except "\n" (unless you use **/s**).

You can specify a series of alternatives for a pattern using "|" to separate them, so that **fee|fie|foe** will match any of "fee", "fie", or "foe" in the target string (as would **f(e|i|o)e**). Note that the first alternative

includes everything from the last pattern delimiter ("(", "[", or the beginning of the pattern) up to the first "|", and the last alternative contains everything from the last "|" to the next pattern delimiter. For this reason, it's common practice to include alternatives in parentheses, to minimize confusion about where they start and end. Note however that "|" is interpreted as a literal with square brackets, so if you write **[fee|fie|foe]** you're really only matching **[feio|]** .

Within a pattern, you may designate subpatterns for later reference by enclosing them in parentheses, and you may refer back to the *n* th subpattern later in the pattern using the metacharacter \ *n* . Subpatterns are numbered based on the left to right order of their opening parenthesis. Note that a backreference matches whatever actually matched the subpattern in the string being examined, not the rules for that subpattern. Therefore, **(0|0x)\d*\s\1\d*** will match "0x1234 0x4321",but not "0x1234 01234", since subpattern 1 actually matched "0x", even though the rule **0|0x** could potentially match the leading 0 in the second number.

NAME

perlrun - how to execute the Perl interpreter

SYNOPSIS

perl [switches] filename args

DESCRIPTION

Upon startup, Perl looks for your script in one of the following places:

1. Specified line by line via -e switches on the command line.
2. Contained in the file specified by the first filename on the command line. (Note that systems supporting the #! notation invoke interpreters this way.)
3. Passed in implicitly via standard input. This only works if there are no filename arguments--to pass arguments to a STDIN script you must explicitly specify a "-" for the script name.

With methods 2 and 3, Perl starts parsing the input file from the.beginning, unless you've specified a -x switch, in which case it scans for the first line starting with #! and containing the word "perl", and starts there instead. This is useful for running a script embedded in a larger message. (In this case you would indicate the end of the script using the __END__ token.)

As of Perl 5, the #! line is always examined for switches as the line is being parsed. Thus, if you're on a machine that only allows one argument with the #! line, or worse, doesn't even recognize the #! line, you still can get consistent switch behavior regardless of how Perl was invoked, even if -x was used to find the beginning of the script.

Because many operating systems silently chop off kernel interpretation of the #! line after 32 characters, some switches may be passed in on the command line, and some may not; you could even get a "-" without its letter, if you're not careful. You probably want to make sure that all your switches fall either before or after that 32 character boundary. Most switches don't actually care if they're processed redundantly, but getting a - instead of a complete switch could cause Perl to try to execute standard input instead of your script. And a partial -I switch could also cause odd results.

Parsing of the #! switches starts wherever "perl" is mentioned in the line. The sequences "-*" and "- " are specifically ignored so that you could, if you were so inclined, say

#!/bin/sh -- # -*- perl -*- -p eval 'exec perl $0 -S ${1+"$@"}' if 0;

to let Perl see the -p switch.

If the #! line does not contain the word "perl", the program named after the #! is executed instead of the Perl interpreter. This is slightly bizarre, but it helps people on machines that don't do #!, because they can tell a program that their SHELL is /usr/bin/perl, and Perl will then dispatch the program to the correct interpreter for them.

After locating your script, Perl compiles the entire script to an internal form. If there are any compilation errors, execution of the script is not attempted. (This is unlike the typical shell script, which might run partway through before finding a syntax error.)

If the script is syntactically correct, it is executed. If the script runs off the end without hitting an *exit()* or *die()* operator, an implicit **exit(0)** is provided to indicate successful completion.

Switches

A single-character switch may be combined with the following switch, if any.

#!/usr/bin/perl -spi.bak # same as -s -p -i.bak

Switches include:

-0 *digits*

specifies the record separator (**$/**) as an octal number. If there are no digits, the null character is the separator. Other switches may precede or follow the digits. For example, if you have a version of **find** which can print filenames terminated by the null character, you can say this:

find . -name '*.bak' -print0 | perl -n0e unlink

The special value 00 will cause Perl to slurp files in paragraph mode. The value 0777 will cause Perl to slurp files whole since there is no legal character with that value.

-a

turns on autosplit mode when used with a -n or -p . An implicit split command to the **@F** array is done as the first thing inside the implicit while loop produced by the -n or -p .

perl -ane 'print pop(@F), "\n";'

is equivalent to

while (<>) { @F = split(' '); print pop(@F), "\n"; }

An alternate delimiter may be specified using -F .

-c

causes Perl to check the syntax of the script and then exit without executing it. Actually, it will execute **BEGIN** and **use** blocks, since these are considered part of the compilation.

-d

runs the script under the Perl debugger. See the *perldebug* manpage .

-D *number*

-D *list*

sets debugging flags. To watch how it executes your script, use **-D14** . (This only works if debugging is compiled into your Perl.) Another nice value is **-D1024** , which lists your compiled syntax tree. And **-D512** displays compiled regular expressions. As an alternative specify a list of letters instead of numbers (e.g. **-D14** is equivalent to **-Dtls**):

1 p Tokenizing and Parsing 2 s Stack Snapshots 4 l Label Stack Processing 8 t Trace Execution 16 o Operator Node Construction 32 c String/Numeric Conversions 64 P Print Preprocessor Command for -P 128 m Memory Allocation 256 f Format Processing 512 r Regular Expression Parsing 1024 x Syntax Tree Dump 2048 u Tainting Checks 4096 L Memory Leaks (not supported anymore) 8192 H Hash Dump -- usurps values() 16384 X Scratchpad Allocation 32768 D Cleaning Up

-e *commandline*

may be used to enter one line of script. If -e is given, Perl will not look for a script filename in the argument list. Multiple -e commands may be given to build up a multi-line script. Make sure to use semicolons where you would in a normal program.

-F *regexp*

specifies a regular expression to split on if -a is also in effect. If regexp has // around it, the slashes will be ignored.

-i *extension*

specifies that files processed by the <> construct are to be edited in-place. It does this by renaming the input file, opening the output file by the original name, and selecting that output file as the default for *print()* statements. The extension, if supplied, is added to the name of the old file to make a backup copy. If no extension is supplied, no backup is made. From the shell, saying

$ perl -p -i.bak -e "s/foo/bar/; ... "

is the same as using the script:

#!/usr/bin/perl -pi.bak s/foo/bar/;

which is equivalent to

#!/usr/bin/perl while (<>) { if ($ARGV ne $oldargv) { rename($ARGV, $ARGV . '.bak'); open(ARGVOUT, ">$ARGV"); select(ARGVOUT); $oldargv = $ARGV; } s/foo/bar/; } continue { print; # this prints to original filename } select(STDOUT);

except that the -i form doesn't need to compare **$ARGV** to **$oldargv** to know when the filename has changed. It does, however, use ARGVOUT for the selected filehandle. Note that STDOUT is restored as the default output filehandle after the loop.

You can use **eof** without parenthesis to locate the end of each input file, in case you want to append to each file, or reset line numbering (see example in *eof*).

-I *directory*

may be used in conjunction with -P to tell the C preprocessor where to look for include files. By default /usr/include and /usr/lib/perl are searched.

-l ***octnum***

enables automatic line-ending processing. It has two effects: first, it automatically chomps the line terminator when used with -n or -p , and second, it assigns " **$** " to have the value of *octnum* so that any print statements will have that line terminator added back on. If *octnum* is omitted, sets " **$** " to the current value of " **$/** ". For instance, to trim lines to 80 columns:

perl -lpe 'substr($_, 80) = """

Note that the assignment **$\ = $/** is done when the switch is processed, so the input record separator can be different than the output record separator if the -l switch is followed by a **-0** switch:

gnufind / -print0 | perl -ln0e 'print "found $_" if -p'

This sets $\ to newline and then sets $/ to the null character.

-n

causes Perl to assume the following loop around your script, which makes it iterate over filename arguments somewhat like **sed -n** or **awk** :

while (<>) { ... # your script goes here }

Note that the lines are not printed by default. See -p to have lines printed. Here is an efficient way to delete all files older than a week:

find . -mtime +7 -print | perl -nle 'unlink;'

This is faster than using the **-exec** switch of **find** because you don't have to start a process on every filename found.

BEGIN and **END** blocks may be used to capture control before or after the implicit loop, just as in **awk** .

-p

causes Perl to assume the following loop around your script, which makes it iterate over filename arguments somewhat like **sed** :

while (<>) { ... # your script goes here } continue { print; }

Note that the lines are printed automatically. To suppress printing use the -n switch. A -p overrides a -n switch.

BEGIN and **END** blocks may be used to capture control before or after the implicit loop, just as in awk.

-P

causes your script to be run through the C preprocessor before compilation by Perl. (Since both comments and cpp directives begin with the # character, you should avoid starting comments with any words recognized by the C preprocessor such as "if", "else" or "define".)

-s

enables some rudimentary switch parsing for switches on the command line after the script name but before any filename arguments (or before a **--**). Any switch found there is removed from **@ARGV** and sets the corresponding variable in the Perl script. The following script prints "true" if and only if the script is invoked with a **-xyz** switch.

#!/usr/bin/perl -s if ($xyz) { print "true\n"; }

-S

makes Perl use the PATH environment variable to search for the script (unless the name of the script starts with a slash). Typically this is used to emulate #! startup on machines that don't support #!, in the following manner:

#!/usr/bin/perl eval "exec /usr/bin/perl -S $0 $*" if $running_under_some_shell;

The system ignores the first line and feeds the script to /bin/sh, which proceeds to try to execute the Perl script as a shell script. The shell executes the second line as a normal shell command, and thus starts up the Perl interpreter. On some systems **$0** doesn't always contain the full pathname, so the -S tells Perl to search for the script if necessary. After Perl locates the script, it parses the lines and ignores them because the variable **$running_under_some_shell** is never true. A better construct than **$*** would be **${ 1+"$@"}** , which handles embedded spaces and such in the filenames, but doesn't work if the script is being interpreted by csh. In order to start up sh rather than csh, some systems may have to replace the #! line with a line containing just a colon, which will be politely ignored by Perl. Other systems can't control that, and need a totally devious construct that will work under any of csh, sh or Perl, such as the following:

eval '(exit $?0)' && eval 'exec /usr/bin/perl -S $0 ${1+"$@"}' & eval 'exec /usr/bin/perl -S $0 $argv:q' if 0;

-T

forces "taint" checks to be turned on. Ordinarily these checks are done only when running setuid or setgid. See the *perlsec* manpage .

-u

causes Perl to dump core after compiling your script. You can then take this core dump and turn it into an executable file by using the **undump** program (not supplied). This speeds startup at the expense of some disk space (which you can minimize by stripping the executable). (Still, a "hello world" executable comes out to about 200K on my machine.) If you want to execute a portion of your script before dumping, use the *dump()* operator instead. Note: availability of **undump** is platform specific and may not be available for a specific port of Perl.

-U

allows Perl to do unsafe operations. Currently the only "unsafe" operations are the unlinking of directories while running as superuser, and running setuid programs with fatal taint checks turned into warnings.

-v

prints the version and patchlevel of your Perl executable.

-w

prints warnings about identifiers that are mentioned only once, and scalar variables that are us before being set. Also warns about redefined subroutines, and references to undefined filehand or filehandles opened readonly that you are attempting to write on. Also warns you if you use values as a number that doesn't look like numbers, using an array as though it were a scalar, if your subroutines recurse more than 100 deep, and innumerable other things. See the *perldiag* manpage and the *perltrap* manpage .

-x *directory*

tells Perl that the script is embedded in a message. Leading garbage will be discarded until the first line that starts with #! and contains the string "perl". Any meaningful switches on that line will be applied (but only one group of switches, as with normal #! processing). If a directory name is specified, Perl will switch to that directory before running the script. The -x switch only controls the the disposal of leading garbage. The script must be terminated with **__END__** if there is trailing garbage to be ignored (the script can process any or all of the trailing garbage via the DATA filehandle if desired).

.

TOC

NAME

perlfunc - Perl builtin functions

DESCRIPTION

The functions in this section can serve as terms in an expression. They fall into two major categories: list operators and named unary operators. These differ in their precedence relationship with a following comma. (See the precedence table in the *perlop* manpage .) List operators take more than one argument, while unary operators can never take more than one argument. Thus, a comma terminates the argument of a unary operator, but merely separates the arguments of a list operator. A unary operator generally provides a scalar context to its argument, while a list operator may provide either scalar and list contexts for its arguments. If it does both, the scalar arguments will be first, and the list argument will follow. (Note that there can only ever be one list argument.) For instance, splice() has three scalar arguments followed by a list.

In the syntax descriptions that follow, list operators that expect a list (and provide list context for the elements of the list) are shown with LIST as an argument. Such a list may consist of any combination of scalar arguments or list values; the list values will be included in the list as if each individual element were interpolated at that point in the list, forming a longer single-dimensional list value. Elements of the LIST should be separated by commas.

Any function in the list below may be used either with or without parentheses around its arguments. (The syntax descriptions omit the parens.) If you use the parens, the simple (but occasionally surprising) rule is this: It *LOOKS* like a function, therefore it *IS* a function, and precedence doesn't matter. Otherwise it's a list operator or unary operator, and precedence does matter. And whitespace between the function and left parenthesis doesn't count--so you need to be careful sometimes:

print 1+2+3; # Prints 6. print(1+2) + 3; # Prints 3. print (1+2)+3; # Also prints 3! print +(1+2)+3; # Prints 6. print ((1+2)+3); # Prints 6.

If you run Perl with the **-w** switch it can warn you about this. For example, the third line above produces:

print (...) interpreted as function at - line 1. Useless use of integer addition in void context at - line 1.

For functions that can be used in either a scalar or list context, non-abortive failure is generally indicated in a scalar context by returning the undefined value, and in a list context by returning the null list.

Remember the following rule:

- * *THERE IS NO GENERAL RULE FOR CONVERTING A LIST INTO A SCALAR!*

Each operator and function decides which sort of value it would be most.appropriate to return in a scalar context. Some operators return the length of the list that would have been returned in a list context. Some

operators return the first value in the list. Some operators return the last value in the list. Some operators return a count of successful operations. In general, they do what you want, unless you want consistency.

-X FILEHANDLE

-X EXPR

-X

A file test, where X is one of the letters listed below. This unary operator takes one argument, either a filename or a filehandle, and tests the associated file to see if something is true about it. If the argument is omitted, tests **$_** , except for **-t** , which tests STDIN. Unless otherwise documented, it returns **1** for TRUE and '' for FALSE, or the undefined value if the file doesn't exist. Despite the funny names, precedence is the same as any other named unary operator, and the argument may be parenthesized like any other unary operator. The operator may be any of:

-r File is readable by effective uid/gid. -w File is writable by effective uid/gid. -x File is executable by effective uid/gid. -o File is owned by effective uid. -R File is readable by real uid/gid. -W File is writable by real uid/gid. -X File is executable by real uid/gid. -O File is owned by real uid. -e File exists. -z File has zero size. -s File has non-zero size (returns size). -f File is a plain file. -d File is a directory. -l File is a symbolic link. -p File is a named pipe (FIFO). -S File is a socket. -b File is a block special file. -c File is a character special file. -t Filehandle is opened to a tty. -u File has setuid bit set. -g File has setgid bit set. -k File has sticky bit set. -T File is a text file. -B File is a binary file (opposite of -T). -M Age of file in days when script started. -A Same for access time. -C Same for inode change time.

The interpretation of the file permission operators **-r** , **-R** , **-w** , **-W** , **-x** and -X is based solely on the mode of the file and the uids and gids of the user. There may be other reasons you can't actually read, write or execute the file. Also note that, for the superuser, **-r** , **-R** , **-w** and **-W** always return 1, and **-x** and -X return 1 if any execute bit is set in the mode. Scripts run by the superuser may thus need to do a stat() in order to determine the actual mode of the file, or temporarily set the uid to something else.

Example:

while (<>) { chop; next unless -f $_; # ignore specials ... }

Note that **-s/a/b/** does not do a negated substitution. Saying **-exp($foo)** still works as expected, however--only single letters following a minus are interpreted as file tests.

The **-T** and **-B** switches work as follows. The first block or so of the file is examined for odd characters such as strange control codes or characters with the high bit set. If too many odd characters (>30%) are found, it's a **-B** file, otherwise it's a **-T** file. Also, any file containing null in the first block is considered a binary file. If **-T** or **-B** is used on a filehandle, the current stdio buffer is examined rather than the first block. Both **-T** and **-B** return TRUE on a null file, or a file at EOF when testing a filehandle.

If any of the file tests (or either the stat() or lstat() operators) are given the special filehandle consisting of a solitary underline, then the stat structure of the previous file test (or stat operator) is used, saving a system call. (This doesn't work with **-t** , and you need to remember that lstat() and **-l** will leave values in the stat structure for the symbolic link, not the real file.) Example:

print "Can do.\n" if -r $a || -w _ || -x _; stat($filename); print "Readable\n" if -r _; print "Writable\n" if -w _; print "Executable\n" if -x _; print "Setuid\n" if -u _; print "Setgid\n" if -g _; print "Sticky\n" if -k _; print "Text\n" if -T _; print "Binary\n" if -B _;

abs VALUE

Returns the absolute value of its argument.

accept NEWSOCKET,GENERICSOCKET

Accepts an incoming socket connect, just as the accept(2) system call does. Returns the packed address if it succeeded, FALSE otherwise. See example in the *perlipc* manpage .

alarm SECONDS

Arranges to have a SIGALRM delivered to this process after the specified number of seconds have elapsed. (On some machines, unfortunately, the elapsed time may be up to one second less than you specified because of how seconds are counted.) Only one timer may be counting at once. Each call disables the previous timer, and an argument of 0 may be supplied to cancel the previous timer without starting a new one. The returned value is the amount of time remaining on the previous timer.

For sleeps of finer granularity than one second, you may use Perl's syscall() interface to access setitimer(2) if your system supports it, or else see *select* below.

atan2 Y,X

Returns the arctangent of Y/X in the range -PI to PI.

bind SOCKET,NAME

Binds a network address to a socket, just as the bind system call does. Returns TRUE if it succeeded, FALSE otherwise. NAME should be a packed address of the appropriate type for the socket. See example in the *perlipc* manpage .

binmode FILEHANDLE

Arranges for the file to be read or written in "binary" mode in operating systems that distinguish between binary and text files. Files that are not in binary mode have CR LF sequences translated to LF on input and LF translated to CR LF on output. Binmode has no effect under Unix; in DOS, it may be imperative. If FILEHANDLE is an expression, the value is taken as the name of the filehandle.

bless REF,PACKAGE

bless REF

This function tells the referenced object (passed as REF) that it is now an object in PACKAGE--or the current package if no PACKAGE is specified, which is the usual case. It returns the reference for convenience, since a bless() is often the last thing in a constructor. See the *perlobj* manpage for more about the blessing (and blessings) of objects.

caller EXPR

caller

Returns the context of the current subroutine call. In a scalar context, returns TRUE if there is a caller, that is, if we're in a subroutine or eval() or require() , and FALSE otherwise. In a list context, returns

($package, $filename, $line) = caller;

With EXPR, it returns some extra information that the debugger uses to print a stack trace. The value of EXPR indicates how many call frames to go back before the current one.

($package, $filename, $line, $subroutine, $hasargs, $wantargs) = caller($i);

Furthermore, when called from within the DB package, caller returns more detailed information: it sets sets the list variable **@DB:args** to be the arguments with which that subroutine was invoked.

chdir EXPR

Changes the working directory to EXPR, if possible. If EXPR is omitted, changes to home directory. Returns TRUE upon success, FALSE otherwise. See example under die() .

chmod LIST

Changes the permissions of a list of files. The first element of the list must be the numerical mode. Returns the number of files successfully changed.

$cnt = chmod 0755, 'foo', 'bar'; chmod 0755, @executables;

chomp VARIABLE

chomp LIST

chomp

This is a slightly safer version of chop (see below). It removes any line ending that corresponds to the current value of **$/** (also known as **$INPUT_RECORD_SEPARATOR** in the **English** module). It returns the number of characters removed. It's often used to remove the newline from the end of an input record when you're worried that the final record may be missing its newline. When in paragraph mode (**$/ = ""**), it removes all trailing newlines from the string. If VARIABLE is omitted, it chomps **$_** . Example:

while (<>) { chomp; # avoid \n on last field @array = split(/:/); ... }

You can actually chomp anything that's an lvalue, including an assignment:

chomp($cwd = `pwd`); chomp($answer = <STDIN>);

If you chomp a list, each element is chomped, and the total number of characters removed is returned.

chop VARIABLE

chop LIST

chop

Chops off the last character of a string and returns the character chopped. It's used primarily to remove the newline from the end of an input record, but is much more efficient than **s/\n//** because it neither scans nor copies the string. If VARIABLE is omitted, chops **$_** . Example:

while (<>) { chop; # avoid \n on last field @array = split(/:/); ... }

You can actually chop anything that's an lvalue, including an assignment:

chop($cwd = `pwd`); chop($answer = <STDIN>);

If you chop a list, each element is chopped. Only the value of the last chop is returned.

Note that chop returns the last character. To return all but the last character, use substr(**$string** , 0, -1) .

chown LIST

Changes the owner (and group) of a list of files. The first two elements of the list must be the *NUMERICAL* uid and gid, in that order. Returns the number of files successfully changed.

$cnt = chown $uid, $gid, 'foo', 'bar'; chown $uid, $gid, @filenames;

Here's an example that looks up non-numeric uids in the passwd file:

print "User: "; chop($user = <STDIN>); print "Files: " chop($pattern = <STDIN>); ($login,$pass,$uid,$gid) = getpwnam($user) or die "$user not in passwd file"; @ary = <${pattern}>; # expand filenames chown $uid, $gid, @ary;

chr NUMBER

Returns the character represented by that NUMBER in the character set. For example, chr(65) is "A" in ASCII.

chroot FILENAME

Does the same as the system call of that name. If you don't know what it does, don't worry about it. If FILENAME is omitted, does chroot to **$_** .

close FILEHANDLE

Closes the file or pipe associated with the file handle, returning TRUE only if stdio successfully flushes buffers and closes the system file descriptor. You don't have to close FILEHANDLE if you are immediately going to do another open on it, since open will close it for you. (See open() .) However, an explicit close on an input file resets the line counter ($.), while the implicit close done by open() does not. Also, closing a pipe will wait for the process executing on the pipe to complete, in case you want to look at the output of the pipe afterwards. Closing a pipe explicitly also puts the status value of the command into **$?** . Example:

open(OUTPUT, '|sort >foo'); # pipe to sort ... # print stuff to output close OUTPUT; # wait for sort to finish open(INPUT, 'foo'); # get sort's results

FILEHANDLE may be an expression whose value gives the real filehandle name.

closedir DIRHANDLE

Closes a directory opened by opendir() .

connect SOCKET,NAME

Attempts to connect to a remote socket, just as the connect system call does. Returns TRUE if it succeeded, FALSE otherwise. NAME should be a packed address of the appropriate type for the socket. See example in the *perlipc* manpage .

cos EXPR

Returns the cosine of EXPR (expressed in radians). If EXPR is omitted takes cosine of **$_** .

crypt PLAINTEXT,SALT

Encrypts a string exactly like the crypt(3) function in the C library. Useful for checking the password file for lousy passwords, amongst other things. Only the guys wearing white hats should do this.

Here's an example that makes sure that whoever runs this program knows their own password:

$pwd = (getpwuid($<))[1]; $salt = substr($pwd, 0, 2); system "stty -echo"; print "Password: "; chop($word = <STDIN>); print "\n"; system "stty echo"; if (crypt($word, $salt) ne $pwd) { die "Sorry...\n"; } else { print "ok\n"; }

Of course, typing in your own password to whoever asks you for it is unwise.

dbmclose ASSOC_ARRAY

[This function has been superseded by the untie() function.]

Breaks the binding between a DBM file and an associative array.

dbmopen ASSOC,DBNAME,MODE

[This function has been superseded by the tie() function.]

This binds a dbm(3) or ndbm(3) file to an associative array. ASSOC is the name of the associative array. (Unlike normal open, the first argument is *NOT* a filehandle, even though it looks like one). DBNAME is the name of the database (without the *.dir* or *.pag* extension). If the database does not exist, it is created with protection specified by MODE (as modified by the umask()). If your system only supports the older DBM functions, you may perform only one dbmopen() in your program. If your system has neither DBM nor ndbm, calling dbmopen() produces a fatal error.

If you don't have write access to the DBM file, you can only read associative array variables, not set them. If you want to test whether you can write, either use file tests or try setting a dummy array entry inside an eval() , which will trap the error.

Note that functions such as keys() and values() may return huge array values when used on large DBM files. You may prefer to use the each() function to iterate over large DBM files. Example:

print out history file offsets dbmopen(%HIST,'/usr/lib/news/history',0666); while (($key,$val) = each %HIST) { print $key, ' = ', unpack('L',$val), "\n"; } dbmclose(%HIST);

defined EXPR

Returns a boolean value saying whether the lvalue EXPR has a real value or not. Many operations return the undefined value under exceptional conditions, such as end of file, uninitialized variable, system error and such. This function allows you to distinguish between an undefined null scalar and a defined null scalar with operations that might return a real null string, such as referencing elements of an array. You may also check to see if arrays or subroutines exist. Use of defined on predefined variables is not guaranteed to produce intuitive results.

When used on a hash array element, it tells you whether the value is defined, not whether the key exists in the hash. Use exists() for that.

Examples:

print if defined $switch{'D'}; print "$val\n" while defined($val = pop(@ary)); die "Can't readlink $sym: $!" unless defined($value = readlink $sym); eval '@foo = ()' if defined(@foo); die "No XYZ package defined" unless defined %_XYZ; sub foo { defined &$bar ? &$bar(@_) : die "No bar"; }

See also undef() .

delete EXPR

Deletes the specified value from its hash array. Returns the deleted value, or the undefined value if nothing was deleted. Deleting from **$ENV {}** modifies the environment. Deleting from an array tied to a DBM file deletes the entry from the DBM file. (But deleting from a tie() d hash doesn't necessarily return anything.)

The following deletes all the values of an associative array:

foreach $key (keys %ARRAY) { delete $ARRAY{$key}; }

(But it would be faster to use the undef() command.) Note that the EXPR can be arbitrarily complicated as long as the final operation is a hash key lookup:

delete $ref->[$x][$y]{$key};

die LIST

Outside of an eval() , prints the value of LIST to **STDERR** and exits with the current value of $! (errno). If $! is 0, exits with the value of **($? >> 8)** (backtick `command` status). If **($? >> 8)** is 0, exits with 255. Inside an eval() , the error message is stuffed into **$@** , and the eval() is terminated with the undefined value.

Equivalent examples:

die "Can't cd to spool: $!\n" unless chdir '/usr/spool/news'; chdir '/usr/spool/news' or die "Can't cd to spool: $!\n"

If the value of EXPR does not end in a newline, the current script line number and input line number (if any) are also printed, and a newline is supplied. Hint: sometimes appending ", stopped" to your message will cause it to make better sense when the string "at foo line 123" is appended. Suppose you are running script "canasta".

die "/etc/games is no good"; die "/etc/games is no good, stopped";

produce, respectively

/etc/games is no good at canasta line 123. /etc/games is no good, stopped at canasta line 123.

See also exit() and warn() .

do BLOCK

Not really a function. Returns the value of the last command in the sequence of commands indicated by BLOCK. When modified by a loop modifier, executes the BLOCK once before

testing the loop condition. (On other statements the loop modifiers test the conditional first.)

do SUBROUTINE(LIST)

A deprecated form of subroutine call. See the *perlsub* manpage .

do EXPR

Uses the value of EXPR as a filename and executes the contents of the file as a Perl script. Its primary use is to include subroutines from a Perl subroutine library.

do 'stat.pl';

is just like

eval `cat stat.pl`;

except that it's more efficient, more concise, keeps track of the current filename for error messages, and searches all the **-I** libraries if the file isn't in the current directory (see also the **@INC** array in *Predefined Names*). It's the same, however, in that it does reparse the file every time you call it, so you probably don't want to do this inside a loop.

Note that inclusion of library modules is better done with the use() and require() operators.

dump LABEL

This causes an immediate core dump. Primarily this is so that you can use the **undump** program to turn your core dump into an executable binary after having initialized all your variables at the beginning of the program. When the new binary is executed it will begin by executing a goto LABEL (with all the restrictions that goto suffers). Think of it as a goto with an intervening core dump and reincarnation. If LABEL is omitted, restarts the program from the top. WARNING: any files opened at the time of the dump will NOT be open any more when the program is reincarnated, with possible resulting confusion on the part of Perl. See also **-u** option in the *perlrun* manpage .

Example:

#!/usr/bin/perl require 'getopt.pl'; require 'stat.pl'; %days = ('Sun' => 1, 'Mon' => 2, 'Tue' => 3, 'Wed' => 4, 'Thu' => 5, 'Fri' => 6, 'Sat' => 7,); dump QUICKSTART if $ARGV[0] eq '-d'; QUICKSTART: Getopt('f');

each ASSOC_ARRAY

Returns a 2 element array consisting of the key and value for the next value of an associative array, so that you can iterate over it. Entries are returned in an apparently random order. When the array is entirely read, a null array is returned (which when assigned produces a FALSE (0) value). The next call to each() after that will start iterating again. The iterator can be reset only by reading all the elements from the array. You should not add elements to an array while you're iterating over it. There is a single iterator for each associative array, shared by all each() , keys() and values() function calls in the program. The following prints out your environment like the printenv(1) program, only in a different order:

while (($key,$value) = each %ENV) { print "$key=$value\n"; }

See also keys() and values() .

eof FILEHANDLE

eof

Returns 1 if the next read on FILEHANDLE will return end of file, or if FILEHANDLE is not open. FILEHANDLE may be an expression whose value gives the real filehandle name. (Note that this function actually reads a character and then *ungetc()* s it, so it is not very useful in an interactive context.) Do not read from a terminal file (or call eof(FILEHANDLE) on it) after end-of-file is reached. Filetypes such as terminals may lose the end-of-file condition if you do.

An eof without an argument uses the last file read as argument. Empty parentheses () may be used to indicate the pseudo file formed of the files listed on the command line, i.e. eof() is reasonable to use inside a while <>) loop to detect the end of only the last file. Use eof(ARGV) or eof without the parentheses to test *EACH* file in a while <>) loop. Examples:

reset line numbering on each input file while (<>) { print "$.\t$_"; close(ARGV) if (eof); # Not eof(). } # insert dashes just before last line of last file while (<>) { if (eof()) { print "----------\n"; close(ARGV); # close or break; is needed if we # are reading from the terminal } print; }

Practical hint: you almost never need to use eof in Perl, because the input operators return undef when they run out of data.

eval EXPR

eval BLOCK

EXPR is parsed and executed as if it were a little Perl program. It is executed in the context of the current Perl program, so that any variable settings, subroutine or format definitions remain afterwards. The value returned is the value of the last expression evaluated, or a return statement may be used, just as with subroutines.

If there is a syntax error or runtime error, or a die() statement is executed, an undefined value is returned by eval() , and **$@** is set to the error message. If there was no error, **$@** is guaranteed to be a null string. If EXPR is omitted, evaluates **$_** . The final semicolon, if any, may be omitted from the expression.

Note that, since eval() traps otherwise-fatal errors, it is useful for determining whether a particular feature (such as dbmopen() or symlink()) is implemented. It is also Perl's exception trapping mechanism, where the die operator is used to raise exceptions.

If the code to be executed doesn't vary, you may use the eval-BLOCK form to trap run-time errors without incurring the penalty of recompiling each time. The error, if any, is still returned in **$@** . Examples:

make divide-by-zero non-fatal eval { $answer = $a / $b; }; warn $@ if $@; # same thing, but less efficient eval '$answer = $a / $b'; warn $@ if $@; # a compile-time error eval { $answer = }; # a run-time error eval '$answer ='; # sets $@

With an eval() , you should be especially careful to remember what's being looked at when:

eval $x; # CASE 1 eval "$x"; # CASE 2 eval '$x'; # CASE 3 eval { $x }; # CASE 4 eval "\$$x # CASE 5 $$x++; # CASE 6

Cases 1 and 2 above behave identically: they run the code contained in the variable **$x** . (Although case 2 has misleading double quotes making the reader wonder what else might be happening (nothing is).) Cases 3 and 4 likewise behave in the same way: they run the code< **$x** >, which does nothing at all. (Case 4 is preferred for purely visual reasons.) Case 5 is a place where normally you *WOULD* like to use double quotes, except in that particular situation, you can just use symbolic references instead, as in case 6.

exec LIST

The exec() function executes a system command *AND NEVER RETURNS* . Use the system() function if you want it to return.

If there is more than one argument in LIST, or if LIST is an array with more than one value, calls execvp(3) with the arguments in LIST. If there is only one scalar argument, the argument is checked for shell metacharacters. If there are any, the entire argument is passed to **/bin/sh -c** for parsing. If there are none, the argument is split into words and passed directly to *execvp()* , which is more efficient. Note: exec() (and system(0) do not flush your output buffer, so you may need to set **$|** to avoid lost output. Examples:

exec '/bin/echo', 'Your arguments are: ', @ARGV; exec "sort $outfile | uniq";

If you don't really want to execute the first argument, but want to lie to the program you are executing about its own name, you can specify the program you actually want to run as an "indirect object" (without a comma) in front of the LIST. (This always forces interpretation of the LIST as a multi-valued list, even if there is only a single scalar in the list.) Example:

$shell = '/bin/csh'; exec $shell '-sh'; # pretend it's a login shell

or, more directly,

exec {'/bin/csh'} '-sh'; # pretend it's a login shell

exists EXPR

Returns TRUE if the specified hash key exists in its hash array, even if the corresponding value is undefined.

print "Exists\n" if exists $array{$key}; print "Defined\n" if defined $array{$key}; print "True\n" if $array{$key};

A hash element can only be TRUE if it's defined, and defined if it exists, but the reverse doesn't necessarily hold true.

Note that the EXPR can be arbitrarily complicated as long as the final operation is a hash key lookup:

if (exists $ref->[$x][$y]{$key}) { ... }

exit EXPR

Evaluates EXPR and exits immediately with that value. (Actually, it calls any defined **END**

routines first, but the **END** routines may not abort the exit. Likewise any object destructors that need to be called are called before exit.) Example:

$ans = <STDIN>; exit 0 if $ans =~ /^[Xx]/;

See also die() . If EXPR is omitted, exits with 0 status.

exp EXPR

Returns *e* (the natural logarithm base) to the power of EXPR. If EXPR is omitted, gives exp(**$_**) .

fcntl FILEHANDLE,FUNCTION,SCALAR

Implements the fcntl(2) function. You'll probably have to say

use Fcntl;

first to get the correct function definitions. Argument processing and value return works just like ioctl() below. Note that fcntl() will produce a fatal error if used on a machine that doesn't implement fcntl(2). For example:

use Fcntl; fcntl($filehandle, F_GETLK, $packed_return_buffer);

fileno FILEHANDLE

Returns the file descriptor for a filehandle. This is useful for constructing bitmaps for select() . If FILEHANDLE is an expression, the value is taken as the name of the filehandle.

flock FILEHANDLE,OPERATION

Calls flock(2) on FILEHANDLE. See the flock (2) manpage for definition of OPERATION. Returns TRUE for success, FALSE on failure. Will produce a fatal error if used on a machine that doesn't implement flock(2). Here's a mailbox appender for BSD systems.

$LOCK_SH = 1; $LOCK_EX = 2; $LOCK_NB = 4; $LOCK_UN = 8; sub lock { flock(MBOX,$LOCK_EX); # and, in case someone appended # while we were waiting... seek(MBOX, 0, 2); } sub unlock { flock(MBOX,$LOCK_UN); }

```
open(MBOX, >>/usr/spool/mail/ $ENV {'USER'}")
        or die "Can't open mailbox: $!";
lock();
print MBOX $msg,"\n\n";
unlock();
```

Note that flock() can't lock things over the network. You need to do locking with fcntl() for that.

fork

Does a fork(2) system call. Returns the child pid to the parent process and 0 to the child process, or undef if the fork is unsuccessful. Note: unflushed buffers remain unflushed in both processes, which means you may need to set **$|** (**$AUTOFLUSH** in English) or call the *autoflush()* FileHandle method to avoid duplicate output.

If you fork() without ever waiting on your children, you will accumulate zombies:

$SIG{'CHLD'} = sub { wait };

There's also the double-fork trick (error checking on fork() returns omitted);

unless ($pid = fork) { unless (fork) { exec "what you really wanna do"; die "no exec"; # ... or ... some_perl_code_here; exit 0; } exit 0; } waitpid($pid,0);

formline PICTURE, LIST

This is an internal function used by formats, though you may call it too. It formats (see the *perlform* manpage) a list of values according to the contents of PICTURE, placing the output into the format output accumulator, **$^ A** . Eventually, when a write() is done, the contents of **$^ A** are written to some filehandle, but you could also read **$^ A** yourself and then set **$^ A** back to "". Note that a format typically does one formline() per line of form, but the formline() function itself doesn't care how many newlines are embedded in the PICTURE. This means that the ~ and ~~ tokens will treat the entire PICTURE as a single line. You may therefore need to use multiple formlines to implement a single record format, just like the format compiler.

Be careful if you put double quotes around the picture, since an " **@** " character may be taken to mean the beginning of an array name. formline() always returns TRUE.

getc FILEHANDLE

getc

Returns the next character from the input file attached to FILEHANDLE, or a null string at end of file. If FILEHANDLE is omitted, reads from STDIN.

getlogin

Returns the current login from */etc/utmp* , if any. If null, use getpwuid() .

$login = getlogin || (getpwuid($<))[0] || "Kilroy";

getpeername SOCKET

Returns the packed sockaddr address of other end of the SOCKET connection.

An internet sockaddr $sockaddr = 'S n a4 x8'; $hersockaddr = getpeername(S); ($family, $port, $heraddr) = unpack($sockaddr,$hersockaddr);

getpgrp PID

Returns the current process group for the specified PID, 0 for the current process. Will produce a fatal error if used on a machine that doesn't implement getpgrp(2). If PID is omitted, returns process group of current process.

getppid

Returns the process id of the parent process.

getpriority WHICH,WHO

Returns the current priority for a process, a process group, or a user. (See the getpriority (2) manpage.) Will produce a fatal error if used on a machine that doesn't implement getpriority(2).

getpwnam NAME

getgrnam NAME

gethostbyname NAME
getnetbyname NAME
getprotobyname NAME
getpwuid UID
getgrgid GID
getservbyname NAME,PROTO
gethostbyaddr ADDR,ADDRTYPE
getnetbyaddr ADDR,ADDRTYPE
getprotobynumber NUMBER
getservbyport PORT,PROTO
getpwent
getgrent
gethostent
getnetent
getprotoent
getservent
setpwent
setgrent
sethostent STAYOPEN
setnetent STAYOPEN
setprotoent STAYOPEN
setservent STAYOPEN
endpwent
endgrent
endhostent
endnetent
endprotoent
endservent

These routines perform the same functions as their counterparts in the system library. Within a list context, the return values from the various get routines are as follows:

($name,$passwd,$uid,$gid, $quota,$comment,$gcos,$dir,$shell) = getpw*
($name,$passwd,$gid,$members) = getgr* ($name,$aliases,$addrtype,$length,@addrs) = gethost*
($name,$aliases,$addrtype,$net) = getnet* ($name,$aliases,$proto) = getproto*
($name,$aliases,$port,$proto) = getserv*

(If the entry doesn't exist you get a null list.)

Within a scalar context, you get the name, unless the function was a lookup by name, in which case you get the other thing, whatever it is. (If the entry doesn't exist you get the undefined value.)

For example:

$uid = getpwnam $name = getpwuid $name = getpwent $gid = getgrnam $name = getgrgid $name = getgrent etc.

The **$members** value returned by *getgr*()* is a space separated list of the login names of the members of the group.

For the *gethost*()* functions, if the **h_errno** variable is supported in C, it will be returned to you via **$?** if the function call fails. The **@addrs** value returned by a successful call is a list of the raw addresses returned by the corresponding system library call. In the Internet domain, each address is four bytes long and you can unpack it by saying something like:

($a,$b,$c,$d) = unpack('C4',$addr[0]);

getsockname SOCKET

Returns the packed sockaddr address of this end of the SOCKET connection.

An internet sockaddr $sockaddr = 'S n a4 x8'; $mysockaddr = getsockname(S); ($family, $port, $myaddr) = unpack($sockaddr,$mysockaddr);

getsockopt SOCKET,LEVEL,OPTNAME

Returns the socket option requested, or undefined if there is an error.

glob EXPR

Returns the value of EXPR with filename expansions such as a shell would do. This is the internal function implementing the<*.*> operator.

gmtime EXPR

Converts a time as returned by the time function to a 9-element array with the time localized for the Greenwich timezone. Typically used as follows:

($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = gmtime(time);

All array elements are numeric, and come straight out of a struct tm. In particular this means that **$mon** has the range 0..11 and **$wday** has the range 0..6. If EXPR is omitted, does gmtime(time()) .

goto LABEL

goto EXPR

goto &NAME

The goto-LABEL form finds the statement labeled with LABEL and resumes execution there. It may not be used to go into any construct that requires initialization, such as a subroutine or a foreach loop. It also can't be used to go into a construct that is optimized away. It can be used to go almost anywhere else within the dynamic scope, including out of subroutines, but it's usually better to use some other construct such as last or die. The author of Perl has never felt the need to use this form of goto (in Perl, that is--C is another matter).

The goto-EXPR form expects a label name, whose scope will be resolved dynamically. This allows for computed gotos per FORTRAN, but isn't necessarily recommended if you're optimizing

for maintainability:

goto ("FOO", "BAR", "GLARCH")[$i];

The goto-&NAME form is highly magical, and substitutes a call to the named subroutine for the currently running subroutine. This is used by AUTOLOAD subroutines that wish to load another subroutine and then pretend that the other subroutine had been called in the first place (except that any modifications to **@_** in the current subroutine are propagated to the other subroutine.) After the goto, not even caller() will be able to tell that this routine was called first.

grep BLOCK LIST

grep EXPR,LIST

Evaluates the BLOCK or EXPR for each element of LIST (locally setting **$_** to each element) and returns the list value consisting of those elements for which the expression evaluated to TRUE. In a scalar context, returns the number of times the expression was TRUE.

@foo = grep(!/^#/, @bar); # weed out comments

or equivalently,

@foo = grep {!/^#/} @bar; # weed out comments

Note that, since **$_** is a reference into the list value, it can be used to modify the elements of the array. While this is useful and supported, it can cause bizarre results if the LIST is not a named array.

hex EXPR

Returns the decimal value of EXPR interpreted as an hex string. (To interpret strings that might start with 0 or 0x see oct() .) If EXPR is omitted, uses **$_** .

import

There is no built-in import() function. It is merely an ordinary method subroutine defined (or inherited) by modules that wish to export names to another module. The use() function calls the import() method for the package used. See also *use* and the *perlmod* manpage .

index STR,SUBSTR,POSITION

index STR,SUBSTR

Returns the position of the first occurrence of SUBSTR in STR at or after POSITION. If POSITION is omitted, starts searching from the beginning of the string. The return value is based at 0, or whatever you've set the $[variable to. If the substring is not found, returns one less than the base, ordinarily -1.

int EXPR

Returns the integer portion of EXPR. If EXPR is omitted, uses **$_** .

ioctl FILEHANDLE,FUNCTION,SCALAR

Implements the ioctl(2) function. You'll probably have to say

require "ioctl.ph"; # probably /usr/local/lib/perl/ioctl.ph

first to get the correct function definitions. If ioctl.ph doesn't exist or doesn't have the correct definitions you'll have to roll your own, based on your C header files such as<sys/ioctl.h>. (There is a Perl script called **h2ph** that comes with the Perl kit which may help you in this.) SCALAR will be read and/or written depending on the FUNCTION--a pointer to the string value of SCALAR will be passed as the third argument of the actual ioctl call. (If SCALAR has no string value but does have a numeric value, that value will be passed rather than a pointer to the string value. To guarantee this to be TRUE, add a 0 to the scalar before using it.) The pack() and unpack() functions are useful for manipulating the values of structures used by ioctl() . The following example sets the erase character to DEL.

require 'ioctl.ph'; $sgttyb_t = "ccccs"; # 4 chars and a short if (ioctl(STDIN,$TIOCGETP,$sgttyb)) { @ary = unpack($sgttyb_t,$sgttyb); $ary[2] = 127; $sgttyb = pack($sgttyb_t,@ary); ioctl(STDIN,$TIOCSETP,$sgttyb) || die "Can't ioctl: $!"; }

The return value of ioctl (and fcntl) is as follows:

if OS returns: then Perl returns: -1 undefined value 0 string "0 but true" anything else that number

Thus Perl returns TRUE on success and FALSE on failure, yet you can still easily determine the actual value returned by the operating system:

($retval = ioctl(...)) || ($retval = -1); printf "System returned %d\n", $retval;

join EXPR,LIST

Joins the separate strings of LIST or ARRAY into a single string with fields separated by the value of EXPR, and returns the string. Example:

$_ = join(':', $login,$passwd,$uid,$gid,$gcos,$home,$shell);

See *split* .

keys ASSOC_ARRAY

Returns a normal array consisting of all the keys of the named associative array. (In a scalar context, returns the number of keys.) The keys are returned in an apparently random order, but it is the same order as either the values() or each() function produces (given that the associative array has not been modified). Here is yet another way to print your environment:

@keys = keys %ENV; @values = values %ENV; while ($#keys >= 0) { print pop(@keys), '=', pop(@values), "\n"; }

or how about sorted by key:

foreach $key (sort(keys %ENV)) { print $key, '=', $ENV{$key}, "\n"; }

kill LIST

Sends a signal to a list of processes. The first element of the list must be the signal to send. Returns the number of processes successfully signaled.

$cnt = kill 1, $child1, $child2; kill 9, @goners;

Unlike in the shell, in Perl if the *SIGNAL* is negative, it kills process groups instead of processes.

(On System V, a negative *PROCESS* number will also kill process groups, but that's not portal That means you usually want to use positive not negative signals. You may also use a signal n in quotes.

last LABEL

last

The last command is like the **break** statement in C (as used in loops); it immediately exits the loop in question. If the LABEL is omitted, the command refers to the innermost enclosing loop. The **continue** block, if any, is not executed:

line: while (<STDIN>) { last line if /^$/; # exit when done with header ... }

lc EXPR

Returns an lowercased version of EXPR. This is the internal function implementing the \L escape in double-quoted strings.

lcfirst EXPR

Returns the value of EXPR with the first character lowercased. This is the internal function implementing the \l escape in double-quoted strings.

length EXPR

Returns the length in characters of the value of EXPR. If EXPR is omitted, returns length of **$_** .

link OLDFILE,NEWFILE

Creates a new filename linked to the old filename. Returns 1 for success, 0 otherwise.

listen SOCKET,QUEUESIZE

Does the same thing that the listen system call does. Returns TRUE if it succeeded, FALSE otherwise. See example in the *perlipc* manpage .

local EXPR

In general, you should be using "my" instead of "local", because it's faster and safer. Format variables often use "local" though, as do other variables whose current value must be visible to called subroutines. This is known as dynamic scoping. Lexical scoping is done with "my", which works more like C's auto declarations.

A local modifies the listed variables to be local to the enclosing block, subroutine, eval or "do". If more than one value is listed, the list must be placed in parens. All the listed elements must be legal lvalues. This operator works by saving the current values of those variables in LIST on a hidden stack and restoring them upon exiting the block, subroutine or eval. This means that called subroutines can also reference the local variable, but not the global one. The LIST may be assigned to if desired, which allows you to initialize your local variables. (If no initializer is given for a particular variable, it is created with an undefined value.) Commonly this is used to name the parameters to a subroutine. Examples:

sub RANGEVAL { local($min, $max, $thunk) = @_; local $result = ''; local $i; # Presumably $thunk makes reference to $i for ($i = $min; $i < $max; $i++) { $result .= eval $thunk; } $result; } if ($sw eq '-v') { # init local array with global array local @ARGV = @ARGV; unshift(@ARGV,'echo'); system @ARGV; } # @ARGV restored # temporarily add to digits associative array if ($base12) { # (NOTE: not claiming this is efficient!) local(%digits) =

(%digits,'t',10,'e',11); parse_num(); }

Note that local() is a run-time command, and so gets executed every time through a loop. In Perl 4 it used more stack storage each time until the loop was exited. Perl 5 reclaims the space each time through, but it's still more efficient to declare your variables outside the loop.

A local is simply a modifier on an lvalue expression. When you assign to a localized EXPR, the local doesn't change whether EXPR is viewed as a scalar or an array. So

local($foo) = <STDIN>; local @FOO = <STDIN>;

both supply a list context to the righthand side, while

local $foo = <STDIN>;

supplies a scalar context.

localtime EXPR

Converts a time as returned by the time function to a 9-element array with the time analyzed for the local timezone. Typically used as follows:

($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime(time);

All array elements are numeric, and come straight out of a struct tm. In particular this means that **$mon** has the range 0..11 and **$wday** has the range 0..6. If EXPR is omitted, does localtime(time).

In a scalar context, prints out the ctime(3) value:

$now_string = localtime; # e.g. "Thu Oct 13 04:54:34 1994"

See also *timelocal* and the strftime(3) function available via the POSIX modulie.

log EXPR

Returns logarithm (base *e*) of EXPR. If EXPR is omitted, returns log of **$_** .

lstat FILEHANDLE

lstat EXPR

Does the same thing as the stat() function, but stats a symbolic link instead of the file the symbolic link points to. If symbolic links are unimplemented on your system, a normal stat() is done.

m//

The match operator. See the *perlop* manpage .

map BLOCK LIST

map EXPR,LIST

Evaluates the BLOCK or EXPR for each element of LIST (locally setting **$_** to each element) and returns the list value composed of the results of each such evaluation. Evaluates BLOCK or EXPR in a list context, so each element of LIST may produce zero, one, or more elements in the returned value.

@chars = map(chr, @nums);

translates a list of numbers to the corresponding characters. And

%hash = map {&key($_), $_} @array;

is just a funny way to write

%hash = (); foreach $_ (@array) { $hash{&key($_)} = $_; }

mkdir FILENAME,MODE

Creates the directory specified by FILENAME, with permissions specified by MODE (as modified by umask). If it succeeds it returns 1, otherwise it returns 0 and sets $! (errno).

msgctl ID,CMD,ARG

Calls the System V IPC function msgctl. If CMD is &IPC_STAT, then ARG must be a variable which will hold the returned msqid_ds structure. Returns like ioctl: the undefined value for error, "0 but true" for zero, or the actual return value otherwise.

msgget KEY,FLAGS

Calls the System V IPC function msgget. Returns the message queue id, or the undefined value if there is an error.

msgsnd ID,MSG,FLAGS

Calls the System V IPC function msgsnd to send the message MSG to the message queue ID. MSG must begin with the long integer message type, which may be created with pack("L", **$type**) . Returns TRUE if successful, or FALSE if there is an error.

msgrcv ID,VAR,SIZE,TYPE,FLAGS

Calls the System V IPC function msgrcv to receive a message from message queue ID into variable VAR with a maximum message size of SIZE. Note that if a message is received, the message type will be the first thing in VAR, and the maximum length of VAR is SIZE plus the size of the message type. Returns TRUE if successful, or FALSE if there is an error.

my EXPR

A "my" declares the listed variables to be local (lexically) to the enclosing block, subroutine, eval or "do". If more than one value is listed, the list must be placed in parens. All the listed elements must be legal lvalues. Only alphanumeric identifiers may be lexically scoped--magical builtins like $/ must be localized with "local" instead. In particular, you're not allowed to say

my $_; # Illegal.

Unlike the "local" declaration, variables declared with "my" are totally hidden from the outside world, including any called subroutines (even if it's the same subroutine--every call gets its own copy).

(An eval() , however, can see the lexical variables of the scope it is being evaluated in so long as the names aren't hidden by declarations within the eval() itself. See the *perlref* manpage .)

The EXPR may be assigned to if desired, which allows you to initialize your variables. (If no initializer is given for a particular variable, it is created with an undefined value.) Commonly this is used to name the parameters to a subroutine. Examples:

sub RANGEVAL { my($min, $max, $thunk) = @_; my $result = ''; my $i; # Presumably $thu
makes reference to $i for ($i = $min; $i < $max; $i++) { $result .= eval $thunk; } $result; } if
eq '-v') { # init my array with global array my @ARGV = @ARGV; unshift(@ARGV,'echo');
system @ARGV; } # Outer @ARGV again visible

The "my" is simply a modifier on something you might assign to. So when you do assign to the EXPR, the "my" doesn't change whether EXPR is viewed as a scalar or an array. So

my ($foo) = <STDIN>; my @FOO = <STDIN>;

both supply a list context to the righthand side, while

my $foo = <STDIN>;

supplies a scalar context. But the following only declares one variable:

my $foo, $bar = 1;

That has the same effect as

my $foo; $bar = 1;

The declared variable is not introduced (is not visible) until after the current statement. Thus,

my $x = $x;

can be used to initialize the new **$x** with the value of the old **$x** , and the expression

my $x = 123 and $x == 123

is false unless the old **$x** happened to have the value 123.

Some users may wish to encourage the use of lexically scoped variables. As an aid to catching implicit references to package variables, if you say

use strict 'vars';

then any variable reference from there to the end of the enclosing block must either refer to a lexical variable, or must be fully qualified with the package name. A compilation error results otherwise. An inner block may countermand this with S<"no strict 'vars'">.

next LABEL

next

The next command is like the **continue** statement in C; it starts the next iteration of the loop:

line: while (<STDIN>) { next line if /^#/; # discard comments ... }

Note that if there were a **continue** block on the above, it would get executed even on discarded lines. If the LABEL is omitted, the command refers to the innermost enclosing loop.

no Module LIST

See the "use" function, which "no" is the opposite of.

oct EXPR

Returns the decimal value of EXPR interpreted as an octal string. (If EXPR happens to start off with 0x, interprets it as a hex string instead.) The following will handle decimal, octal, and hex in the standard Perl or C notation:

$val = oct($val) if $val =~ /^0/;

If EXPR is omitted, uses **$_** .

open FILEHANDLE,EXPR

open FILEHANDLE

Opens the file whose filename is given by EXPR, and associates it with FILEHANDLE. If FILEHANDLE is an expression, its value is used as the name of the real filehandle wanted. If EXPR is omitted, the scalar variable of the same name as the FILEHANDLE contains the filename. If the filename begins with <" or nothing, the file is opened for input. If the filename begins with ">", the file is opened for output. If the filename begins with >>", the file is opened for appending. (You can put a '+' in front of the '>' or <' to indicate that you want both read and write access to the file.) If the filename begins with "|", the filename is interpreted as a command to which output is to be piped, and if the filename ends with a "|", the filename is interpreted as command which pipes input to us. (You may not have a command that pipes both in and out.) Opening '-' opens STDIN and opening '>-' opens STDOUT. Open returns non-zero upon success, the undefined value otherwise. If the open involved a pipe, the return value happens to be the pid of the subprocess. Examples:

$ARTICLE = 100; open ARTICLE or die "Can't find article $ARTICLE: $!\n"; while (<ARTICLE>) {...

```
open(LOG, >>/usr/spool/news/twitlog'); # (log is reserved)
open(article, "caesar <$article |");    # decrypt article
open(extract, "|sort >/tmp/Tmp$$");     # $$ is our process id
# process argument list of files along with any includes
foreach $file (@ARGV) {
    process($file, 'fh00');
}
sub process {
    local($filename, $input) = @_;
    $input++;               # this is a string increment
    unless (open($input, $filename)) {
        print STDERR "Can't open $filename: $!\n";
        return;
    }
    while (<$input>) {              # note use of indirection
        if (/^#include "(.*)"/) {
            process($1, $input);
            next;
        }
        ...         # whatever
    }
```


```
    }
```

You may also, in the Bourne shell tradition, specify an EXPR beginning with ">&", in which case the rest of the string is interpreted as the name of a filehandle (or file descriptor, if numeric) which is to be duped and opened. You may use & after >,>>,<, +>, >> and <. The mode you specify should match the mode of the original filehandle. Here is a script that saves, redirects, and restores STDOUT and STDERR:

#!/usr/bin/perl open(SAVEOUT, ">&STDOUT"); open(SAVEERR, ">&STDERR"); open(STDOUT, ">foo.out") || die "Can't redirect stdout"; open(STDERR, ">&STDOUT") || die "Can't dup stdout"; select(STDERR); $| = 1; # make unbuffered select(STDOUT); $| = 1; # make unbuffered print STDOUT "stdout 1\n"; # this works for print STDERR "stderr 1\n"; # subprocesses too close(STDOUT); close(STDERR); open(STDOUT, ">&SAVEOUT"); open(STDERR, ">&SAVEERR"); print STDOUT "stdout 2\n"; print STDERR "stderr 2\n";

If you specify <&=N", where N is a number, then Perl will do an equivalent of C's *fdopen()* of that file descriptor. For example:

open(FILEHANDLE, "<&=$fd")

If you open a pipe on the command "-", i.e. either "|-" or "-|", then there is an implicit fork done, and the return value of open is the pid of the child within the parent process, and 0 within the child process. (Use defined(**$pid**) to determine whether the open was successful.) The filehandle behaves normally for the parent, but i/o to that filehandle is piped from/to the STDOUT/STDIN of the child process. In the child process the filehandle isn't opened--i/o happens from/to the new STDOUT or STDIN. Typically this is used like the normal piped open when you want to exercise more control over just how the pipe command gets executed, such as when you are running setuid, and don't want to have to scan shell commands for metacharacters. The following pairs are more or less equivalent:

open(FOO, "|tr '[a-z]' '[A-Z]'"); open(FOO, "|-") || exec 'tr', '[a-z]', '[A-Z]'; open(FOO, "cat -n '$file'|"); open(FOO, "-|") || exec 'cat', '-n', $file;

Explicitly closing any piped filehandle causes the parent process to wait for the child to finish, and returns the status value in $?. Note: on any operation which may do a fork, unflushed buffers remain unflushed in both processes, which means you may need to set $| to avoid duplicate output.

The filename that is passed to open will have leading and trailing whitespace deleted. In order to open a file with arbitrary weird characters in it, it's necessary to protect any leading and trailing whitespace thusly:

$file =~ s#^(\s)#./$1#; open(FOO, "< $file\0");

opendir DIRHANDLE,EXPR

Opens a directory named EXPR for processing by readdir() , telldir() , seekdir() , rewinddir() and closedir() . Returns TRUE if successful. DIRHANDLEs have their own namespace separate from FILEHANDLEs.

ord EXPR

Returns the numeric ascii value of the first character of EXPR. If EXPR is omitted, uses **$_** .

pack TEMPLATE,LIST

Takes an array or list of values and packs it into a binary structure, returning the string containing the structure. The TEMPLATE is a sequence of characters that give the order and type of values, as follows:

- A An ascii string, will be space padded.
- a An ascii string, will be null padded.
- b A bit string (ascending bit order, like vec()).
- B A bit string (descending bit order).
- h A hex string (low nybble first).
- H A hex string (high nybble first).
- c A signed char value.
- C An unsigned char value.
- s A signed short value.
- S An unsigned short value.
- i A signed integer value.
- I An unsigned integer value.
- l A signed long value.
- L An unsigned long value.
- n A short in "network" order.
- N A long in "network" order.
- v A short in "VAX" (little-endian) order.
- V A long in "VAX" (little-endian) order.
- f A single-precision float in the native format.
- d A double-precision float in the native format.
- p A pointer to a null-terminated string.
- P A pointer to a structure (fixed-length string).
- u A uuencoded string.
- x A null byte.
- X Back up a byte.
- @ Null fill to absolute position.

Each letter may optionally be followed by a number which gives a repeat count. With all types except "a", "A", "b", "B", "h" and "H", and "P" the pack function will gobble up that many values from the LIST. A * for the repeat count means to use however many items are left. The "a" and "A" types gobble just one value, but pack it as a string of length count, padding with nulls or spaces as necessary. (When unpacking, "A" strips trailing spaces and nulls, but "a" does not.) Likewise, the "b" and "B" fields pack a string that many bits long. The "h" and "H" fields pack a string that many nybbles long. The "P" packs a pointer to a structure of the size indicated by the

length. Real numbers (floats and doubles) are in the native machine format only; due to the multiplicity of floating formats around, and the lack of a standard "network" representation, no facility for interchange has been made. This means that packed floating point data written on one machine may not be readable on another - even if both use IEEE floating point arithmetic (as the endian-ness of the memory representation is not part of the IEEE spec). Note that Perl uses doubles internally for all numeric calculation, and converting from double into float and thence back to double again will lose precision (i.e. unpack("f", pack("f", **$foo**)) will not in general equal **$foo**).

Examples:

$foo = pack("cccc",65,66,67,68); # foo eq "ABCD" $foo = pack("c4",65,66,67,68); # same thing $foo = pack("ccxxcc",65,66,67,68); # foo eq "AB\0\0CD" $foo = pack("s2",1,2); # "\1\0\2\0" on little-endian # "\0\1\0\2" on big-endian $foo = pack("a4","abcd","x","y","z"); # "abcd" $foo = pack("aaaa","abcd","x","y","z"); # "axyz" $foo = pack("a14","abcdefg"); # "abcdefg\0\0\0\0\0\0\0" $foo = pack("i9pl", gmtime); # a real struct tm (on my system anyway) sub bintodec { unpack("N", pack("B32", substr("0" x 32 . shift, -32))); }

The same template may generally also be used in the unpack function.

pipe READHANDLE,WRITEHANDLE

Opens a pair of connected pipes like the corresponding system call. Note that if you set up a loop of piped processes, deadlock can occur unless you are very careful. In addition, note that Perl's pipes use stdio buffering, so you may need to set $| to flush your WRITEHANDLE after each command, depending on the application.

pop ARRAY

Pops and returns the last value of the array, shortening the array by 1. Has a similar effect to

$tmp = $ARRAY[$#ARRAY--];

If there are no elements in the array, returns the undefined value.

pos SCALAR

Returns the offset of where the last m//g search left off for the variable in question. May be modified to change that offset.

print FILEHANDLE LIST

print LIST

print

Prints a string or a comma-separated list of strings. Returns non-zero if successful. FILEHANDLE may be a scalar variable name, in which case the variable contains the name of the filehandle, thus introducing one level of indirection. (NOTE: If FILEHANDLE is a variable and the next token is a term, it may be misinterpreted as an operator unless you interpose a + or put parens around the arguments.) If FILEHANDLE is omitted, prints by default to standard output (or to the last selected output channel--see select()). If LIST is also omitted, prints **$_** to STDOUT. To set the default output channel to something other than STDOUT use the select operation. Note that, because print takes a LIST, anything in the LIST is evaluated in a list context, and any subroutine that you call will have one or more of its expressions evaluated in a list context. Also be careful

not to follow the print keyword with a left parenthesis unless you want the corresponding right parenthesis to terminate the arguments to the print--interpose a + or put parens around all the arguments.

printf FILEHANDLE LIST

printf LIST

Equivalent to a "print FILEHANDLE sprintf(LIST)". The first argument of the list will be interpreted as the printf format.

push ARRAY,LIST

Treats ARRAY as a stack, and pushes the values of LIST onto the end of ARRAY. The length of ARRAY increases by the length of LIST. Has the same effect as

for $value (LIST) { $ARRAY[++$#ARRAY] = $value; }

but is more efficient. Returns the new number of elements in the array.

q/STRING/

qq/STRING/

qx/STRING/

qw/STRING/

Generalized quotes. See the *perlop* manpage .

quotemeta EXPR

Returns the value of EXPR with with all regular expression metacharacters backslashed. This is the internal function implementing the \Q escape in double-quoted strings.

rand EXPR

rand

Returns a random fractional number between 0 and the value of EXPR. (EXPR should be positive.) If EXPR is omitted, returns a value between 0 and 1. This function produces repeatable sequences unless srand() is invoked. See also srand() .

(Note: if your rand function consistently returns numbers that are too large or too small, then your version of Perl was probably compiled with the wrong number of RANDBITS. As a workaround, you can usually multiply EXPR by the correct power of 2 to get the range you want. This will make your script unportable, however. It's better to recompile if you can.)

read FILEHANDLE,SCALAR,LENGTH,OFFSET

read FILEHANDLE,SCALAR,LENGTH

Attempts to read LENGTH bytes of data into variable SCALAR from the specified FILEHANDLE. Returns the number of bytes actually read, or undef if there was an error. SCALAR will be grown or shrunk to the length actually read. An OFFSET may be specified to place the read data at some other place than the beginning of the string. This call is actually implemented in terms of stdio's fread call. To get a true read system call, see sysread() .

readdir DIRHANDLE

Returns the next directory entry for a directory opened by opendir() . If used in a list context,

returns all the rest of the entries in the directory. If there are no more entries, returns an undefi value in a scalar context or a null list in a list context.

readlink EXPR

Returns the value of a symbolic link, if symbolic links are implemented. If not, gives a fatal error. If there is some system error, returns the undefined value and sets $! (errno). If EXPR is omitted, uses **$_** .

recv SOCKET,SCALAR,LEN,FLAGS

Receives a message on a socket. Attempts to receive LENGTH bytes of data into variable SCALAR from the specified SOCKET filehandle. Actually does a C *recvfrom()* , so that it can returns the address of the sender. Returns the undefined value if there's an error. SCALAR will be grown or shrunk to the length actually read. Takes the same flags as the system call of the same name.

redo LABEL

redo

The redo command restarts the loop block without evaluating the conditional again. The **continue** block, if any, is not executed. If the LABEL is omitted, the command refers to the innermost enclosing loop. This command is normally used by programs that want to lie to themselves about what was just input:

a simpleminded Pascal comment stripper # (warning: assumes no { or } in strings) line: while (<STDIN>) { while (s|({.*}.*){.*}|$1 |) {} s|{.*}| |; if (s|{.*| |) { $front = $_; while (<STDIN>) { if (/}/) { # end of comment? s|^|$front{|; redo line; } } } print; }

ref EXPR

Returns a TRUE value if EXPR is a reference, FALSE otherwise. The value returned depends on the type of thing the reference is a reference to. Builtin types include:

REF SCALAR ARRAY HASH CODE GLOB

If the referenced object has been blessed into a package, then that package name is returned instead. You can think of ref() as a *typeof()* operator.

if (ref($r) eq "HASH") { print "r is a reference to an associative array.\n"; } if (!ref ($r) { print "r is not a reference at all.\n"; }

See also the *perlref* manpage .

rename OLDNAME,NEWNAME

Changes the name of a file. Returns 1 for success, 0 otherwise. Will not work across filesystem boundaries.

require EXPR

require

Demands some semantics specified by EXPR, or by **$_** if EXPR is not supplied. If EXPR is numeric, demands that the current version of Perl ($] or **$PERL_VERSION**) be equal or greater than EXPR.

Otherwise, demands that a library file be included if it hasn't already been included. The file is included via the do-FILE mechanism, which is essentially just a variety of eval() . Has semant similar to the following subroutine:

sub require { local($filename) = @_; return 1 if $INC{$filename}; local($realfilename,$result); ITER: { foreach $prefix (@INC) { $realfilename = "$prefix/$filename"; if (-f $realfilename) { $result = do $realfilename; last ITER; } } die "Can't find $filename in \@INC"; } die $@ if $@; die "$filename did not return true value" unless $result; $INC{$filename} = $realfilename; $result; }

Note that the file will not be included twice under the same specified name. The file must return TRUE as the last statement to indicate successful execution of any initialization code, so it's customary to end such a file with "1;" unless you're sure it'll return TRUE otherwise. But it's better just to put the " **1;** ", in case you add more statements.

If EXPR is a bare word, the require assumes a " *.pm* " extension for you, to make it easy to load standard modules. This form of loading of modules does not risk altering your namespace.

For a yet-more-powerful import facility, see the *use* and the *perlmod* manpage .

reset EXPR

reset

Generally used in a **continue** block at the end of a loop to clear variables and reset ?? searches so that they work again. The expression is interpreted as a list of single characters (hyphens allowed for ranges). All variables and arrays beginning with one of those letters are reset to their pristine state. If the expression is omitted, one-match searches (?pattern?) are reset to match again. Only resets variables or searches in the current package. Always returns 1. Examples:

reset 'X'; # reset all X variables reset 'a-z'; # reset lower case variables reset; # just reset ?? searches

Resetting "A-Z" is not recommended since you'll wipe out your ARGV and ENV arrays. Only resets package variables--lexical variables are unaffected, but they clean themselves up on scope exit anyway, so anymore you probably want to use them instead. See *my* .

return LIST

Returns from a subroutine or eval with the value specified. (Note that in the absence of a return a subroutine or eval will automatically return the value of the last expression evaluated.)

reverse LIST

In a list context, returns a list value consisting of the elements of LIST in the opposite order. In a scalar context, returns a string value consisting of the bytes of the first element of LIST in the opposite order.

rewinddir DIRHANDLE

Sets the current position to the beginning of the directory for the readdir() routine on DIRHANDLE.

rindex STR,SUBSTR,POSITION

rindex STR,SUBSTR

Works just like index except that it returns the position of the LAST occurrence of SUBSTR in STR. If POSITION is specified, returns the last occurrence at or before that position.

rmdir FILENAME

Deletes the directory specified by FILENAME if it is empty. If it succeeds it returns 1, otherwise it returns 0 and sets $! (errno). If FILENAME is omitted, uses **$_** .

s///

The substitution operator. See the *perlop* manpage .

scalar EXPR

Forces EXPR to be interpreted in a scalar context and returns the value of EXPR.

seek FILEHANDLE,POSITION,WHENCE

Randomly positions the file pointer for FILEHANDLE, just like the *fseek()* call of stdio. FILEHANDLE may be an expression whose value gives the name of the filehandle. The values for WHENCE are 0 to set the file pointer to POSITION, 1 to set the it to current plus POSITION, and 2 to set it to EOF plus offset. You may use the values SEEK_SET, SEEK_CUR, and SEEK_END for this is using the POSIX module. Returns 1 upon success, 0 otherwise.

seekdir DIRHANDLE,POS

Sets the current position for the readdir() routine on DIRHANDLE. POS must be a value returned by telldir() . Has the same caveats about possible directory compaction as the corresponding system library routine.

select FILEHANDLE

select

Returns the currently selected filehandle. Sets the current default filehandle for output, if FILEHANDLE is supplied. This has two effects: first, a write or a print without a filehandle will default to this FILEHANDLE. Second, references to variables related to output will refer to this output channel. For example, if you have to set the top of form format for more than one output channel, you might do the following:

select(REPORT1); $^ = 'report1_top'; select(REPORT2); $^ = 'report2_top';

FILEHANDLE may be an expression whose value gives the name of the actual filehandle. Thus:

$oldfh = select(STDERR); $| = 1; select($oldfh);

With Perl 5, filehandles are objects with methods, and the last example is preferably written

use FileHandle; STDERR->autoflush(1);

select RBITS,WBITS,EBITS,TIMEOUT

This calls the select system(2) call with the bitmasks specified, which can be constructed using fileno() and vec() , along these lines:

$rin = $win = $ein = ''; vec($rin,fileno(STDIN),1) = 1; vec($win,fileno(STDOUT),1) = 1; $ein = $rin | $win;

If you want to select on many filehandles you might wish to write a subroutine:

sub fhbits { local(@fhlist) = split(' ',$_[0]); local($bits); for (@fhlist) { vec($bits,fileno($_),1) } $bits; } $rin = &fhbits('STDIN TTY SOCK');

The usual idiom is:

($nfound,$timeleft) = select($rout=$rin, $wout=$win, $eout=$ein, $timeout);

or to block until something becomes ready:

$nfound = select($rout=$rin, $wout=$win, $eout=$ein, undef);

Any of the bitmasks can also be undef. The timeout, if specified, is in seconds, which may be fractional. Note: not all implementations are capable of returning the **$timeleft** . If not, they always return **$timeleft** equal to the supplied **$timeout** .

You can effect a 250 microsecond sleep this way:

select(undef, undef, undef, 0.25);

semctl ID,SEMNUM,CMD,ARG

Calls the System V IPC function semctl. If CMD is &IPC_STAT or &GETALL, then ARG must be a variable which will hold the returned semid_ds structure or semaphore value array. Returns like ioctl: the undefined value for error, "0 but true" for zero, or the actual return value otherwise.

semget KEY,NSEMS,FLAGS

Calls the System V IPC function semget. Returns the semaphore id, or the undefined value if there is an error.

semop KEY,OPSTRING

Calls the System V IPC function semop to perform semaphore operations such as signaling and waiting. OPSTRING must be a packed array of semop structures. Each semop structure can be generated with pack("sss", **$semnum** , **$semop** , **$semflag**) . The number of semaphore operations is implied by the length of OPSTRING. Returns TRUE if successful, or FALSE if there is an error. As an example, the following code waits on semaphore **$semnum** of semaphore id **$semid:**

$semop = pack("sss", $semnum, -1, 0); die "Semaphore trouble: $!\n" unless semop($semid, $semop);

To signal the semaphore, replace "-1" with "1".

send SOCKET,MSG,FLAGS,TO

send SOCKET,MSG,FLAGS

Sends a message on a socket. Takes the same flags as the system call of the same name. On unconnected sockets you must specify a destination to send TO, in which case it does a C *sendto()* . Returns the number of characters sent, or the undefined value if there is an error.

setpgrp PID,PGRP

Sets the current process group for the specified PID, 0 for the current process. Will produce a fatal error if used on a machine that doesn't implement setpgrp(2).

setpriority WHICH,WHO,PRIORITY

Sets the current priority for a process, a process group, or a user. (See setpriority(2).) Will produce a fatal error if used on a machine that doesn't implement setpriority(2).

setsockopt SOCKET,LEVEL,OPTNAME,OPTVAL

Sets the socket option requested. Returns undefined if there is an error. OPTVAL may be specified as undef if you don't want to pass an argument.

shift ARRAY

shift

Shifts the first value of the array off and returns it, shortening the array by 1 and moving everything down. If there are no elements in the array, returns the undefined value. If ARRAY is omitted, shifts the **@ARGV** array in the main program, and the **@_** array in subroutines. (This is determined lexically.) See also unshift() , push() , and pop() . *Shift()* and unshift() do the same thing to the left end of an array that push() and pop() do to the right end.

shmctl ID,CMD,ARG

Calls the System V IPC function shmctl. If CMD is &IPC_STAT, then ARG must be a variable which will hold the returned shmid_ds structure. Returns like ioctl: the undefined value for error, "0 but true" for zero, or the actual return value otherwise.

shmget KEY,SIZE,FLAGS

Calls the System V IPC function shmget. Returns the shared memory segment id, or the undefined value if there is an error.

shmread ID,VAR,POS,SIZE

shmwrite ID,STRING,POS,SIZE

Reads or writes the System V shared memory segment ID starting at position POS for size SIZE by attaching to it, copying in/out, and detaching from it. When reading, VAR must be a variable which will hold the data read. When writing, if STRING is too long, only SIZE bytes are used; if STRING is too short, nulls are written to fill out SIZE bytes. Return TRUE if successful, or FALSE if there is an error.

shutdown SOCKET,HOW

Shuts down a socket connection in the manner indicated by HOW, which has the same interpretation as in the system call of the same name.

sin EXPR

Returns the sine of EXPR (expressed in radians). If EXPR is omitted, returns sine of **$_** .

sleep EXPR

sleep

Causes the script to sleep for EXPR seconds, or forever if no EXPR. May be interrupted by sending the process a SIGALRM. Returns the number of seconds actually slept. You probably cannot mix alarm() and sleep() calls, since sleep() is often implemented using alarm() .

On some older systems, it may sleep up to a full second less than what you requested, depending on how it counts seconds. Most modern systems always sleep the full amount.

socket SOCKET,DOMAIN,TYPE,PROTOCOL

Opens a socket of the specified kind and attaches it to filehandle SOCKET. DOMAIN, TYPE and PROTOCOL are specified the same as for the system call of the same name. You should "use Socket;" first to get the proper definitions imported. See the example in the *perlipc* manpage .

socketpair SOCKET1,SOCKET2,DOMAIN,TYPE,PROTOCOL

Creates an unnamed pair of sockets in the specified domain, of the specified type. DOMAIN, TYPE and PROTOCOL are specified the same as for the system call of the same name. If unimplemented, yields a fatal error. Returns TRUE if successful.

sort SUBNAME LIST

sort BLOCK LIST

sort LIST

Sorts the LIST and returns the sorted list value. Nonexistent values of arrays are stripped out. If SUBNAME or BLOCK is omitted, sorts in standard string comparison order. If SUBNAME is specified, it gives the name of a subroutine that returns an integer less than, equal to, or greater than 0, depending on how the elements of the array are to be ordered. (The<=> and cmp operators are extremely useful in such routines.) SUBNAME may be a scalar variable name, in which case the value provides the name of the subroutine to use. In place of a SUBNAME, you can provide a BLOCK as an anonymous, in-line sort subroutine.

In the interests of efficiency the normal calling code for subroutines is bypassed, with the following effects: the subroutine may not be a recursive subroutine, and the two elements to be compared are passed into the subroutine not via **@_** but as **$a** and **$b** (see example below). They are passed by reference, so don't modify **$a** and **$b** .

Examples:

sort lexically @articles = sort @files; # same thing, but with explicit sort routine @articles = sort {$a cmp $b} @files; # same thing in reversed order @articles = sort {$b cmp $a} @files; # sort numerically ascending @articles = sort {$a <=> $b} @files; # sort numerically descending @articles = sort {$b <=> $a} @files; # sort using explicit subroutine name sub byage { $age{$a} <=> $age{$b}; # presuming integers } @sortedclass = sort byage @class; sub backwards { $b cmp $a; } @harry = ('dog','cat','x','Cain','Abel'); @george = ('gone','chased','yz','Punished','Axed'); print sort @harry; # prints AbelCaincatdogx print sort backwards @harry; # prints xdogcatCainAbel print sort @george, 'to', @harry; # prints AbelAxedCainPunishedcatchaseddoggonetoxyz

splice ARRAY,OFFSET,LENGTH,LIST

splice ARRAY,OFFSET,LENGTH

splice ARRAY,OFFSET

Removes the elements designated by OFFSET and LENGTH from an array, and replaces them with the elements of LIST, if any. Returns the elements removed from the array. The array grows or shrinks as necessary. If LENGTH is omitted, removes everything from OFFSET onward. The following equivalencies hold (assuming $[== 0):

push(@a,$x,$y) splice(@a,$#a+1,0,$x,$y) pop(@a) splice(@a,-1) shift(@a) splice(@a,0,1) unshift(@a,$x,$y) splice(@a,0,0,$x,$y) $a[$x] = $y splice(@a,$x,1,$y);

Example, assuming array lengths are passed before arrays:

sub aeq { # compare two list values local(@a) = splice(@_,0,shift); local(@b) = splice(@_,0,shift); return 0 unless @a == @b; # same len? while (@a) { return 0 if pop(@a) ne pop(@b); } return 1; } if (&aeq($len,@foo[1..$len],0+@bar,@bar)) { ... }

split /PATTERN/,EXPR,LIMIT

split /PATTERN/,EXPR

split /PATTERN/

split

Splits a string into an array of strings, and returns it.

If not in a list context, returns the number of fields found and splits into the **@_** array. (In a list context, you can force the split into **@_** by using **??** as the pattern delimiters, but it still returns the array value.) The use of implicit split to **@_** is deprecated, however.

If EXPR is omitted, splits the **$_** string. If PATTERN is also omitted, splits on whitespace (after skipping any leading whitespace). Anything matching PATTERN is taken to be a delimiter separating the fields. (Note that the delimiter may be longer than one character.) If LIMIT is specified and is not negative, splits into no more than that many fields (though it may split into fewer). If LIMIT is unspecified, trailing null fields are stripped (which potential users of pop() would do well to remember). If LIMIT is negative, it is treated as if an arbitrarily large LIMIT had been specified.

A pattern matching the null string (not to be confused with a null pattern // , which is just one member of the set of patterns matching a null string) will split the value of EXPR into separate characters at each point it matches that way. For example:

print join(':', split(/ */, 'hi there'));

produces the output 'h:i:t:h:e:r:e'.

The LIMIT parameter can be used to partially split a line

($login, $passwd, $remainder) = split(/:/, $_, 3);

When assigning to a list, if LIMIT is omitted, Perl supplies a LIMIT one larger than the number of variables in the list, to avoid unnecessary work. For the list above LIMIT would have been 4 by default. In time critical applications it behooves you not to split into more fields than you really need.

If the PATTERN contains parentheses, additional array elements are created from each matching substring in the delimiter.

split(/([,-])/, "1-10,20");

produces the list value

(1, '-', 10, ',', 20)

The pattern **/PATTERN/** may be replaced with an expression to specify patterns that vary at runtime. (To do runtime compilation only once, use **/ $variable /o** .)

As a special case, specifying a PATTERN of space (' ') will split on white space just as split with no arguments does. Thus, split(' ') can be used to emulate **awk** 's default behavior, whereas split(/ /) will give you as many null initial fields as there are leading spaces. A split on /\s+/ is like a split(' ') except that any leading whitespace produces a null first field. A split with no arguments really does a split(' ', $_) internally.

Example:

open(passwd, '/etc/passwd'); while (<passwd>) { ($login, $passwd, $uid, $gid, $gcos, $home, $shell) = split(/:/); ... }

(Note that **$shell** above will still have a newline on it. See *chop* , *chomp* , and *join* .)

sprintf FORMAT,LIST

Returns a string formatted by the usual printf conventions of the C language. (The * character for an indirectly specified length is not supported, but you can get the same effect by interpolating a variable into the pattern.)

sqrt EXPR

Return the square root of EXPR. If EXPR is omitted, returns square root of **$_** .

srand EXPR

Sets the random number seed for the rand operator. If EXPR is omitted, does srand(time) . Of course, you'd need something much more random than that for cryptographic purposes, since it's easy to guess the current time. Checksumming the compressed output of rapidly changing operating system status programs is the usual method. Examples are posted regularly to comp.security.unix.

stat FILEHANDLE

stat EXPR

Returns a 13-element array giving the status info for a file, either the file opened via FILEHANDLE, or named by EXPR. Returns a null list if the stat fails. Typically used as follows:

($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size, $atime,$mtime,$ctime,$blksize,$blocks) = stat($filename);

If stat is passed the special filehandle consisting of an underline, no stat is done, but the current contents of the stat structure from the last stat or filetest are returned. Example:

if (-x $file && (($d) = stat(_)) && $d < 0) { print "$file is executable NFS file\n"; }

(This only works on machines for which the device number is negative under NFS.)

study SCALAR

study

Takes extra time to study SCALAR (**$_** if unspecified) in anticipation of doing many pattern matches on the string before it is next modified. This may or may not save time, depending on the

nature and number of patterns you are searching on, and on the distribution of character frequencies in the string to be searched--you probably want to compare runtimes with and without it to see which runs faster. Those loops which scan for many short constant strings (including the constant parts of more complex patterns) will benefit most. You may have only one study active at a time--if you study a different scalar the first is "unstudied". (The way study works is this: a linked list of every character in the string to be searched is made, so we know, for example, where all the 'k' characters are. From each search string, the rarest character is selected, based on some static frequency tables constructed from some C programs and English text. Only those places that contain this "rarest" character are examined.)

For example, here is a loop which inserts index producing entries before any line containing a certain pattern:

while (<>) { study; print ".IX foo\n" if /\bfoo\b/; print ".IX bar\n" if /\bbar\b/; print ".IX blurfl\n" if /\bblurfl\b/; ... print; }

In searching for /\bfoo\b/, only those locations in **$_** that contain "f" will be looked at, because "f" is rarer than "o". In general, this is a big win except in pathological cases. The only question is whether it saves you more time than it took to build the linked list in the first place.

Note that if you have to look for strings that you don't know till runtime, you can build an entire loop as a string and eval that to avoid recompiling all your patterns all the time. Together with undefining $/ to input entire files as one record, this can be very fast, often faster than specialized programs like fgrep(1). The following scans a list of files (**@files**) for a list of words (**@words**), and prints out the names of those files that contain a match:

$search = 'while (<>) { study;'; foreach $word (@words) { $search .= "++\$seen{\$ARGV} if /\\b$word\\b/;\n"; } $search .= "}"; @ARGV = @files; undef $/; eval $search; # this screams $/ = "\n"; # put back to normal input delim foreach $file (sort keys(%seen)) { print $file, "\n"; }

substr EXPR,OFFSET,LEN

substr EXPR,OFFSET

Extracts a substring out of EXPR and returns it. First character is at offset 0, or whatever you've set $[to. If OFFSET is negative, starts that far from the end of the string. If LEN is omitted, returns everything to the end of the string. If LEN is negative, leaves that many characters off the end of the string.

You can use the substr() function as an lvalue, in which case EXPR must be an lvalue. If you assign something shorter than LEN, the string will shrink, and if you assign something longer than LEN, the string will grow to accommodate it. To keep the string the same length you may need to pad or chop your value using sprintf() .

symlink OLDFILE,NEWFILE

Creates a new filename symbolically linked to the old filename. Returns 1 for success, 0 otherwise. On systems that don't support symbolic links, produces a fatal error at run time. To check for that, use eval:

$symlink_exists = (eval 'symlink("","");', $@ eq '');

syscall LIST

Calls the system call specified as the first element of the list, passing the remaining elements as arguments to the system call. If unimplemented, produces a fatal error. The arguments are interpreted as follows: if a given argument is numeric, the argument is passed as an int. If not, the pointer to the string value is passed. You are responsible to make sure a string is pre-extended long enough to receive any result that might be written into a string. If your integer arguments are not literals and have never been interpreted in a numeric context, you may need to add 0 to them to force them to look like numbers.

require 'syscall.ph'; # may need to run h2ph syscall(&SYS_write, fileno(STDOUT), "hi there\n", 9);

Note that Perl only supports passing of up to 14 arguments to your system call, which in practice should usually suffice.

sysread FILEHANDLE,SCALAR,LENGTH,OFFSET

sysread FILEHANDLE,SCALAR,LENGTH

Attempts to read LENGTH bytes of data into variable SCALAR from the specified FILEHANDLE, using the system call read(2). It bypasses stdio, so mixing this with other kinds of reads may cause confusion. Returns the number of bytes actually read, or undef if there was an error. SCALAR will be grown or shrunk to the length actually read. An OFFSET may be specified to place the read data at some other place than the beginning of the string.

system LIST

Does exactly the same thing as "exec LIST" except that a fork is done first, and the parent process waits for the child process to complete. Note that argument processing varies depending on the number of arguments. The return value is the exit status of the program as returned by the wait() call. To get the actual exit value divide by 256. See also *exec* .

syswrite FILEHANDLE,SCALAR,LENGTH,OFFSET

syswrite FILEHANDLE,SCALAR,LENGTH

Attempts to write LENGTH bytes of data from variable SCALAR to the specified FILEHANDLE, using the system call write(2). It bypasses stdio, so mixing this with prints may cause confusion. Returns the number of bytes actually written, or undef if there was an error. An OFFSET may be specified to place the read data at some other place than the beginning of the string.

tell FILEHANDLE

tell

Returns the current file position for FILEHANDLE. FILEHANDLE may be an expression whose value gives the name of the actual filehandle. If FILEHANDLE is omitted, assumes the file last read.

telldir DIRHANDLE

Returns the current position of the readdir() routines on DIRHANDLE. Value may be given to seekdir() to access a particular location in a directory. Has the same caveats about possible directory compaction as the corresponding system library routine.

tie VARIABLE,PACKAGENAME,LIST

This function binds a variable to a package that will provide the implementation for the variab VARIABLE is the name of the variable to be enchanted. PACKAGENAME is the name of a package implementing objects of correct type. Any additional arguments are passed to the "new" method of the package (meaning TIESCALAR, TIEARRAY, or TIEHASH). Typically these are arguments such as might be passed to the *dbm_open()* function of C.

Note that functions such as keys() and values() may return huge array values when used on large objects, like DBM files. You may prefer to use the each() function to iterate over such. Example:

print out history file offsets tie(%HIST, NDBM_File, '/usr/lib/news/history', 1, 0); while (($key,$val) = each %HIST) { print $key, ' = ', unpack('L',$val), "\n"; } untie(%HIST);

A package implementing an associative array should have the following methods:

TIEHASH objectname, LIST DESTROY this FETCH this, key STORE this, key, value DELETE this, key EXISTS this, key FIRSTKEY this NEXTKEY this, lastkey

A package implementing an ordinary array should have the following methods:

TIEARRAY objectname, LIST DESTROY this FETCH this, key STORE this, key, value [others TBD]

A package implementing a scalar should have the following methods:

TIESCALAR objectname, LIST DESTROY this FETCH this, STORE this, value

time

Returns the number of non-leap seconds since 00:00:00 UTC, January 1, 1970. Suitable for feeding to gmtime() and localtime() .

times

Returns a four-element array giving the user and system times, in seconds, for this process and the children of this process.

($user,$system,$cuser,$csystem) = times;

tr///

The translation operator. See the *perlop* manpage .

truncate FILEHANDLE,LENGTH

truncate EXPR,LENGTH

Truncates the file opened on FILEHANDLE, or named by EXPR, to the specified length. Produces a fatal error if truncate isn't implemented on your system.

uc EXPR

Returns an uppercased version of EXPR. This is the internal function implementing the \U escape in double-quoted strings.

ucfirst EXPR

Returns the value of EXPR with the first character uppercased. This is the internal function implementing the \u escape in double-quoted strings.

umask EXPR

umask

Sets the umask for the process and returns the old one. If EXPR is omitted, merely returns current umask.

undef EXPR

undef

Undefines the value of EXPR, which must be an lvalue. Use only on a scalar value, an entire array, or a subroutine name (using "&"). (Using undef() will probably not do what you expect on most predefined variables or DBM list values, so don't do that.) Always returns the undefined value. You can omit the EXPR, in which case nothing is undefined, but you still get an undefined value that you could, for instance, return from a subroutine. Examples:

undef $foo; undef $bar{'blurfl'}; undef @ary; undef %assoc; undef &mysub; return (wantarray ? () : undef) if $they_blew_it;

unlink LIST

Deletes a list of files. Returns the number of files successfully deleted.

$cnt = unlink 'a', 'b', 'c'; unlink @goners; unlink <*.bak>;

Note: unlink will not delete directories unless you are superuser and the **-U** flag is supplied to Perl. Even if these conditions are met, be warned that unlinking a directory can inflict damage on your filesystem. Use rmdir instead.

unpack TEMPLATE,EXPR

Unpack does the reverse of pack: it takes a string representing a structure and expands it out into a list value, returning the array value. (In a scalar context, it merely returns the first value produced.) The TEMPLATE has the same format as in the pack function. Here's a subroutine that does substring:

sub substr { local($what,$where,$howmuch) = @_; unpack("x$where a$howmuch", $what); }

and then there's

sub ordinal { unpack("c",$_[0]); } # same as ord()

In addition, you may prefix a field with a <number> to indicate that you want a<number>-bit checksum of the items instead of the items themselves. Default is a 16-bit checksum. For example, the following computes the same number as the System V sum program:

while (<>) { $checksum += unpack("%16C*", $_); } $checksum %= 65536;

The following efficiently counts the number of set bits in a bit vector:

$setbits = unpack("%32b*", $selectmask);

untie VARIABLE

Breaks the binding between a variable and a package. (See tie() .)

unshift ARRAY,LIST

Does the opposite of a shift . Or the opposite of a push , depending on how you look at it. Prepends list to the front of the array, and returns the new number of elements in the array.

unshift(ARGV, '-e') unless $ARGV[0] =~ /^-/;

Note the LIST is prepended whole, not one element at a time, so the prepended elements stay in the same order. Use reverse to do the reverse.

use Module LIST

use Module

Imports some semantics into the current package from the named module, generally by aliasing certain subroutine or variable names into your package. It is exactly equivalent to

BEGIN { require Module; import Module LIST; }

If you don't want your namespace altered, use require instead.

The BEGIN forces the require and import to happen at compile time. The require makes sure the module is loaded into memory if it hasn't been yet. The import is not a builtin--it's just an ordinary static method call into the "Module" package to tell the module to import the list of features back into the current package. The module can implement its import method any way it likes, though most modules just choose to derive their import method via inheritance from the Exporter class that is defined in the Exporter module.

Because this is a wide-open interface, pragmas (compiler directives) are also implemented this way. Currently implemented pragmas are:

use integer; use sigtrap qw(SEGV BUS); use strict qw(subs vars refs); use subs qw(afunc blurfl);

These pseudomodules import semantics into the current block scope, unlike ordinary modules, which import symbols into the current package (which are effective through the end of the file).

There's a corresponding "no" command that unimports meanings imported by use.

no integer; no strict 'refs';

See the *perlmod* manpage for a list of standard modules and pragmas.

utime LIST

Changes the access and modification times on each file of a list of files. The first two elements of the list must be the NUMERICAL access and modification times, in that order. Returns the number of files successfully changed. The inode modification time of each file is set to the current time. Example of a "touch" command:

#!/usr/bin/perl $now = time; utime $now, $now, @ARGV;

values ASSOC_ARRAY

Returns a normal array consisting of all the values of the named associative array. (In a scalar context, returns the number of values.) The values are returned in an apparently random order, but it is the same order as either the keys() or each() function would produce on the same array. See also keys() and each() .

vec EXPR,OFFSET,BITS

Treats a string as a vector of unsigned integers, and returns the value of the bitfield specified. May also be assigned to. BITS must be a power of two from 1 to 32.

Vectors created with vec() can also be manipulated with the logical operators |, & and ^, which will assume a bit vector operation is desired when both operands are strings.

To transform a bit vector into a string or array of 0's and 1's, use these:

$bits = unpack("b*", $vector); @bits = split(//, unpack("b*", $vector));

If you know the exact length in bits, it can be used in place of the *.

wait

Waits for a child process to terminate and returns the pid of the deceased process, or -1 if there are no child processes. The status is returned in $?.

waitpid PID,FLAGS

Waits for a particular child process to terminate and returns the pid of the deceased process, or -1 if there is no such child process. The status is returned in $?. If you say

use POSIX "wait_h"; ... waitpid(-1,&WNOHANG);

then you can do a non-blocking wait for any process. Non-blocking wait is only available on machines supporting either the waitpid(2) or wait4(2) system calls. However, waiting for a particular pid with FLAGS of 0 is implemented everywhere. (Perl emulates the system call by remembering the status values of processes that have exited but have not been harvested by the Perl script yet.)

wantarray

Returns TRUE if the context of the currently executing subroutine is looking for a list value. Returns FALSE if the context is looking for a scalar.

return wantarray ? () : undef;

warn LIST

Produces a message on STDERR just like die() , but doesn't exit or throw an exception.

write FILEHANDLE

write EXPR

write

Writes a formatted record (possibly multi-line) to the specified file, using the format associated with that file. By default the format for a file is the one having the same name is the filehandle, but the format for the current output channel (see the select() function) may be set explicitly by assigning the name of the format to the $~ variable.

Top of form processing is handled automatically: if there is insufficient room on the current page for the formatted record, the page is advanced by writing a form feed, a special top-of-page format is used to format the new page header, and then the record is written. By default the top-of-page format is the name of the filehandle with "_TOP" appended, but it may be dynamically set to the

format of your choice by assigning the name to the $^ variable while the filehandle is selected. The number of lines remaining on the current page is in variable $-, which can be set to 0 to force a new page.

If FILEHANDLE is unspecified, output goes to the current default output channel, which starts out as STDOUT but may be changed by the select operator. If the FILEHANDLE is an EXPR, then the expression is evaluated and the resulting string is used to look up the name of the FILEHANDLE at run time. For more on formats, see the *perlform* manpage .

Note that write is *NOT* the opposite of read. Unfortunately.

y///

The translation operator. See *tr///* .

.

NAME

perlvar - Perl predefined variables

DESCRIPTION

Predefined Names

The following names have special meaning to Perl. Most of the punctuational names have reasonable mnemonics, or analogues in one of the shells. Nevertheless, if you wish to use the long variable names, you just need to say

use English;

at the top of your program. This will alias all the short names to the long names in the current package. Some of them even have medium names, generally borrowed from **awk** .

To go a step further, those variables that depend on the currently selected filehandle may instead be set by calling an object method on the FileHandle object. (Summary lines below for this contain the word HANDLE.) First you must say

use FileHandle;

after which you may use either

method HANDLE EXPR

or

HANDLE->method(EXPR)

Each of the methods returns the old value of the FileHandle attribute. The methods each take an optional EXPR, which if supplied specifies the new value for the FileHandle attribute in question. If not supplied, most of the methods do nothing to the current value, except for autoflush() , which will assume a 1 for you, just to be different.

A few of these variables are considered "read-only". This means that if you try to assign to this variable, either directly or indirectly through a reference, you'll raise a run-time exception.

$ARG

$_

The default input and pattern-searching space. The following pairs are equivalent:

while (<>) {...} # only equivalent in while! while ($_ = <>) {...} /^Subject:/ $_ =~ /^Subject:/

tr/a-z/A-Z/ $_ =~ tr/a-z/A-Z/ chop chop($_)

(Mnemonic: underline is understood in certain operations.)

< *digit* >

Contains the subpattern from the corresponding set of parentheses in the last pattern matched, not counting patterns matched in nested blocks that have been exited already. (Mnemonic: like \digit.) These variables are all read-only.

$MATCH

$&

The string matched by the last successful pattern match (not counting any matches hidden within a BLOCK or *eval()* enclosed by the current BLOCK). (Mnemonic: like & in some editors.) This variable is read-only.

$PREMATCH

$`

The string preceding whatever was matched by the last successful pattern match (not counting any matches hidden within a BLOCK or eval enclosed by the current BLOCK). (Mnemonic: ` often precedes a quoted string.) This variable is read-only.

$POSTMATCH

$'

The string following whatever was matched by the last successful pattern match (not counting any matches hidden within a BLOCK or *eval()* enclosed by the current BLOCK). (Mnemonic: ' often follows a quoted string.) Example:

$_ = 'abcdefghi'; /def/; print "$`:$&:$'\n"; # prints abc:def:ghi

This variable is read-only.

$LAST_PAREN_MATCH

$+

The last bracket matched by the last search pattern. This is useful if you don't know which of a set of alternative patterns matched. For example:

/Version: (.*)|Revision: (.*)/ && ($rev = $+);

(Mnemonic: be positive and forward looking.) This variable is read-only.

$MULTILINE_MATCHING

$*

Set to 1 to do multiline matching within a string, 0 to tell Perl that it can assume that strings contain a single line, for the purpose of optimizing pattern matches. Pattern matches on strings containing multiple newlines can produce confusing results when " $* " is 0. Default is 0. (Mnemonic: * matches multiple things.) Note that this variable only influences the interpretation of " ^ " and " **$** ". A literal newline can be searched for even when **$* == 0** .

Use of " $* " is deprecated in Perl 5.

input_line_number HANDLE EXPR

$INPUT_LINE_NUMBER

$NR

$.

The current input line number of the last filehandle that was read. This variable should be considered read-only. Remember that only an explicit close on the filehandle resets the line number. Since " <> " never does an explicit close, line numbers increase across ARGV files (but see examples under *eof()*). (Mnemonic: many programs use "." to mean the current line number.)

input_record_separator HANDLE EXPR

$INPUT_RECORD_SEPARATOR

$RS

$/

The input record separator, newline by default. Works like **awk** 's RS variable, including treating blank lines as delimiters if set to the null string. You may set it to a multicharacter string to match a multi-character delimiter. Note that setting it to **"\n\n"** means something slightly different than setting it to "" , if the file contains consecutive blank lines. Setting it to "" will treat two or more consecutive blank lines as a single blank line. Setting it to **"\n\n"** will blindly assume that the next input character belongs to the next paragraph, even if it's a newline. (Mnemonic: / is used to delimit line boundaries when quoting poetry.)

undef $/; $_ = <FH>; # whole file now here s/\n[\t]+/ /g;

autoflush HANDLE EXPR

$OUTPUT_AUTOFLUSH

$|

If set to nonzero, forces a flush after every write or print on the currently selected output channel. Default is 0. Note that STDOUT will typically be line buffered if output is to the terminal and block buffered otherwise. Setting this variable is useful primarily when you are outputting to a pipe, such as when you are running a Perl script under rsh and want to see the output as it's happening. (Mnemonic: when you want your pipes to be piping hot.)

output_field_separator HANDLE EXPR

$OUTPUT_FIELD_SEPARATOR

$OFS

$,

The output field separator for the print operator. Ordinarily the print operator simply prints out the comma separated fields you specify. In order to get behavior more like **awk** , set this variable as you would set **awk** 's OFS variable to specify what is printed between fields. (Mnemonic: what is printed when there is a , in your print statement.)

output_record_separator HANDLE EXPR

$OUTPUT_RECORD_SEPARATOR

$ORS

$

The output record separator for the print operator. Ordinarily the print operator simply prints out the comma separated fields you specify, with no trailing newline or record separator assumed. In order to get behavior more like **awk** , set this variable as you would set **awk** 's ORS variable to specify what is printed at the end of the print. (Mnemonic: you set " $\ " instead of adding \n at the end of the print. Also, it's just like /, but it's what you get "back" from Perl.)

$LIST_SEPARATOR

$"

This is like " $, " except that it applies to array values interpolated into a double-quoted string (or similar interpreted string). Default is a space. (Mnemonic: obvious, I think.)

$SUBSCRIPT_SEPARATOR

$SUBSEP

$;

The subscript separator for multi-dimensional array emulation. If you refer to a hash element as

$foo{$a,$b,$c}

it really means

$foo{join($;, $a, $b, $c)}

But don't put

@foo{$a,$b,$c} # a slice--note the @

which means

($foo{$a},$foo{$b},$foo{$c})

Default is "\034", the same as SUBSEP in **awk** . Note that if your keys contain binary data there might not be any safe value for " $; ". (Mnemonic: comma (the syntactic subscript separator) is a semi-semicolon. Yeah, I know, it's pretty lame, but " $, " is already taken for something more important.)

Consider using "real" multi-dimensional arrays in Perl 5.

$OFMT

$#

The output format for printed numbers. This variable is a half-hearted attempt to emulate **awk** 's OFMT variable. There are times, however, when **awk** and Perl have differing notions of what is in fact numeric. Also, the initial value is **%.** 20g rather than **%.** 6g, so you need to set " $# " explicitly to get **awk** 's value. (Mnemonic: # is the number sign.)

Use of " $# " is deprecated in Perl 5.

format_page_number HANDLE EXPR

$FORMAT_PAGE_NUMBER

$%

The current page number of the currently selected output channel. (Mnemonic: **%** is page number in **nroff** .)

format_lines_per_page HANDLE EXPR

$FORMAT_LINES_PER_PAGE

$=

The current page length (printable lines) of the currently selected output channel. Default is 60. (Mnemonic: = has horizontal lines.)

format_lines_left HANDLE EXPR

$FORMAT_LINES_LEFT

$-

The number of lines left on the page of the currently selected output channel. (Mnemonic: lines_on_page - lines_printed.)

format_name HANDLE EXPR

$FORMAT_NAME

$~

The name of the current report format for the currently selected output channel. Default is name of the filehandle. (Mnemonic: brother to " $^ ".)

format_top_name HANDLE EXPR

$FORMAT_TOP_NAME

$^

The name of the current top-of-page format for the currently selected output channel. Default is name of the filehandle with _TOP appended. (Mnemonic: points to top of page.)

format_line_break_characters HANDLE EXPR

$FORMAT_LINE_BREAK_CHARACTERS

$:

The current set of characters after which a string may be broken to fill continuation fields (starting with ^) in a format. Default is S<" \n-">, to break on whitespace or hyphens. (Mnemonic: a "colon" in poetry is a part of a line.)

format_formfeed HANDLE EXPR

$FORMAT_FORMFEED

$^ L

What formats output to perform a formfeed. Default is \f.

$ACCUMULATOR

$^ A

The current value of the *write()* accumulator for *format()* lines. A format contains *formline()* commands that put their result into $^ A . After calling its format, *write()* prints out the contents of $^ A and empties. So you never actually see the contents of $^ A unless you call *formline()*

yourself and then look at it. See the *perlform* manpage and *formline* .

$CHILD_ERROR

$?

The status returned by the last pipe close, backtick (``) command, or *system()* operator. Note that this is the status word returned by the *wait()* system call, so the exit value of the subprocess is actually (**$? >> 8**). Thus on many systems, **$? & 255** gives which signal, if any, the process died from, and whether there was a core dump. (Mnemonic: similar to **sh** and **ksh** .)

$OS_ERROR

$ERRNO

$!

If used in a numeric context, yields the current value of errno, with all the usual caveats. (This means that you shouldn't depend on the value of " $! " to be anything in particular unless you've gotten a specific error return indicating a system error.) If used in a string context, yields the corresponding system error string. You can assign to " $! " in order to set *errno* if, for instance, you want " $! " to return the string for error *n* , or you want to set the exit value for the *die()* operator. (Mnemonic: What just went bang?)

$EVAL_ERROR

$@

The Perl syntax error message from the last *eval()* command. If null, the last *eval()* parsed and executed correctly (although the operations you invoked may have failed in the normal fashion). (Mnemonic: Where was the syntax error "at"?)

Note that warning messages are not collected in this variable. You can, however, set up a routine to process warnings by setting **$SIG** {__WARN__} below.

$PROCESS_ID

$PID

$$

The process number of the Perl running this script. (Mnemonic: same as shells.)

$REAL_USER_ID

$UID

<

The real uid of this process. (Mnemonic: it's the uid you came *FROM* , if you're running setuid.)

$EFFECTIVE_USER_ID

$EUID

$>

The effective uid of this process. Example:

$< = $>; # set real to effective uid ($<,$>) = ($>,$<); # swap real and effective uid

(Mnemonic: it's the uid you went *TO* , if you're running setuid.) Note: " **$& lt;** " and " **$& gt;** " can only be swapped on machines supporting *setreuid()* .

$REAL_GROUP_ID

$GID

$(

The real gid of this process. If you are on a machine that supports membership in multiple groups simultaneously, gives a space separated list of groups you are in. The first number is the one returned by *getgid()* , and the subsequent ones by *getgroups()* , one of which may be the same as the first number. (Mnemonic: parentheses are used to *GROUP* things. The real gid is the group you *LEFT* , if you're running setgid.)

$EFFECTIVE_GROUP_ID

$EGID

$)

The effective gid of this process. If you are on a machine that supports membership in multiple groups simultaneously, gives a space separated list of groups you are in. The first number is the one returned by *getegid()* , and the subsequent ones by *getgroups()* , one of which may be the same as the first number. (Mnemonic: parentheses are used to *GROUP* things. The effective gid is the group that's *RIGHT* for you, if you're running setgid.)

Note: " **$& lt;** ", " **$& gt;** ", " $(" and " $) " can only be set on machines that support the corresponding *set[re][ug] id()* routine. " $(" and " $) " can only be swapped on machines supporting *setregid()* .

$PROGRAM_NAME

$0

Contains the name of the file containing the Perl script being executed. Assigning to " $0 " modifies the argument area that the ps(1) program sees. This is more useful as a way of indicating the current program state than it is for hiding the program you're running. (Mnemonic: same as **sh** and **ksh** .)

$[

The index of the first element in an array, and of the first character in a substring. Default is 0, but you could set it to 1 to make Perl behave more like **awk** (or Fortran) when subscripting and when evaluating the *index()* and *substr()* functions. (Mnemonic: [begins subscripts.)

As of Perl 5, assignment to " $[" is treated as a compiler directive, and cannot influence the behavior of any other file. Its use is discouraged.

$PERL_VERSION

$]

The string printed out when you say **perl -v** . It can be used to determine at the beginning of a script whether the perl interpreter executing the script is in the right range of versions. If used in a numeric context, returns the version + patchlevel / 1000. Example:

see if getc is available ($version,$patchlevel) = $] =~ /(\d+\.\d+).*\nPatch level: (\d+)/; print STDERR "(No filename completion available.)\n" if $version * 1000 + $patchlevel < 2016;

or, used numerically,

warn "No checksumming!\n" if $] < 3.019;

(Mnemonic: Is this version of perl in the right bracket?)

$DEBUGGING

$^ D

The current value of the debugging flags. (Mnemonic: value of **-D** switch.)

$SYSTEM_FD_MAX

$^ F

The maximum system file descriptor, ordinarily 2. System file descriptors are passed to *exec()* ed processes, while higher file descriptors are not. Also, during an *open()* , system file descriptors are preserved even if the *open()* fails. (Ordinary file descriptors are closed before the *open()* is attempted.) Note that the close-on-exec status of a file descriptor will be decided according to the value of $^ F at the time of the open, not the time of the exec.

$INPLACE_EDIT

$^ I

The current value of the inplace-edit extension. Use **undef** to disable inplace editing. (Mnemonic: value of **-i** switch.)

$PERLDB

$^ P

The internal flag that the debugger clears so that it doesn't debug itself. You could conceivable disable debugging yourself by clearing it.

$BASETIME

$^ T

The time at which the script began running, in seconds since the epoch (beginning of 1970). The values returned by the **-M** , **-A** and **-C** filetests are based on this value.

$WARNING

$^ W

The current value of the warning switch, either TRUE or FALSE. (Mnemonic: related to the **-w** switch.)

$EXECUTABLE_NAME

$^ X

The name that the Perl binary itself was executed as, from C's **argv[0]** .

$ARGV

contains the name of the current file when reading from<>.

@ARGV

The array @ARGV contains the command line arguments intended for the script. Note that **$# ARGV** is the generally number of arguments minus one, since **$ARGV [0]** is the first argument, *NOT* the command name. See " $0 " for the command name.

@INC

The array @INC contains the list of places to look for Perl scripts to be evaluated by the **do EXPR** , **require** , or **use** constructs. It initially consists of the arguments to any **-I** command line switches, followed by the default Perl library, probably "/usr/local/lib/perl", followed by ".", to represent the current directory.

%INC

The hash %INC contains entries for each filename that has been included via **do** or **require** . The key is the filename you specified, and the value is the location of the file actually found. The **require** command uses this array to determine whether a given file has already been included.

$ENV {expr}

The hash **%ENV** contains your current environment. Setting a value in **ENV** changes the environment for child processes.

$SIG {expr}

The hash **%SIG** is used to set signal handlers for various signals. Example:

sub handler { # 1st argument is signal name local($sig) = @_; print "Caught a SIG$sig--shutting down\n"; close(LOG); exit(0); } $SIG{'INT'} = 'handler'; $SIG{'QUIT'} = 'handler'; ... $SIG{'INT'} = 'DEFAULT'; # restore default action $SIG{'QUIT'} = 'IGNORE'; # ignore SIGQUIT

The **%SIG** array only contains values for the signals actually set within the Perl script. Here are some other examples:

$SIG{PIPE} = Plumber; # SCARY!! $SIG{"PIPE"} = "Plumber"; # just fine, assumes main::Plumber $SIG{"PIPE"} = \&Plumber; # just fine; assume current Plumber $SIG{"PIPE"} = Plumber(); # oops, what did Plumber() return??

The one marked scary is problematic because it's a bareword, which means sometimes it's a string representing the function, and sometimes it's going to call the subroutine call right then and there! Best to be sure and quote it or take a reference to it. *Plumber works too. See the *perlsubs* manpage .

Certain internal hooks can be also set using the **%SIG** hash. The routine indicated by **$SIG** {__WARN__} is called when a warning message is about to be printed. The warning message is passed as the first argument. The presence of a __WARN__ hook causes the ordinary printing of warnings to STDERR to be suppressed. You can use this to save warnings in a variable, or turn warnings into fatal errors, like this:

local $SIG{__WARN__} = sub { die $_[0] }; eval $proggie;

The routine indicated by **$SIG** {__DIE__} is called when a fatal exception is about to be thrown. The error message is passed as the first argument. When a __DIE__ hook routine returns, the exception processing continues as it would have in the absence of the hook, unless the hook routine itself exits via a goto, a loop exit, or a die.

.

NAME

perlsub - Perl subroutines

SYNOPSIS

To declare subroutines:

sub NAME; # A "forward" declaration. sub NAME BLOCK # A declaration and a definition.

To define an anonymous subroutine at runtime:

$subref = sub BLOCK;

To import subroutines:

use PACKAGE qw(NAME1 NAME2 NAME3);

To call subroutines:

&NAME # Passes current @_ to subroutine. &NAME(LIST); # Parens required with & form. NAME(LIST); # & is optional with parens. NAME LIST; # Parens optional if predeclared/imported.

DESCRIPTION

Any arguments passed to the routine come in as array **@_** , that is (**$_** [0], **$_** [1], ...). The array **@_** is a local array, but its values are references to the actual scalar parameters. The return value of the subroutine is the value of the last expression evaluated, and can be either an array value or a scalar value. Alternatively, a return statement may be used to specify the returned value and exit the subroutine. To create local variables see the *local()* and *my()* operators.

A subroutine may be called using the "&" prefix. The "&" is optional in Perl 5, and so are the parens if the subroutine has been predeclared. (Note, however, that the "&" is *NOT* optional when you're just naming the subroutine, such as when it's used as an argument to *defined()* or *undef()* . Nor is it optional when you want to do an indirect subroutine call with a subroutine name or reference using the **& $< EM> subref()** or **&{ $subref }()** constructs. See the *perlref* manpage for more on that.)

Example:

sub MAX { my $max = pop(@_); foreach $foo (@_) { $max = $foo if $max < $foo; } $max; } ... $bestday = &MAX($mon,$tue,$wed,$thu,$fri);

Example:

get a line, combining continuation lines # that start with whitespace sub get_line { $thisline = $lookahead; LINE: while ($lookahead = <STDIN>) { if ($lookahead =~ /^[\t]/) { $thisline .= $lookahead; } else { last LINE; } } $thisline; } $lookahead = <STDIN>; # get first line while ($_ = get_line()) { ... }

Use array assignment to a local list to name your formal arguments:

sub maybeset { my($key, $value) = @_; $foo{$key} = $value unless $foo{$key}; }

This also has the effect of turning call-by-reference into call-by-value, since the assignment copies the values.

Subroutines may be called recursively. If a subroutine is called using the "&" form, the argument list is optional. If omitted, no **@_** array is set up for the subroutine; the **@_** array at the time of the call is visible to subroutine instead.

&foo(1,2,3); # pass three arguments foo(1,2,3); # the same foo(); # pass a null list &foo(); # the same &foo; # pass no arguments--more efficient

If a module wants to create a private subroutine that cannot be called from outside the module, it can declare a lexical variable containing an anonymous sub reference:

my $subref = sub { ... } &$subref(1,2,3);

As long as the reference is never returned by any function within the module, no outside module can see the subroutine, since its name is not in any package's symbol table.

Passing Symbol Table Entries

[Note: The mechanism described in this section works fine in Perl 5, but the new reference mechanism is generally easier to work with. See the *perlref* manpage .]

Sometimes you don't want to pass the value of an array to a subroutine but rather the name of it, so that the subroutine can modify the global copy of it rather than working with a local copy. In perl you can refer to all the objects of a particular name by prefixing the name with a star: ***foo** . This is often known as a "type glob", since the star on the front can be thought of as a wildcard match for all the funny prefix characters on variables and subroutines and such.

When evaluated, the type glob produces a scalar value that represents all the objects of that name, including any filehandle, format or subroutine. When assigned to, it causes the name mentioned to refer to whatever "*" value was assigned to it. Example:

sub doubleary { local(*someary) = @_; foreach $elem (@someary) { $elem *= 2; } } doubleary(*foo); doubleary(*bar);

Note that scalars are already passed by reference, so you can modify scalar arguments without using this mechanism by referring explicitly to **$_** [0] etc. You can modify all the elements of an array by passing all the elements as scalars, but you have to use the * mechanism (or the equivalent reference mechanism)

to push, pop or change the size of an array. It will certainly be faster to pass the typeglob (or referenc

Even if you don't want to modify an array, this mechanism is useful for passing multiple arrays in a single LIST, since normally the LIST mechanism will merge all the array values so that you can't extract out the individual arrays.

Overriding builtin functions

Many builtin functions may be overridden, though this should only be tried occasionally and for good reason. Typically this might be done by a package attempting to emulate missing builtin functionality on a non-Unix system.

Overriding may only be done by importing the name from a module--ordinary predeclaration isn't good enough. However, the **subs** pragma (compiler directive) lets you, in effect, predeclare subs via the import syntax, and these names may then override the builtin ones:

use subs 'chdir', 'chroot', 'chmod', 'chown'; chdir $somewhere; sub chdir { ... }

Library modules should not in general export builtin names like "open" or "chdir" as part of their default **@EXPORT** list, since these may sneak into someone else's namespace and change the semantics unexpectedly. Instead, if the module adds the name to the **@EXPORT_OK** list, then it's possible for a user to import the name explicitly, but not implicitly. That is, they could say

use Module 'open';

and it would import the open override, but if they said

use Module;

they would get the default imports without the overrides.

Autoloading

If you call a subroutine that is undefined, you would ordinarily get an immediate fatal error complaining that the subroutine doesn't exist. (Likewise for subroutines being used as methods, when the method doesn't exist in any of the base classes of the class package.) If, however, there is an **AUTOLOAD** subroutine defined in the package or packages that were searched for the original subroutine, then that **AUTOLOAD** subroutine is called with the arguments that would have been passed to the original subroutine. The fully qualified name of the original subroutine magically appears in the **$AUTOLOAD** variable in the same package as the **AUTOLOAD** routine. The name is not passed as an ordinary argument because, er, well, just because, that's why...

Most **AUTOLOAD** routines will load in a definition for the subroutine in question using eval, and then execute that subroutine using a special form of "goto" that erases the stack frame of the **AUTOLOAD** routine without a trace. (See the standard **AutoLoader** module, for example.) But an **AUTOLOAD** routine can also just emulate the routine and never define it. A good example of this is the standard Shell module, which can treat undefined subroutine calls as calls to Unix programs.

There are mechanisms available for modules to help them split themselves up into autoloadable files be used with the standard AutoLoader module. See the document on extensions.

TOC

NAME

perlmod - Perl modules (packages)

DESCRIPTION

Packages

Perl provides a mechanism for alternative namespaces to protect packages from stomping on each others variables. In fact, apart from certain magical variables, there's really no such thing as a global variable in Perl. By default, a Perl script starts compiling into the package known as **main** . You can switch namespaces using the **package** declaration. The scope of the package declaration is from the declaration itself to the end of the enclosing block (the same scope as the *local()* operator). Typically it would be the first declaration in a file to be included by the **require** operator. You can switch into a package in more than one place; it merely influences which symbol table is used by the compiler for the rest of that block. You can refer to variables and filehandles in other packages by prefixing the identifier with the package name and a double colon: **$Package::Variable** . If the package name is null, the **main** package as assumed. That is, **$::sail** is equivalent to **$main::sail** .

(The old package delimiter was a single quote, but double colon is now the preferred delimiter, in part because it's more readable to humans, and in part because it's more readable to **emacs** macros. It also makes C++ programmers feel like they know what's going on.)

Packages may be nested inside other packages: **$OUTER::INNER::var** . This implies nothing about the order of name lookups, however. All symbols are either local to the current package, or must be fully qualified from the outer package name down. For instance, there is nowhere within package **OUTER** that **$INNER::var** refers to **$OUTER::INNER::var** . It would treat package **INNER** as a totally separate global package.

Only identifiers starting with letters (or underscore) are stored in a package's symbol table. All other symbols are kept in package **main** . In addition, the identifiers STDIN, STDOUT, STDERR, ARGV, ARGVOUT, ENV, INC and SIG are forced to be in package **main** , even when used for other purposes than their built-in one. Note also that, if you have a package called **m** , **s** or **y** , then you can't use the qualified form of an identifier because it will be interpreted instead as a pattern match, a substitution, or a translation.

(Variables beginning with underscore used to be forced into package main, but we decided it was more useful for package writers to be able to use leading underscore to indicate private variables and method names.)

Eval() ed strings are compiled in the package in which the *eval()* was compiled. (Assignments to **$SIG {}**

, however, assume the signal handler specified is in the **main** package. Qualify the signal handler na you wish to have a signal handler in a package.) For an example, examine *perldb.pl* in the Perl libra initially switches to the **DB** package so that the debugger doesn't interfere with variables in the script you are trying to debug. At various points, however, it temporarily switches back to the **main** package to evaluate various expressions in the context of the **main** package (or wherever you came from). See the *perldebug* manpage .

Symbol Tables

The symbol table for a package happens to be stored in the associative array of that name appended with two colons. The main symbol table's name is thus **%main::** , or **%::** for short. Likewise the nested package mentioned earlier is named **%OUTER::INNER::** .

The value in each entry of the associative array is what you are referring to when you use the ***name** notation. In fact, the following have the same effect, though the first is more efficient because it does the symbol table lookups at compile time:

local(*main::foo) = *main::bar; local($main::{'foo'}) = $main::{'bar'};

You can use this to print out all the variables in a package, for instance. Here is *dumpvar.pl* from the Perl library:

package dumpvar; sub main::dumpvar { ($package) = @_; local(*stab) = eval("*${package}::"); while (($key,$val) = each(%stab)) { local(*entry) = $val; if (defined $entry) { print "\$$key = '$entry'\n"; } if (defined @entry) { print "\@$key = (\n"; foreach $num ($[.. $#entry) { print " $num\t'",$entry[$num],"'\n"; } print ")\n"; } if ($key ne "${package}::" && defined %entry) { print "\%$key = (\n"; foreach $key (sort keys(%entry)) { print " $key\t'",$entry{$key},"'\n"; } print ")\n"; } } }

Note that even though the subroutine is compiled in package **dumpvar** , the name of the subroutine is qualified so that its name is inserted into package **main** .

Assignment to a symbol table entry performs an aliasing operation, i.e.,

*dick = *richard;

causes variables, subroutines and file handles accessible via the identifier **richard** to also be accessible via the symbol **dick** . If you only want to alias a particular variable or subroutine, you can assign a reference instead:

*dick = \$richard;

makes **$richard** and **$dick** the same variable, but leaves **@richard** and **@dick** as separate arrays. Tricky, eh?

Package Constructors and Destructors

There are two special subroutine definitions that function as package constructors and destructors. These are the **BEGIN** and **END** routines. The **sub** is optional for these routines.

A **BEGIN** subroutine is executed as soon as possible, that is, the moment it is completely defined, even before the rest of the containing file is parsed. You may have multiple **BEGIN** blocks within a file--they will execute in order of definition. Because a **BEGIN** block executes immediately, it can pull in definitions of subroutines and such from other files in time to be visible to the rest of the file.

An **END** subroutine is executed as late as possible, that is, when the interpreter is being exited, even if it is exiting as a result of a *die()* function. (But not if it's is being blown out of the water by a signal--you have to trap that yourself (if you can).) You may have multiple **END** blocks within a file--they will execute in reverse order of definition; that is: last in, first out (LIFO).

Note that when you use the **-n** and **-p** switches to Perl, **BEGIN** and **END** work just as they do in **awk** , as a degenerate case.

Perl Classes

There is no special class syntax in Perl 5, but a package may function as a class if it provides subroutines that function as methods. Such a package may also derive some of its methods from another class package by listing the other package name in its **@ISA** array. For more on this, see the *perlobj* manpage .

Perl Modules

In Perl 5, the notion of packages has been extended into the notion of modules. A module is a package that is defined in a library file of the same name, and is designed to be reusable. It may do this by providing a mechanism for exporting some of its symbols into the symbol table of any package using it. Or it may function as a class definition and make its semantics available implicitly through method calls on the class and its objects, without explicit exportation of any symbols. Or it can do a little of both.

Perl modules are included by saying

use Module;

or

use Module LIST;

This is exactly equivalent to

BEGIN { require "Module.pm"; import Module; }

or

BEGIN { require "Module.pm"; import Module LIST; }

All Perl module files have the extension *.pm* . **use** assumes this so that you don't have to spell out " *Module.pm* " in quotes. This also helps to differentiate new modules from old *.pl* and *.ph* files. Modu names are also capitalized unless they're functioning as pragmas, "Pragmas" are in effect compiler directives, and are sometimes called "pragmatic modules" (or even "pragmata" if you're a classicist).

Because the **use** statement implies a **BEGIN** block, the importation of semantics happens at the moment the **use** statement is compiled, before the rest of the file is compiled. This is how it is able to function as a pragma mechanism, and also how modules are able to declare subroutines that are then visible as list operators for the rest of the current file. This will not work if you use **require** instead of **use** . Therefore, if you're planning on the module altering your namespace, use **use** ; otherwise, use **require** . Otherwise you can get into this problem:

require Cwd; # make Cwd:: accessible $here = Cwd::getcwd(); use Cwd; # import names from Cwd:: $here = getcwd(); require Cwd; # make Cwd:: accessible $here = getcwd(); # oops! no main::getcwd()

Perl packages may be nested inside other package names, so we can have package names containing **::** . But if we used that package name directly as a filename it would makes for unwieldy or impossible filenames on some systems. Therefore, if a module's name is, say, **Text::Soundex** , then its definition is actually found in the library file *Text/Soundex.pm* .

Perl modules always have a *.pm* file, but there may also be dynamically linked executables or autoloaded subroutine definitions associated with the module. If so, these will be entirely transparent to the user of the module. It is the responsibility of the *.pm* file to load (or arrange to autoload) any additional functionality. The POSIX module happens to do both dynamic loading and autoloading, but the user can just say **use POSIX** to get it all.

For more information on writing extension modules, see the *perlapi* manpage and the *perlguts* manpage .

NOTE

Perl does not enforce private and public parts of its modules as you may have been used to in other languages like C++, Ada, or Modula-17. Perl doesn't have an infatuation with enforced privacy. It would prefer that you stayed out of its living room because you weren't invited, not because it has a shotgun.

The module and its user have a contract, part of which is common law, and part of which is "written". Part of the common law contract is that a module doesn't pollute any namespace it wasn't asked to. The written contract for the module (AKA documentation) may make other provisions. But then you know when you **use RedefineTheWorld** that you're redefining the world and willing to take the consequences.

THE PERL MODULE LIBRARY

A number of modules are included the the Perl distribution. These are described below, and all end in *.pm* . You may also discover files in the library directory that end in either *.pl* or *.ph* . These are old libraries supplied so that old programs that use them still run. The *.pl* files will all eventually be converted into standard modules, and the *.ph* files made by **h2ph** will probably end up as extension modules made by **h2xs** . (Some *.ph* values may already be available through the POSIX module.) The **pl2pm** file in the distribution may help in your conversion, but it's just a mechanical process, so is far from bullet proof.

Pragmatic Modules

They work somewhat like pragmas in that they tend to affect the compilation of your program, and thus will usually only work well when used within a **use** , or **no** . These are locally scoped, so an inner BLOCK may countermand any of these by saying

no integer; no strict 'refs';

which lasts until the end of that BLOCK.

The following programs are defined (and have their own documentation).

integer
: Perl pragma to compute arithmetic in integer instead of double

less
: Perl pragma to request less of something from the compiler

sigtrap
: Perl pragma to enable stack backtrace on unexpected signals

strict
: Perl pragma to restrict unsafe constructs

subs
: Perl pragma to predeclare sub names

.

Standard Modules

The following modules are all expected to behave in a well-defined manner with respect to namespace pollution because they use the Exporter module. See their own documentation for details.

Abbrev
: create an abbreviation table from a list

AnyDBM_File
: provide framework for multiple DBMs

AutoLoader

load functions only on demand

AutoSplit

split a package for autoloading

Basename

parse file name and path from a specification

Benchmark

benchmark running times of code

Carp

warn or die of errors (from perspective of caller)

CheckTree

run many filetest checks on a tree

Collate

compare 8-bit scalar data according to the current locale

Config

access Perl configuration option

Cwd

get pathname of current working directory

DynaLoader

Dynamically load C libraries into Perl code

English

use nice English (or **awk**) names for ugly punctuation variables

Env

Perl module that imports environment variables

Exporter

module to control namespace manipulations

Fcntl

load the C Fcntl.h defines

FileHandle

supply object methods for filehandles

Find

traverse a file tree

Finddepth

traverse a directory structure depth-first

Getopt

basic and extended getopt(3) processing

MakeMaker

generate a Makefile for Perl extension

Open2

open a process for both reading and writing

Open3

open a process for reading, writing, and error handling

POSIX

Perl interface to IEEE 1003.1 namespace

Ping

check a host for upness

Socket

load the C socket.h defines

.

Extension Modules

Extension modules are written in C (or a mix of Perl and C) and get dynamically loaded into Perl if and when you need them. Supported extension modules include the Socket, Fcntl, and POSIX modules.

The following are popular C extension modules, which while available at Perl 5.0 release time, do not come bundled (at least, not completely) due to their size, volatility, or simply lack of time for adequate testing and configuration across the multitude of platforms on which Perl was beta-tested. You are encouraged to look for them in archie(1L), the Perl FAQ or Meta-FAQ, the WWW page, and even with their authors before randomly posting asking for their present condition and disposition. There's no guarantee that the names or addresses below have not changed since printing, and in fact, they probably have!

Curses

Written by William Setzer< *William_Setzer* ***@ncsu*** *.edu* >, while not included with the standard distribution, this extension module ports to most systems. FTP from your nearest Perl archive site, or try

ftp://ftp.ncsu.edu/pub/math/wsetzer/cursperl5??.tar.gz

It is currently in alpha test, so the name and ftp location may change.

DBI

This is the portable database interface written by < *Tim.Bunce* ***@ig*** *.co.uk* >. This supersedes the many perl4 ports for database extensions. The official archive for DBperl extensions is *ftp.demon.co.uk:/pub/perl/db* . This archive contains copies of perl4 ports for Ingres, Oracle, Sybase, Informix, Unify, Postgres, and Interbase, as well as rdb and shql and other non-SQL systems.

DB_File

Fastest and most restriction-free of the DBM bindings, this extension module uses the popular Berkeley DB to *tie()* into your hashes. This has a standardly-distributed man page and dynamic

loading extension module, but you'll have to fetch the Berkeley code yourself. See *DB_File* fo where.

Sx

This extension module is a front to the Athena and Xlib libraries for Perl GUI programming, originally written by by Dominic Giampaolo < *dbg* ***@sgi*** *.com* >, then and rewritten for Sx by Frédéric Chauveau< *fmc* ***@pasteur*** *.fr* >. It's available for FTP from

ftp.pasteur.fr:/pub/Perl/Sx.tar.gz

Tk

This extension module is an object-oriented Perl5 binding to the popular tcl/tk X11 package. However, you need know no TCL to use it! It was written by Malcolm Beattie< *mbeattie* ***@sable*** *.ox.ac.uk* >. If you are unable to locate it using archie(1L) or a similar tool, you may try retrieving it from */private/Tk-october.tar.gz* from Malcolm's machine listed above.

.

NAME

perlref - Perl references and nested data structures

DESCRIPTION

In Perl 4 it was difficult to represent complex data structures, because all references had to be symbolic, and even that was difficult to do when you wanted to refer to a variable rather than a symbol table entry. Perl 5 not only makes it easier to use symbolic references to variables, but lets you have "hard" references to any piece of data. Any scalar may hold a hard reference. Since arrays and hashes contain scalars, you can now easily build arrays of arrays, arrays of hashes, hashes of arrays, arrays of hashes of functions, and so on.

Hard references are smart--they keep track of reference counts for you, automatically freeing the thing referred to when its reference count goes to zero. If that thing happens to be an object, the object is destructed. See the *perlobj* manpage for more about objects. (In a sense, everything in Perl is an object, but we usually reserve the word for references to objects that have been officially "blessed" into a class package.)

A symbolic reference contains the name of a variable, just as a symbolic link in the filesystem merely contains the name of a file. The ***glob** notation is a kind of symbolic reference. Hard references are more like hard links in the file system: merely another way at getting at the same underlying object, irrespective of its name.

"Hard" references are easy to use in Perl. There is just one overriding principle: Perl does no implicit referencing or dereferencing. When a scalar is holding a reference, it always behaves as a scalar. It doesn't magically start being an array or a hash unless you tell it so explicitly by dereferencing it.

References can be constructed several ways.

1. By using the backslash operator on a variable, subroutine, or value. (This works much like the & (address-of) operator works in C.) Note that this typically creates *ANOTHER* reference to a variable, since there's already a reference to the variable in the symbol table. But the symbol table reference might go away, and you'll still have the reference that the backslash returned. Here are some examples:

   $scalarref = \$foo; $arrayref = \@ARGV; $hashref = \%ENV; $coderef = \&handler;
2. A reference to an anonymous array can be constructed using square brackets:

   $arrayref = [1, 2, ['a', 'b', 'c']];

   Here we've constructed a reference to an anonymous array of three elements whose final element is itself reference to another anonymous array of three elements. (The multidimensional syntax

described later can be used to access this. For example, after the above, **$arrayref** ->[2][1] would have the value "b".)

3. A reference to an anonymous hash can be constructed using curly brackets:

   $hashref = { 'Adam' => 'Eve', 'Clyde' => 'Bonnie', };

   Anonymous hash and array constructors can be intermixed freely to produce as complicated a structure as you want. The multidimensional syntax described below works for these too. The values above are literals, but variables and expressions would work just as well, because assignment operators in Perl (even within *local()* or *my()*) are executable statements, not compile-time declarations.

   Because curly brackets (braces) are used for several other things including BLOCKs, you may occasionally have to disambiguate braces at the beginning of a statement by putting a **+** or a **return** in front so that Perl realizes the opening brace isn't starting a BLOCK. The economy and mnemonic value of using curlies is deemed worth this occasional extra hassle.

   For example, if you wanted a function to make a new hash and return a reference to it, you have these options:

   sub hashem { { @_ } } # silently wrong sub hashem { +{ @_ } } # ok sub hashem { return { @_ } } # ok

4. A reference to an anonymous subroutine can be constructed by using **sub** without a subname:

   $coderef = sub { print "Boink!\n" };

   Note the presence of the semicolon. Except for the fact that the code inside isn't executed immediately, a **sub {}** is not so much a declaration as it is an operator, like **do{}** or **eval{}** . (However, no matter how many times you execute that line (unless you're in an **eval("...")**), **$coderef** will still have a reference to the *SAME* anonymous subroutine.)

   Anonymous subroutines act as closures with respect to *my()* variables, that is, variables visible lexically within the current scope. Closure is a notion out of the Lisp world that says if you define an anonymous function in a particular lexical context, it pretends to run in that context even when it's called outside of the context.

   In human terms, it's a funny way of passing arguments to a subroutine when you define it as well as when you call it. It's useful for setting up little bits of code to run later, such as callbacks. You can even do object-oriented stuff with it, though Perl provides a different mechanism to do that already--see the *perlobj* manpage .

   You can also think of closure as a way to write a subroutine template without using eval. (In fact, in version 5.000, eval was the *only* way to get closures. You may wish to use "require 5.001" if you use closures.)

   Here's a small example of how closures works:

   sub newprint { my $x = shift; return sub { my $y = shift; print "$x, $y!\n"; }; } $h = newprint("Howdy"); $g = newprint("Greetings"); # Time passes... &$h("world"); &$g("earthlings");

This prints

Howdy, world! Greetings, earthlings!

Note particularly that **$x** continues to refer to the value passed into *newprint()* *despite* the fact that the "my **$x** " has seemingly gone out of scope by the time the anonymous subroutine runs. That's what closure is all about.

This only applies to lexical variables, by the way. Dynamic variables continue to work as they have always worked. Closure is not something that most Perl programmers need trouble themselves about to begin with.

5. References are often returned by special subroutines called constructors. Perl objects are just references to a special kind of object that happens to know which package it's associated with. Constructors are just special subroutines that know how to create that association. They do so by starting with an ordinary reference, and it remains an ordinary reference even while it's also being an object. Constructors are customarily named *new()* , but don't have to be:

   $objref = new Doggie (Tail => 'short', Ears => 'long');

6. References of the appropriate type can spring into existence if you dereference them in a context that assumes they exist. Since we haven't talked about dereferencing yet, we can't show you any examples yet.

That's it for creating references. By now you're probably dying to.know how to use references to get back to your long-lost data. There are several basic methods.

1. Anywhere you'd put an identifier as part of a variable or subroutine name, you can replace the identifier with a simple scalar variable containing a reference of the correct type:

   $bar = $$scalarref; push(@$arrayref, $filename); $$arrayref[0] = "January"; $$hashref{"KEY"} = "VALUE"; &$coderef(1,2,3);

   It's important to understand that we are specifically *NOT* dereferencing **$arrayref [0]** or **$hashref {"KEY"}** there. The dereference of the scalar variable happens *BEFORE* it does any key lookups. Anything more complicated than a simple scalar variable must use methods 2 or 3 below. However, a "simple scalar" includes an identifier that itself uses method 1 recursively. Therefore, the following prints "howdy".

   $refrefref = \\\"howdy"; print $$$$refrefref;

2. Anywhere you'd put an identifier as part of a variable or subroutine name, you can replace the identifier with a BLOCK returning a reference of the correct type. In other words, the previous examples could be written like this:

   $bar = ${$scalarref}; push(@{$arrayref}, $filename); ${$arrayref}[0] = "January"; ${$hashref}{"KEY"} = "VALUE"; &{$coderef}(1,2,3);

   Admittedly, it's a little silly to use the curlies in this case, but the BLOCK can contain any arbitrary expression, in particular, subscripted expressions:

   &{ $dispatch{$index} }(1,2,3); # call correct routine

Because of being able to omit the curlies for the simple case of **$$ x** , people often make the mistake of viewing the dereferencing symbols as proper operators, and wonder about their precedence. If they were, though, you could use parens instead of braces. That's not the case. Consider the difference below; case 0 is a short-hand version of case 1, *NOT* case 2:

$$hashref{"KEY"} = "VALUE"; # CASE 0 ${$hashref}{"KEY"} = "VALUE"; # CASE 1 ${$hashref{"KEY"}} = "VALUE"; # CASE 2 ${$hashref->{"KEY"}} = "VALUE"; # CASE 3

Case 2 is also deceptive in that you're accessing a variable called **%hashref** , not dereferencing through **$hashref** to the hash it's presumably referencing. That would be case 3.

3. The case of individual array elements arises often enough that it gets cumbersome to use method 2. As a form of syntactic sugar, the two lines like that above can be written:

   $arrayref->[0] = "January"; $hashref->{"KEY"} = "VALUE";

   The left side of the array can be any expression returning a reference, including a previous dereference. Note that **$array [$x]** is *NOT* the same thing as **$array ->[$x]** here:

   $array[$x]->{"foo"}->[0] = "January";

   This is one of the cases we mentioned earlier in which references could spring into existence when in an lvalue context. Before this statement, **$array [$x]** may have been undefined. If so, it's automatically defined with a hash reference so that we can look up **{"foo"}** in it. Likewise **$array [$x]->{"foo"}** will automatically get defined with an array reference so that we can look up **[0]** in it.

   One more thing here. The arrow is optional *BETWEEN* brackets subscripts, so you can shrink the above down to

   $array[$x]{"foo"}[0] = "January";

   Which, in the degenerate case of using only ordinary arrays, gives you multidimensional arrays just like C's:

   $score[$x][$y][$z] += 42;

   Well, okay, not entirely like C's arrays, actually. C doesn't know how to grow its arrays on demand. Perl does.

4. If a reference happens to be a reference to an object, then there are probably methods to access the things referred to, and you should probably stick to those methods unless you're in the class package that defines the object's methods. In other words, be nice, and don't violate the object's encapsulation without a very good reason. Perl does not enforce encapsulation. We are not totalitarians here. We do expect some basic civility though.

The *ref()* operator may be used to determine what type of thing the.reference is pointing to. See the *perlfunc* manpage .

The *bless()* operator may be used to associate a reference with a package functioning as an object class. See the *perlobj* manpage .

A type glob may be dereferenced the same way a reference can, since the dereference syntax always

indicates the kind of reference desired. So **${*foo}** and **${\ $foo }** both indicate the same scalar variable.

Here's a trick for interpolating a subroutine call into a string:

print "My sub returned ${\mysub(1,2,3)}\n";

The way it works is that when the **${...}** is seen in the double-quoted string, it's evaluated as a block. The block executes the call to **mysub(1,2,3)** , and then takes a reference to that. So the whole block returns a reference to a scalar, which is then dereferenced by **${...}** and stuck into the double-quoted string.

Symbolic references

We said that references spring into existence as necessary if they are undefined, but we didn't say what happens if a value used as a reference is already defined, but *ISN'T* a hard reference. If you use it as a reference in this case, it'll be treated as a symbolic reference. That is, the value of the scalar is taken to be the NAME of a variable, rather than a direct link to a (possibly) anonymous value.

People frequently expect it to work like this. So it does.

$name = "foo"; $$name = 1; # Sets $foo ${$name} = 2; # Sets $foo ${$name x 2} = 3; # Sets $foofoo $name->[0] = 4; # Sets $foo[0] @$name = (); # Clears @foo &$name(); # Calls &foo() (as in Perl 4) $pack = "THAT"; ${"${pack}::$name"} = 5; # Sets $THAT::foo without eval

This is very powerful, and slightly dangerous, in that it's possible to intend (with the utmost sincerity) to use a hard reference, and accidentally use a symbolic reference instead. To protect against that, you can say

use strict 'refs';

and then only hard references will be allowed for the rest of the enclosing block. An inner block may countermand that with

no strict 'refs';

Only package variables are visible to symbolic references. Lexical variables (declared with *my()*) aren't in a symbol table, and thus are invisible to this mechanism. For example:

local($value) = 10; $ref = \$value; { my $value = 20; print $$ref; }

This will still print 10, not 20. Remember that *local()* affects package variables, which are all "global" to the package.

Not-so-symbolic references

A new feature contributing to readability in 5.001 is that the brackets around a symbolic reference behave more like quotes, just as they always have within a string. That is,

$push = "pop on "; print "${push}over";

TOC

has always meant to print "pop on over", despite the fact that push is a reserved word. This has been generalized to work the same outside of quotes, so that

print ${push} . "over";

and even

print ${ push } . "over";

will have the same effect. (This would have been a syntax error in 5.000, though Perl 4 allowed it in the spaceless form.) Note that this construct is *not* considered to be a symbolic reference when you're using strict refs:

use strict 'refs'; ${ bareword }; # Okay, means $bareword. ${ "bareword" }; # Error, symbolic reference.

Similarly, because of all the subscripting that is done using single words, we've applied the same rule to any bareword that is used for subscripting a hash. So now, instead of writing

$array{ "aaa" }{ "bbb" }{ "ccc" }

you can just write

$array{ aaa }{ bbb }{ ccc }

and not worry about whether the subscripts are reserved words. In the rare event that you do wish to do something like

$array{ shift }

you can force interpretation as a reserved word by adding anything that makes it more than a bareword:

$array{ shift() } $array{ +shift } $array{ shift @_ }

The **-w** switch will warn you if it interprets a reserved word as a string. But it will no longer warn you about using lowercase words, since the string is effectively quoted.

WARNING

You may not (usefully) use a reference as the key to a hash. It will be converted into a string:

$x{ \$a } = $a;

If you try to dereference the key, it won't do a hard dereference, and you won't accomplish what you're attemping.

Further Reading

Besides the obvious documents, source code can be instructive. Some rather pathological examples of the use of references can be found in the *t/op/ref.t* regression test in the Perl source directory.

NAME

perlobj - Perl objects

DESCRIPTION

First of all, you need to understand what references are in Perl. See the *perlref* manpage for that.

Here are three very simple definitions that you should find reassuring.

1. An object is simply a reference that happens to know which class it belongs to.
2. A class is simply a package that happens to provide methods to deal with object references.
3. A method is simply a subroutine that expects an object reference (or a package name, for static methods) as the first argument.

We'll cover these points now in more depth..

An Object is Simply a Reference

Unlike say C++, Perl doesn't provide any special syntax for constructors. A constructor is merely a subroutine that returns a reference that has been "blessed" into a class, generally the class that the subroutine is defined in. Here is a typical constructor:

package Critter; sub new { bless {} }

The **{}** constructs a reference to an anonymous hash containing no key/value pairs. The *bless()* takes that reference and tells the object it references that it's now a Critter, and returns the reference. This is for convenience, since the referenced object itself knows that it has been blessed, and its reference to it could have been returned directly, like this:

sub new { my $self = {}; bless $self; return $self; }

In fact, you often see such a thing in more complicated constructors that wish to call methods in the class as part of the construction:

sub new { my $self = {} bless $self; $self->initialize(); $self; }

Within the class package, the methods will typically deal with the reference as an ordinary reference. Outside the class package, the reference is generally treated as an opaque value that may only be accessed through the class's methods.

A constructor may re-bless a referenced object currently belonging to another class, but then the new class is responsible for all cleanup later. The previous blessing is forgotten, as an object may only belong to one class at a time. (Although of course it's free to inherit methods from many classes.)

A clarification: Perl objects are blessed. References are not. Objects know which package they belon References do not. The *bless()* function simply uses the reference in order to find the object. Conside following example:

$a = {}; $b = $a; bless $a, BLAH; print "\$b is a ", ref($b), "\n";

This reports **$b** as being a BLAH, so obviously *bless()* operated on the object and not on the reference.

A Class is Simply a Package

Unlike say C++, Perl doesn't provide any special syntax for class definitions. You just use a package as a class by putting method definitions into the class.

There is a special array within each package called **@ISA** which says where else to look for a method if you can't find it in the current package. This is how Perl implements inheritance. Each element of the **@ISA** array is just the name of another package that happens to be a class package. The classes are searched (depth first) for missing methods in the order that they occur in **@ISA** . The classes accessible through **@ISA** are known as base classes of the current class.

If a missing method is found in one of the base classes, it is cached in the current class for efficiency. Changing **@ISA** or defining new subroutines invalidates the cache and causes Perl to do the lookup again.

If a method isn't found, but an AUTOLOAD routine is found, then that is called on behalf of the missing method.

If neither a method nor an AUTOLOAD routine is found in **@ISA** , then one last try is made for the method (or an AUTOLOAD routine) in a class called UNIVERSAL. If that doesn't work, Perl finally gives up and complains.

Perl classes only do method inheritance. Data inheritance is left up to the class itself. By and large, this is not a problem in Perl, because most classes model the attributes of their object using an anonymous hash, which serves as its own little namespace to be carved up by the various classes that might want to do something with the object.

A Method is Simply a Subroutine

Unlike say C++, Perl doesn't provide any special syntax for method definition. (It does provide a little syntax for method invocation though. More on that later.) A method expects its first argument to be the object or package it is being invoked on. There are just two types of methods, which we'll call static and virtual, in honor of the two C++ method types they most closely resemble.

A static method expects a class name as the first argument. It provides functionality for the class as a whole, not for any individual object belonging to the class. Constructors are typically static methods. Many static methods simply ignore their first argument, since they already know what package they're in, and don't care what package they were invoked via. (These aren't necessarily the same, since static methods follow the inheritance tree just like ordinary virtual methods.) Another typical use for static

methods is to look up an object by name:

sub find { my ($class, $name) = @_; $objtable{$name}; }

A virtual method expects an object reference as its first argument. Typically it shifts the first argument into a "self" or "this" variable, and then uses that as an ordinary reference.

sub display { my $self = shift; my @keys = @_ ? @_ : sort keys %$self; foreach $key (@keys) { print "\t$key => $self->{$key}\n"; } }

Method Invocation

There are two ways to invoke a method, one of which you're already familiar with, and the other of which will look familiar. Perl 4 already had an "indirect object" syntax that you use when you say

print STDERR "help!!!\n";

This same syntax can be used to call either static or virtual methods. We'll use the two methods defined above, the static method to lookup an object reference and the virtual method to print out its attributes.

$fred = find Critter "Fred"; display $fred 'Height', 'Weight';

These could be combined into one statement by using a BLOCK in the indirect object slot:

display {find Critter "Fred"} 'Height', 'Weight';

For C++ fans, there's also a syntax using -> notation that does exactly the same thing. The parentheses are required if there are any arguments.

$fred = Critter->find("Fred"); $fred->display('Height', 'Weight');

or in one statement,

Critter->find("Fred")->display('Height', 'Weight');

There are times when one syntax is more readable, and times when the other syntax is more readable. The indirect object syntax is less cluttered, but it has the same ambiguity as ordinary list operators. Indirect object method calls are parsed using the same rule as list operators: "If it looks like a function, it is a function". (Presuming for the moment that you think two words in a row can look like a function name. C++ programmers seem to think so with some regularity, especially when the first word is "new".) Thus, the parens of

new Critter ('Barney', 1.5, 70)

are assumed to surround ALL the arguments of the method call, regardless of what comes after. Saying

new Critter ('Bam' x 2), 1.4, 45

would be equivalent to

Critter->new('Bam' x 2), 1.4, 45

which is unlikely to do what you want.

There are times when you wish to specify which class's method to use. In this case, you can call your method as an ordinary subroutine call, being sure to pass the requisite first argument explicitly:

$fred = MyCritter::find("Critter", "Fred"); MyCritter::display($fred, 'Height', 'Weight');

Note however, that this does not do any inheritance. If you merely wish to specify that Perl should *START* looking for a method in a particular package, use an ordinary method call, but qualify the method name with the package like this:

$fred = Critter->MyCritter::find("Fred"); $fred->MyCritter::display('Height', 'Weight');

Sometimes you want to call a method when you don't know the method name ahead of time. You can use the arrow form, replacing the method name with a simple scalar variable containing the method name:

$method = $fast ? "findfirst" : "findbest"; $fred->$method(@args);

Destructors

When the last reference to an object goes away, the object is automatically destroyed. (This may even be after you exit, if you've stored references in global variables.) If you want to capture control just before the object is freed, you may define a DESTROY method in your class. It will automatically be called at the appropriate moment, and you can do any extra cleanup you need to do.

Perl doesn't do nested destruction for you. If your constructor reblessed a reference from one of your base classes, your DESTROY may need to call DESTROY for any base classes that need it. But this only applies to reblessed objects--an object reference that is merely *CONTAINED* in the current object will be freed and destroyed automatically when the current object is freed.

WARNING

An indirect object is limited to a name, a scalar variable, or a block, because it would have to do too much lookahead otherwise, just like any other postfix dereference in the language. The left side of -> is not so limited, because it's an infix operator, not a postfix operator.

That means that below, A and B are equivalent to each other, and C and D are equivalent, but AB and CD are different:

A: method $obref->{"fieldname"} B: (method $obref)->{"fieldname"} C: $obref->{"fieldname"}->method() D: method {$obref->{"fieldname"}}

Summary

That's about all there is to it. Now you just need to go off and buy a book about object-oriented design methodology, and bang your forehead with it for the next six months or so.

SEE ALSO

You should also check out the *perlbot* manpage for other object tricks, traps, and tips.

TOC

NAME

perlbot - Bag'o Object Tricks For Perl5 (the BOT)

INTRODUCTION

The following collection of tricks and hints is intended to whet curious appetites about such things as the use of instance variables and the mechanics of object and class relationships. The reader is encouraged to consult relevant textbooks for discussion of Object Oriented definitions and methodology. This is not intended as a comprehensive guide to Perl5's object oriented features, nor should it be construed as a style guide.

The Perl motto still holds: There's more than one way to do it.

INSTANCE VARIABLES

An anonymous array or anonymous hash can be used to hold instance variables. Named parameters are also demonstrated.

package Foo; sub new { my $type = shift; my %params = @_; my $self = {}; $self->{'High'} = $params{'High'}; $self->{'Low'} = $params{'Low'}; bless $self; } package Bar; sub new { my $type = shift; my %params = @_; my $self = []; $self->[0] = $params{'Left'}; $self->[1] = $params{'Right'}; bless $self; } package main; $a = new Foo ('High' => 42, 'Low' => 11); print "High=$a->{'High'}\n"; print "Low=$a->{'Low'}\n"; $b = new Bar ('Left' => 78, 'Right' => 40); print "Left=$b->[0]\n"; print "Right=$b->[1]\n";

SCALAR INSTANCE VARIABLES

An anonymous scalar can be used when only one instance variable is needed.

package Foo; sub new { my $type = shift; my $self; $self = shift; bless \$self; } package main; $a = new Foo 42; print "a=$$a\n";

INSTANCE VARIABLE INHERITANCE

This example demonstrates how one might inherit instance variables from a superclass for inclusion in the new class. This requires calling the superclass's constructor and adding one's own instance variables to the new object.

package Bar; sub new { my $self = {}; $self->{'buz'} = 42; bless $self; } package Foo; @ISA = qw(Bar); sub new { my $self = new Bar; $self->{'biz'} = 11; bless $self; } package main; $a = new Foo; print "buz = ", $a->{'buz'}, "\n"; print "biz = ", $a->{'biz'}, "\n";

OBJECT RELATIONSHIPS

The following demonstrates how one might implement "containing" and "using" relationships between objects.

package Bar; sub new { my $self = {}; $self->{'buz'} = 42; bless $self; } package Foo; sub new { my $self = {}; $self->{'Bar'} = new Bar (); $self->{'biz'} = 11; bless $self; } package main; $a = new Foo; print "buz = ", $a->{'Bar'}->{'buz'}, "\n"; print "biz = ", $a->{'biz'}, "\n";

OVERRIDING SUPERCLASS METHODS

The following example demonstrates how one might override a superclass method and then call the method after it has been overridden. The Foo::Inherit class allows the programmer to call an overridden superclass method without actually knowing where that method is defined.

package Buz; sub goo { print "here's the goo\n" } package Bar; @ISA = qw(Buz); sub google { print "google here\n" } package Baz; sub mumble { print "mumbling\n" } package Foo; @ISA = qw(Bar Baz); @Foo::Inherit::ISA = @ISA; # Access to overridden methods. sub new { bless [] } sub grr { print "grumble\n" } sub goo { my $self = shift; $self->Foo::Inherit::goo(); } sub mumble { my $self = shift; $self->Foo::Inherit::mumble(); } sub google { my $self = shift; $self->Foo::Inherit::google(); } package main; $foo = new Foo; $foo->mumble; $foo->grr; $foo->goo; $foo->google;

USING RELATIONSHIP WITH SDBM

This example demonstrates an interface for the SDBM class. This creates a "using" relationship between the SDBM class and the new class Mydbm.

use SDBM_File; use POSIX; package Mydbm; sub TIEHASH { my $self = shift; my $ref = SDBM_File->new(@_); bless {'dbm' => $ref}; } sub FETCH { my $self = shift; my $ref = $self->{'dbm'}; $ref->FETCH(@_); } sub STORE { my $self = shift; if (defined $_[0]){ my $ref = $self->{'dbm'}; $ref->STORE(@_); } else { die "Cannot STORE an undefined key in Mydbm\n"; } } package main; tie %foo, Mydbm, "Sdbm", O_RDWR|O_CREAT, 0640; $foo{'bar'} = 123; print "foo-bar = $foo{'bar'}\n"; tie %bar, Mydbm, "Sdbm2", O_RDWR|O_CREAT, 0640; $bar{'Cathy'} = 456; print "bar-Cathy = $bar{'Cathy'}\n";

THINKING OF CODE REUSE

One strength of Object-Oriented languages is the ease with which old code can use new code. The following examples will demonstrate first how one can hinder code reuse and then how one can promote code reuse.

This first example illustrates a class which uses a fully-qualified method call to access the "private" method *BAZ()* . The second example will show that it is impossible to override the *BAZ()* method.

package FOO; sub new { bless {} } sub bar { my $self = shift; $self->FOO::private::BAZ; } package FOO::private; sub BAZ { print "in BAZ\n"; } package main; $a = FOO->new; $a->bar;

Now we try to override the *BAZ()* method. We would like FOO:: *bar()* to call GOOP:: *BAZ()* , but this cannot happen since FOO:: *bar()* explicitly calls FOO::private:: *BAZ()* .

package FOO; sub new { bless {} } sub bar { my $self = shift; $self->FOO::private::BAZ; } package FOO::private; sub BAZ { print "in BAZ\n"; } package GOOP; @ISA = qw(FOO); sub new { bless {} } sub BAZ { print "in GOOP::BAZ\n"; } package main; $a = GOOP->new; $a->bar;

To create reusable code we must modify class FOO, flattening class FOO::private. The next example shows a reusable class FOO which allows the method GOOP:: *BAZ()* to be used in place of FOO:: *BAZ()* .

package FOO; sub new { bless {} } sub bar { my $self = shift; $self->BAZ; } sub BAZ { print "in BAZ\n"; } package GOOP; @ISA = qw(FOO); sub new { bless {} } sub BAZ { print "in GOOP::BAZ\n"; } package main; $a = GOOP->new; $a->bar;

CLASS CONTEXT AND THE OBJECT

TOC

Use the object to solve package and class context problems. Everything a method needs should be available via the object or should be passed as a parameter to the method.

A class will sometimes have static or global data to be used by the methods. A subclass may want to override that data and replace it with new data. When this happens the superclass may not know how to find the new copy of the data.

This problem can be solved by using the object to define the context of the method. Let the method look in the object for a reference to the data. The alternative is to force the method to go hunting for the data ("Is it in my class, or in a subclass? Which subclass?"), and this can be inconvenient and will lead to hackery. It is better to just let the object tell the method where that data is located.

package Bar; %fizzle = ('Password' => 'XYZZY'); sub new { my $self = {}; $self->{'fizzle'} = \%fizzle; bless $self; } sub enter { my $self = shift; # Don't try to guess if we should use %Bar::fizzle # or %Foo::fizzle. The object already knows which # we should use, so just ask it. # my $fizzle = $self->{'fizzle'}; print "The word is ", $fizzle->{'Password'}, "\n"; } package Foo; @ISA = qw(Bar); %fizzle = ('Password' => 'Rumple'); sub new { my $self = Bar->new; $self->{'fizzle'} = \%fizzle; bless $self; } package main; $a = Bar->new; $b = Foo->new; $a->enter; $b->enter;

NAME

perldebug - Perl debugging

DESCRIPTION

First of all, have you tried using the **-w** switch?

Debugging

If you invoke Perl with a **-d** switch, your script will be run under the debugger. However, the Perl debugger is not a separate program as it is in a C environment. Instead, the **-d** flag tells the compiler to insert source information into the pseudocode it's about to hand to the interpreter. (That means your code must compile correctly for the debugger to work on it.) Then when the interpreter starts up, it pre-loads a Perl library file containing the debugger itself. The program will halt before the first executable statement (but see below) and ask you for one of the following commands:

h

Prints out a help message.

T

Stack trace. If you do bizarre things to your **@_** arguments in a subroutine, the stack backtrace will not always show the original values.

s

Single step. Executes until it reaches the beginning of another statement.

n

Next. Executes over subroutine calls, until it reaches the beginning of the next statement.

f

Finish. Executes statements until it has finished the current subroutine.

c

Continue. Executes until the next breakpoint is reached.

c line

Continue to the specified line. Inserts a one-time-only breakpoint at the specified line.

Repeat last n or s.

l min+incr

List incr+1 lines starting at min. If min is omitted, starts where last listing left off. If incr is omitted, previous value of incr is used.

l min-max

List lines in the indicated range.

l line

List just the indicated line.

l

List next window.

-

List previous window.

w line

List window (a few lines worth of code) around line.

l subname

List subroutine. If it's a long subroutine it just lists the beginning. Use "l" to list more.

/pattern/

Regular expression search forward in the source code for pattern; the final / is optional.

?pattern?

Regular expression search backward in the source code for pattern; the final ? is optional.

L

List lines that have breakpoints or actions.

S

Lists the names of all subroutines.

t

Toggle trace mode on or off.

b line [condition]

Set a breakpoint. If line is omitted, sets a breakpoint on the line that is about to be executed. If a condition is specified, it is evaluated each time the statement is reached and a breakpoint is taken only if the condition is true. Breakpoints may only be set on lines that begin an executable statement. Conditions don't use **if** :

b 237 $x > 30 b 33 /pattern/i

b subname [condition]

Set breakpoint at first executable line of subroutine.

d line

Delete breakpoint. If line is omitted, deletes the breakpoint on the line that is about to be executed.

D

Delete all breakpoints.

a line command

Set an action for line. A multiline command may be entered by backslashing the newlines. This command is Perl code, not another debugger command.

A

Delete all line actions.

< command

Set an action to happen before every debugger prompt. A multiline command may be entered by backslashing the newlines.

> command

Set an action to happen after the prompt when you've just given a command to return to executing the script. A multiline command may be entered by backslashing the newlines.

V package [symbols]

Display all (or some) variables in package (defaulting to the **main** package) using a data pretty-printer (hashes show their keys and values so you see what's what, control characters are made printable, etc.). Make sure you don't put the type specifier (like $) there, just the symbol names, like this:

V DB filename line

X [symbols]

Same as as "V" command, but within the current package.

! number

Redo a debugging command. If number is omitted, redoes the previous command.

! -number

Redo the command that was that many commands ago.

H -number

Display last n commands. Only commands longer than one character are listed. If number is omitted, lists them all.

q or ^D

Quit. ("quit" doesn't work for this.)

command

Execute command as a Perl statement. A missing semicolon will be supplied.

p expr

Same as **print DB::OUT expr** . The DB::OUT filehandle is opened to /dev/tty, regardless of where STDOUT may be redirected to.

Any command you type in that isn't recognized by the debugger will be.directly executed (**eval** 'd) as Perl code. Leading white space will cause the debugger to think it's **NOT** a debugger command.

If you have any compile-time executable statements (code within a BEGIN block or a **use** statement), these will *NOT* be stopped by debugger, although **require** s will. From your own code, however, you can transfer control back to the debugger using the following statement, which is harmless if the debugger is not running:

$DB::single = 1;

Customization

If you want to modify the debugger, copy *perl5db.pl* from the Perl library to another name and modify it as necessary. You'll also want to set environment variable PERL5DB to say something like this:

BEGIN { require "myperl5db.pl" }

You can do some customization by setting up a *.perldb* file which contains initialization code. For instance, you could make aliases like these (the last one in particular most people seem to expect to be there):

$DB::alias{'len'} = 's/^len(.*)/p length($1)/'; $DB::alias{'stop'} = 's/^stop (at|in)/b/'; $DB::alias{'.'} = 's/^\./p ' . '"\$DB::sub(\$DB::filename:\$DB::line):\t"' . ',\$DB::dbline[\$DB::line]/' ;

Other resources

You did try the **-w** switch, didn't you?

BUGS

If your program *exit()* s or *die()* s, so does the debugger.

There's no builtin way to restart the debugger without exiting and coming back into it. You could use an alias like this:

$DB::alias{'rerun'} = 'exec "perl -d $DB::filename"';

But you'd lose any pending breakpoint information, and that might not be the right path, etc.

NAME

perldiag - various Perl diagnostics

DESCRIPTION

These messages are classified as follows (listed in increasing order of desperation):

(W) A warning (optional). (D) A deprecation (optional). (S) A severe warning (mandatory). (F) A fatal error (trappable). (P) An internal error you should never see (trappable). (X) A very fatal error (non-trappable).

Optional warnings are enabled by using the **-w** switch. Warnings may be captured by setting **$^ Q** to a reference to a routine that will be called on each warning instead of printing it. See the *perlvar* manpage . Trappable errors may be trapped using the eval operator. See *eval* .

Some of these messages are generic. Spots that vary are denoted with a %s , just as in a printf format. Note that some message start with a %s ! The symbols **"%-?@** sort before the letters, while **[** and \ sort after.

"my" variable %s can't be in a package

(F) Lexically scoped variables aren't in a package, so it doesn't make sense to try to declare one with a package qualifier on the front. Use *local()* if you want to localize a package variable.

"no" not allowed in expression

(F) The "no" keyword is recognized and executed at compile time, and returns no useful value. See the *perlmod* manpage .

"use" not allowed in expression

(F) The "use" keyword is recognized and executed at compile time, and returns no useful value. See the *perlmod* manpage .

% may only be used in unpack

(F) You can't pack a string by supplying a checksum, since the checksumming process loses information, and you can't go the other way. See *unpack* .

%s (...) interpreted as function

(W) You've run afoul of the rule that says that any list operator followed by parentheses turns into a function, with all the list operators arguments found inside the parens. See *Terms and List Operators (Leftward)* .

%s argument is not a HASH element

(F) The argument to *delete()* or *exists()* must be a hash element, such as

$foo{$bar} $ref->[12]->{"susie"}

%s did not return a true value

(F) A required (or used) file must return a true value to indicate that it compiled correctly and ran its initialization code correctly. It's traditional to end such a file with a "1;", though any true value would do. See *require* .

%s found where operator expected

(S) The Perl lexer knows whether to expect a term or an operator. If it sees what it knows to be a term when it was expecting to see an operator, it gives you this warning. Usually it indicates that an operator or delimiter was omitted, such as a semicolon.

%s had compilation errors.

(F) The final summary message when a **perl -c** fails.

%s has too many errors.

(F) The parser has given up trying to parse the program after 10 errors. Further error messages would likely be uninformative.

%s matches null string many times

(W) The pattern you've specified would be an infinite loop if the regular expression engine didn't specifically check for that. See the *perlre* manpage .

%s never introduced

(S) The symbol in question was declared but somehow went out of scope before it could possibly have been used.

%s syntax OK

(F) The final summary message when a **perl -c** succeeds.

-P not allowed for setuid/setgid script

(F) The script would have to be opened by the C preprocessor by name, which provides a race condition that breaks security.

-T and -B not implemented on filehandles

(F) Perl can't peek at the stdio buffer of filehandles when it doesn't know about your kind of stdio. You'll have to use a filename instead.

?+* follows nothing in regexp

(F) You started a regular expression with a quantifier. Backslash it if you meant it literally. See the *perlre* manpage .

@ outside of string

(F) You had a pack template that specified an absolution position outside the string being unpacked. See *pack* .

accept() on closed fd

(W) You tried to do an accept on a closed socket. Did you forget to check the return value of your *socket()* call? See *accept* .

Allocation too large: %lx

(F) You can't allocate more than 64K on an MSDOS machine.

Arg too short for msgsnd

(F) *msgsnd()* requires a string at least as long as sizeof(long).

Ambiguous use of %s resolved as %s

(W)(S) You said something that may not be interpreted the way you thought. Normally it's pretty easy to disambiguate it by supplying a missing quote, operator, paren pair or declaration.

Args must match #! line

(F) The setuid emulator requires that the arguments Perl was invoked with match the arguments specified on the #! line.

Argument " %s " isn't numeric

(W) The indicated string was fed as an argument to an operator that expected a numeric value instead. If you're fortunate the message will identify which operator was so unfortunate.

Array @% s missing the @ in argument %d of %< EM> s()

(D) Really old Perl let you omit the **@** on array names in some spots. This is now heavily deprecated.

assertion botched: %s

(P) The malloc package that comes with Perl had an internal failure.

Assertion failed: file " %s "

(P) A general assertion failed. The file in question must be examined.

Assignment to both a list and a scalar

(F) If you assign to a conditional operator, the 2nd and 3rd arguments must either both be scalars or both be lists. Otherwise Perl won't know which context to supply to the right side.

Attempt to free non-arena SV: 0x %lx

(P) All SV objects are supposed to be allocated from arenas that will be garbage collected on exit. An SV was discovered to be outside any of those arenas.

Attempt to free temp prematurely

(W) Mortalized values are supposed to be freed by the *free_tmps()* routine. This indicates that something else is freeing the SV before the *free_tmps()* routine gets a chance, which means that the *free_tmps()* routine will be freeing an unreferenced scalar when it does try to free it.

Attempt to free unreferenced glob pointers

(P) The reference counts got screwed up on symbol aliases.

Attempt to free unreferenced scalar

(W) Perl went to decrement the reference count of a scalar to see if it would go to 0, and discovered that it had already gone to 0 earlier, and should have been freed, and in fact, probably was freed. This could indicate that *SvREFCNT_dec()* was called too many times, or that *SvREFCNT_inc()* was called too few times, or that the SV was mortalized when it shouldn't have been, or that memory has been corrupted.

Bad arg length for %s , is %d , should be %d

(F) You passed a buffer of the wrong size to one of *msgctl()* , *semctl()* or *shmctl()* . In C parlance, the correct sized are, respectively, S

Bad associative array

(P) One of the internal hash routines was passed a null HV pointer.

Bad filehandle: %s

(F) A symbol was passed to something wanting a filehandle, but the symbol has no filehandle associated with it. Perhaps you didn't do an *open()* , or did it in another package.

Bad *free()* ignored

(S) An internal routine called *free()* on something that had never been *malloc()* ed in the first place.

Bad name after %s::

(F) You started to name a symbol by using a package prefix, and then didn't finish the symbol. In particular, you can't interpolate outside of quotes, so

$var = 'myvar'; $sym = mypack::$var;

is not the same as

$var = 'myvar'; $sym = "mypack::$var";

Bad symbol for array

(P) An internal request asked to add an array entry to something that wasn't a symbol table entry.

Bad symbol for filehandle

(P) An internal request asked to add a filehandle entry to something that wasn't a symbol table entry.

Bad symbol for hash

(P) An internal request asked to add a hash entry to something that wasn't a symbol table entry.

BEGIN failed--compilation aborted

(F) An untrapped exception was raised while executing a BEGIN subroutine. Compilation stops immediately and the interpreter is exited.

bind() on closed fd

(W) You tried to do a bind on a closed socket. Did you forget to check the return value of your *socket()* call? See *bind* .

Callback called exit

(F) A subroutine invoked from an external package via *perl_call_sv()* exited by calling exit.

Can't "last" outside a block

(F) A "last" statement was executed to break out of the current block, except that there's this itty bitty problem called there isn't a current block. Note that an "if" or "else" block doesn't count as a "loopish" block. You can usually double the curlies to get the same effect though, since the inner curlies will be considered a block that loops once. See *last* .

Can't "next" outside a block

(F) A "next" statement was executed to reiterate the current block, but there isn't a current block. Note that an "if" or "else" block doesn't count as a "loopish" block. You can usually double the curlies to get the same effect though, since the inner curlies will be considered a block that loops once. See *last* .

Can't "redo" outside a block

(F) A "redo" statement was executed to restart the current block, but there isn't a current block. Note that an "if" or "else" block doesn't count as a "loopish" block. You can usually double the curlies to get the same effect though, since the inner curlies will be considered a block that loops once. See *last* .

Can't bless non-reference value

(F) Only hard references may be blessed. This is how Perl "enforces" encapsulation of objects. See the *perlobj* manpage .

Can't break at that line

(S) A warning intended for while running within the debugger, indicating the line number specified wasn't the location of a statement that could be stopped at.

Can't call method " %s " in empty package " %s "

(F) You called a method correctly, and it correctly indicated a package functioning as a class, but that package doesn't have ANYTHING defined in it, let alone methods. See the *perlobj* manpage .

Can't call method " %s " on unblessed reference

(F) A method call must know what package it's supposed to run in. It ordinarily finds this out from the object reference you supply, but you didn't supply an object reference in this case. A reference isn't an object reference until it has been blessed. See the *perlobj* manpage .

Can't call method " %s " without a package or object reference

(F) You used the syntax of a method call, but the slot filled by the object reference or package name contains an expression that returns neither an object reference nor a package name. (Perhaps it's null?) Something like this will reproduce the error:

$BADREF = undef; process $BADREF 1,2,3; $BADREF->process(1,2,3);

Can't chdir to %s

(F) You called **perl -x/foo/bar** , but **/foo/bar** is not a directory that you can chdir to, possibly because it doesn't exist.

Can't coerce %s to integer in %s

(F) Certain types of SVs, in particular real symbol table entries (type GLOB), can't be forced to stop being what they are. So you can't say things like:

*foo += 1;

You CAN say

$foo = *foo; $foo += 1;

but then **$foo** no longer contains a glob.

Can't coerce %s to number in %s

(F) Certain types of SVs, in particular real symbol table entries (type GLOB), can't be forced to stop being what they are.

Can't coerce %s to string in %s

(F) Certain types of SVs, in particular real symbol table entries (type GLOB), can't be forced to

stop being what they are.

Can't create pipe mailbox

(P) An error peculiar to VMS. The process is suffering from exhausted quotas or other plumbing problems.

Can't declare %s in my

(F) Only scalar, array and hash variables may be declared as lexical variables. They must have ordinary identifiers as names.

Can't do inplace edit on %s: %s

(S) The creation of the new file failed for the indicated reason.

Can't do inplace edit without backup

(F) You're on a system such as MSDOS that gets confused if you try reading from a deleted (but still opened) file. You have to say **-i .bak** , or some such.

Can't do inplace edit: %s > 14 characters

(S) There isn't enough room in the filename to make a backup name for the file.

Can't do inplace edit: %s is not a regular file

(S) You tried to use the **-i** switch on a special file, such as a file in /dev, or a FIFO. The file was ignored.

Can't do setegid!

(P) The setegid() call failed for some reason in the setuid emulator of suidperl.

Can't do seteuid!

(P) The setuid emulator of suidperl failed for some reason.

Can't do setuid

(F) This typically means that ordinary perl tried to exec suidperl to do setuid emulation, but couldn't exec it. It looks for a name of the form sperl5.000 in the same directory that the perl executable resides under the name perl5.000, typically /usr/local/bin on Unix machines. If the file is there, check the execute permissions. If it isn't, ask your sysadmin why he and/or she removed it.

Can't do waitpid with flags

(F) This machine doesn't have either *waitpid()* or *wait4()* , so only *waitpid()* without flags is emulated.

Can't do {n,m} with n > m

(F) Minima must be less than or equal to maxima. If you really want your regexp to match something 0 times, just put {0}. See the *perlre* manpage .

Can't emulate - %s on #! line

(F) The #! line specifies a switch that doesn't make sense at this point. For example, it'd be kind of silly to put a **-x** on the #! line.

Can't exec " %s ": %s

(W) An *system()* , *exec()* or piped open call could not execute the named program for the indicated reason. Typical reasons include: the permissions were wrong on the file, the file wasn't found in **$ENV {PATH}** , the executable in question was compiled for another architecture, or the #! line in

a script points to an interpreter that can't be run for similar reasons. (Or maybe your system doesn't support #! at all.)

Can't exec %s

(F) Perl was trying to execute the indicated program for you because that's what the #! line said. If that's not what you wanted, you may need to mention "perl" on the #! line somewhere.

Can't execute %s

(F) You used the **-S** switch, but the script to execute could not be found in the PATH, or at least not with the correct permissions.

Can't find label %s

(F) You said to goto a label that isn't mentioned anywhere that it's possible for us to go to. See *goto* .

Can't find string terminator %s anywhere before EOF

(F) Perl strings can stretch over multiple lines. This message means that the closing delimiter was omitted. Since bracketed quotes count nesting levels, the following is missing its final parenthesis:

print q(The character '(' starts a side comment.)

Can't fork

(F) A fatal error occurred while trying to fork while opening a pipeline.

Can't get filespec - stale stat buffer?

(S) A warning peculiar to VMS. This arises because of the difference between access checks under VMS and under the Unix model Perl assumes. Under VMS, access checks are done by filename, rather than by bits in the stat buffer, so that ACLs and other protections can be taken into account. Unfortunately, Perl assumes that the stat buffer contains all the necessary information, and passes it, instead of the filespec, to the access checking routine. It will try to retrieve the filespec using the device name and FID present in the stat buffer, but this works only if you haven't made a subsequent call to the CRTL *stat()* routine, since the device name is overwritten with each call. If this warning appears, the name lookup failed, and the access checking routine gave up and returned FALSE, just to be conservative. (Note: The access checking routine knows about the Perl **stat** operator and file tests, so you shouldn't ever see this warning in response to a Perl command; it arises only if some internal code takes stat buffers lightly.)

Can't get pipe mailbox device name

(P) An error peculiar to VMS. After creating a mailbox to act as a pipe, Perl can't retrieve its name for later use.

Can't get SYSGEN parameter value for MAXBUF

(P) An error peculiar to VMS. Perl asked **$GETSYI** how big you want your mailbox buffers to be, and didn't get an answer.

Can't goto subroutine outside a subroutine

(F) The deeply magical "goto subroutine" call can only replace one subroutine call for another. It can't manufacture one out of whole cloth. In general you should only be calling it out of an AUTOLOAD routine anyway. See *goto* .

Can't localize lexical variable %s

(F) You used local on a variable name that was previous declared as a lexical variable using "r This is not allowed. If you want to localize a package variable of the same name, qualify it wit the package name.

Can't locate %s in @INC

(F) You said to do (or require, or use) a file that couldn't be found in any of the libraries mentioned in **@INC** . Perhaps you need to set the PERL5LIB environment variable to say where the extra library is, or maybe the script needs to add the library name to **@INC** . Or maybe you just misspelled the name of the file. See *require* .

Can't locate object method " %s " via package " %s "

(F) You called a method correctly, and it correctly indicated a package functioning as a class, but that package doesn't define that particular method, nor does any of it's base classes. See the *perlobj* manpage .

Can't locate package %s for @% s::ISA

(W) The **@ISA** array contained the name of another package that doesn't seem to exist.

Can't *mktemp()*

(F) The *mktemp()* routine failed for some reason while trying to process a **-e** switch. Maybe your /tmp partition is full, or clobbered.

Can't modify %s in %s

(F) You aren't allowed to assign to the item indicated, or otherwise try to change it, such as with an autoincrement.

Can't modify non-existent substring

(P) The internal routine that does assignment to a substr() was handed a NULL.

Can't msgrcv to readonly var

(F) The target of a msgrcv must be modifiable in order to be used as a receive buffer.

Can't open %s: %s

(S) An inplace edit couldn't open the original file for the indicated reason. Usually this is because you don't have read permission for the file.

Can't open bidirectional pipe

(W) You tried to say **open(CMD, "|cmd|")** , which is not supported. You can try any of several modules in the Perl library to do this, such as "open2.pl". Alternately, direct the pipe's output to a file using ">", and then read it in under a different file handle.

Can't open error file %s as stderr

(F) An error peculiar to VMS. Perl does its own command line redirection, and couldn't open the file specified after '2>' or '>>' on the command line for writing.

Can't open input file %s as stdin

(F) An error peculiar to VMS. Perl does its own command line redirection, and couldn't open the file specified after <' on the command line for reading.

Can't open output file %s as stdout

(F) An error peculiar to VMS. Perl does its own command line redirection, and couldn't open the

file specified after '>' or >>' on the command line for writing.

Can't open output pipe (name: %s)

(P) An error peculiar to VMS. Perl does its own command line redirection, and couldn't open the pipe into which to send data destined for stdout.

Can't open perl script " %s ": %s

(F) The script you specified can't be opened for the indicated reason.

Can't rename %s to %s: %s , skipping file

(S) The rename done by the **-i** switch failed for some reason, probably because you don't have write permission to the directory.

Can't reopen input pipe (name: %s) in binary mode

(P) An error peculiar to VMS. Perl thought stdin was a pipe, and tried to reopen it to accept binary data. Alas, it failed.

Can't reswap uid and euid

(P) The *setreuid()* call failed for some reason in the setuid emulator of suidperl.

Can't return outside a subroutine

(F) The return statement was executed in mainline code, that is, where there was no subroutine call to return out of. See the *perlsub* manpage .

Can't stat script " %s "

(P) For some reason you can't *fstat()* the script even though you have it open already. Bizarre.

Can't swap uid and euid

(P) The *setreuid()* call failed for some reason in the setuid emulator of suidperl.

Can't take log of %g

(F) Logarithms are only defined on positive real numbers.

Can't take sqrt of %g

(F) For ordinary real numbers, you can't take the square root of a negative number. There's a Complex package available for Perl, though, if you really want to do that.

Can't undef active subroutine

(F) You can't undefine a routine that's currently running. You can, however, redefine it while it's running, and you can even undef the redefined subroutine while the old routine is running. Go figure.

Can't unshift

(F) You tried to unshift an "unreal" array that can't be unshifted, such as the main Perl stack.

Can't upgrade that kind of scalar

(P) The internal sv_upgrade routine adds "members" to an SV, making it into a more specialized kind of SV. The top several SV types are so specialized, however, that they cannot be interconverted. This message indicates that such a conversion was attempted.

Can't upgrade to undef

(P) The undefined SV is the bottom of the totem pole, in the scheme of upgradability. Upgrading to undef indicates an error in the code calling sv_upgrade.

Can't use %s for loop variable

(F) Only a simple scalar variable may be used as a loop variable on a foreach.

Can't use %s ref as %s ref

(F) You've mixed up your reference types. You have to dereference a reference of the type needed. You can use the *ref()* function to test the type of the reference, if need be.

Can't use \1 to mean $1 in expression

(W) In an ordinary expression, backslash is a unary operator that creates a reference to its argument. The use of backslash to indicate a backreference to a matched substring is only valid as part of a regular expression pattern. Trying to do this in ordinary Perl code produces a value that prints out looking like SCALAR(0xdecaf). Use the **$1** form instead.

Can't use string (" %s ") as %s ref while "strict refs" in use

(F) Only hard references are allowed by "strict refs". Symbolic references are disallowed. See the *perlref* manpage .

Can't use an undefined value as %s reference

(F) A value used as either a hard reference or a symbolic reference must be a defined value. This helps to de-lurk some insidious errors.

Can't use delimiter brackets within expression

(F) The **${** name} construct is for disambiguating identifiers in strings, not in ordinary code.

Can't use global %s in "my"

(F) You tried to declare a magical variable as a lexical variable. This is not allowed, because the magic can only be tied to one location (namely the global variable) and it would be incredibly confusing to have variables in your program that looked like magical variables but weren't.

Can't use subscript on %s

(F) The compiler tried to interpret a bracketed expression as a subscript. But to the left of the brackets was an expression that didn't look like an array reference, or anything else subscriptable.

Can't write to temp file for -e : %s

(F) The write routine failed for some reason while trying to process a **-e** switch. Maybe your /tmp partition is full, or clobbered.

Can't x= to readonly value

(F) You tried to repeat a constant value (often the undefined value) with an assignment operator, which implies modifying the value itself. Perhaps you need to copy the value to a temporary, and repeat that.

Cannot open temporary file

(F) The create routine failed for some reaon while trying to process a **-e** switch. Maybe your /tmp partition is full, or clobbered.

chmod: mode argument is missing initial 0

(W) A novice will sometimes say

chmod 777, $filename

not realizing that 777 will be interpreted as a decimal number, equivalent to 01411. Octal constants

are introduced with a leading 0 in Perl, as in C.

Close on unopened file< %s >

(W) You tried to close a filehandle that was never opened.

connect() on closed fd

(W) You tried to do a connect on a closed socket. Did you forget to check the return value of your *socket()* call? See *connect* .

Corrupt malloc ptr 0x %lx at 0x %lx

(P) The malloc package that comes with Perl had an internal failure.

corrupted regexp pointers

(P) The regular expression engine got confused by what the regular expression compiler gave it.

corrupted regexp program

(P) The regular expression engine got passed a regexp program without a valid magic number.

Deep recursion on subroutine " %s "

(W) This subroutine has called itself (directly or indirectly) 100 times than it has returned. This probably indicates an infinite recursion, unless you're writing strange benchmark programs, in which case it indicates something else.

Did you mean $ or @ instead of %?

(W) You probably said **%hash** { **$key** } when you meant **$hash** { **$key** } or **@hash** { **@keys** }. On the other hand, maybe you just meant **%hash** and got carried away.

Do you need to predeclare %s ?

(S) This is an educated guess made in conjunction with the message " %s found where operator expected". It often means a subroutine or module name is being referenced that hasn't been declared yet. This may be because of ordering problems in your file, or because of a missing "sub", "package", "require", or "use" statement. If you're referencing something that isn't defined yet, you don't actually have to define the subroutine or package before the current location. You can use an empty "sub foo;" or "package FOO;" to enter a "forward" declaration.

Don't know how to handle magic of type ' %s '

(P) The internal handling of magical variables has been cursed.

do_study: out of memory

(P) This should have been caught by *safemalloc()* instead.

Duplicate *free()* ignored

(S) An internal routine called *free()* on something that had already been freed.

END failed--cleanup aborted

(F) An untrapped exception was raised while executing an END subroutine. The interpreter is immediately exited.

Error converting file specification %s

(F) An error peculiar to VMS. Since Perl may have to deal with file specifications in either VMS or Unix syntax, it converts them to a single form when it must operate on them directly. Either you've passed an invalid file specification to Perl, or you've found a case the conversion routines

don't handle. Drat.

Execution of %s aborted due to compilation errors.

(F) The final summary message when a Perl compilation fails.

Exiting eval via %s

(W) You are exiting an eval by unconventional means, such as a a goto, or a loop control statement.

Exiting subroutine via %s

(W) You are exiting a subroutine by unconventional means, such as a a goto, or a loop control statement.

Exiting substitution via %s

(W) You are exiting a substitution by unconventional means, such as a a return, a goto, or a loop control statement.

Fatal VMS error at %s , line %d

(P) An error peculiar to VMS. Something untoward happened in a VMS system service or RTL routine; Perl's exit status should provide more details. The filename in "at %s " and the line number in "line **%d** " tell you which section of the Perl source code is distressed.

fcntl is not implemented

(F) Your machine apparently doesn't implement fcntl() . What is this, a PDP-11 or something?

Filehandle %s never opened

(W) An I/O operation was attempted on a filehandle that was never initialized. You need to do an *open()* or a *socket()* call, or call a constructor from the FileHandle package.

Filehandle %s opened only for input

(W) You tried to write on a read-only filehandle. If you intended it to be a read-write filehandle, you needed to open it with "<" or "+>" or ">>" instead of with <" or nothing. If you only intended to write the file, use ">" or >>". See *open* .

Filehandle only opened for input

(W) You tried to write on a read-only filehandle. If you intended it to be a read-write filehandle, you needed to open it with "<" or "+>" or ">>" instead of with <" or nothing. If you only intended to write the file, use ">" or >>". See *open* .

Final $ should be \ $ or $name

(F) You must now decide whether the final **$** in a string was meant to be a literal dollar sign, or was meant to introduce a variable name that happens to be missing. So you have to put either the backslash or the name.

Final @ should be \ @ or @name

(F) You must now decide whether the final **@** in a string was meant to be a literal "at" sign, or was meant to introduce a variable name that happens to be missing. So you have to put either the backslash or the name.

Format %s redefined

(W) You redefined a format. To suppress this warning, say

{ local $^W = 0; eval "format NAME =..."; }

Format not terminated

(F) A format must be terminated by a line with a solitary dot. Perl got to the end of your file without finding such a line.

Found = in conditional, should be ==

(W) You said

if ($foo = 123)

when you meant

if ($foo == 123)

(or something like that).

gdbm store returned %d , errno %d , key " %s "

(S) A warning from the GDBM_File extension that a store failed.

gethostent not implemented

(F) Your C library apparently doesn't implement gethostent() , probably because if it did, it'd feel morally obligated to return every hostname on the Internet.

get{sock,peer} *name()* on closed fd

(W) You tried to get a socket or peer socket name on a closed socket. Did you forget to check the return value of your *socket()* call?

getpwnam returned invalid UIC %# o for user " %s "

(S) A warning peculiar to VMS. The call to **sys $getuai** underlying the getpwnam operator returned an invalid UIC.

Glob not terminated

(F) The lexer saw a left angle bracket in a place where it was expecting a term, so it's looking for the corresponding right angle bracket, and not finding it. Chances are you left some needed parentheses out earlier in the line, and you really meant a "less than".

Global symbol " %s " requires explicit package name

(F) You've said "use strict vars", which indicates that all variables must either be lexically scoped (using "my"), or explicitly qualified to say which package the global variable is in (using "::").

goto must have label

(F) Unlike with "next" or "last", you're not allowed to goto an unspecified destination. See *goto* .

Had to create %s unexpectedly

(S) A routine asked for a symbol from a symbol table that ought to have existed already, but for some reason it didn't, and had to be created on an emergency basis to prevent a core dump.

Hash %% s missing the % in argument %d of %< EM> s()

(D) Really old Perl let you omit the **%** on hash names in some spots. This is now heavily deprecated.

Identifier " %s:: %s " used only once: possible typo

(W) Typographical errors often show up as unique identifiers. If you had a good reason for ha a unique identifier, then just mention it again somehow to suppress the message.

Illegal division by zero

(F) You tried to divide a number by 0. Either something was wrong in your logic, or you need to put a conditional in to guard against meaningless input.

Illegal modulus zero

(F) You tried to divide a number by 0 to get the remainder. Most numbers don't take to this kindly.

Illegal octal digit

(F) You used an 8 or 9 in a octal number.

Illegal octal digit ignored

(W) You may have tried to use an 8 or 9 in a octal number. Interpretation of the octal number stopped before the 8 or 9.

Insecure dependency in %s

(F) You tried to do something that the tainting mechanism didn't like. The tainting mechanism is turned on when you're running setuid or setgid, or when you specify -T to turn it on explicitly. The tainting mechanism labels all data that's derived directly or indirectly from the user, who is considered to be unworthy of your trust. If any such data is used in a "dangerous" operation, you get this error. See the *perlsec* manpage for more information.

Insecure directory in %s

(F) You can't use *system()* , *exec()* , or a piped open in a setuid or setgid script if **$ENV** {PATH} contains a directory that is writable by the world. See the *perlsec* manpage .

Insecure PATH

(F) You can't use *system()* , *exec()* , or a piped open in a setuid or setgid script if **$ENV** {PATH} is derived from data supplied (or potentially supplied) by the user. The script must set the path to a known value, using trustworthy data. See the *perlsec* manpage .

Internal inconsistency in tracking vforks

(S) A warning peculiar to VMS. Perl keeps track of the number of times you've called **fork** and **exec** , in order to determine whether the current call to **exec** should be affect the current script or a subprocess (see *exec*). Somehow, this count has become scrambled, so Perl is making a guess and treating this **exec** as a request to terminate the Perl script and execute the specified command.

internal disaster in regexp

(P) Something went badly wrong in the regular expression parser.

internal urp in regexp at / %s /

(P) Something went badly awry in the regular expression parser.

invalid [] range in regexp

(F) The range specified in a character class had a minimum character greater than the maximum character. See the *perlre* manpage .

ioctl is not implemented

(F) Your machine apparently doesn't implement ioctl() , which is pretty strange for a machine that

supports C.

junk on end of regexp

(P) The regular expression parser is confused.

Label not found for "last %s "

(F) You named a loop to break out of, but you're not currently in a loop of that name, not even if you count where you were called from. See *last* .

Label not found for "next %s "

(F) You named a loop to continue, but you're not currently in a loop of that name, not even if you count where you were called from. See *last* .

Label not found for "redo %s "

(F) You named a loop to restart, but you're not currently in a loop of that name, not even if you count where you were called from. See *last* .

listen() on closed fd

(W) You tried to do a listen on a closed socket. Did you forget to check the return value of your *socket()* call? See *listen* .

Literal @% s now requires backslash

(F) It used to be that Perl would try to guess whether you wanted an array interpolated or a literal @. It did this when the string was first used at runtime. Now strings are parsed at compile time, and ambiguous instances of **@** must be disambiguated, either by putting a backslash to indicate a literal, or by declaring (or using) the array within the program before the string (lexically). (Someday it will simply assume that an unbackslashed **@** interpolates an array.)

Method for operation %s not found in package %s during blessing

(F) An attempt was made to specify an entry in an overloading table that doesn't somehow point to a valid method. See the *perlovl* manpage .

Might be a runaway multi-line %s string starting on line %d

(S) An advisory indicating that the previous error may have been caused by a missing delimiter on a string or pattern, because it eventually ended earlier on the current line.

Misplaced _ in number

(W) An underline in a decimal constant wasn't on a 3-digit boundary.

Missing $ on loop variable

(F) Apparently you've been programming in csh too much. Variables are always mentioned with the **$** in Perl, unlike in the shells, where it can vary from one line to the next.

Missing comma after first argument to %s function

(F) While certain functions allow you to specify a filehandle or an "indirect object" before the argument list, this ain't one of them.

Missing operator before %s ?

(S) This is an educated guess made in conjunction with the message " %s found where operator expected". Often the missing operator is a comma.

Missing right bracket

(F) The lexer counted more opening curly brackets (braces) than closing ones. As a general rul you'll find it's missing near the place you were last editing.

Missing semicolon on previous line?

(S) This is an educated guess made in conjunction with the message " %s found where operator expected". Don't automatically put a semicolon on the previous line just because you saw this message.

Modification of a read-only value attempted

(F) You tried, directly or indirectly, to change the value of a constant. You didn't, of course, try "2 = 1", since the compiler catches that. But an easy way to do the same thing is:

sub mod { $_[0] = 1 } mod(2);

Another way is to assign to a substr() that's off the end of the string.

Modification of non-creatable array value attempted, subscript %d

(F) You tried to make an array value spring into existence, and the subscript was probably negative, even counting from end of the array backwards.

Modification of non-creatable hash value attempted, subscript " %s "

(F) You tried to make a hash value spring into existence, and it couldn't be created for some peculiar reason.

Module name must be constant

(F) Only a bare module name is allowed as the first argument to a "use".

msg %s not implemented

(F) You don't have System V message IPC on your system.

Multidimensional syntax %s not supported

(W) Multidimensional arrays aren't written like **$foo** [1,2,3]. They're written like **$foo** [1][2][3], as in C.

Negative length

(F) You tried to do a read/write/send/recv operation with a buffer length that is less than 0. This is difficult to imagine.

nested *?+ in regexp

(F) You can't quantify a quantifier without intervening parens. So things like ** or +* or ?* are illegal.

Note, however, that the minimal matching quantifiers, *?, +? and ?? appear to be nested quantifiers, but aren't. See the *perlre* manpage .

No #! line

(F) The setuid emulator requires that scripts have a well-formed #! line even on machines that don't support the #! construct.

No %s allowed while running setuid

(F) Certain operations are deemed to be too insecure for a setuid or setgid script to even be allowed to attempt. Generally speaking there will be another way to do what you want that is, if not secure,

at least securable. See the *perlsec* manpage .

No -e allowed in setuid scripts

(F) A setuid script can't be specified by the user.

No comma allowed after %s

(F) A list operator that has a filehandle or "indirect object" is not allowed to have a comma between that and the following arguments. Otherwise it'd be just another one of the arguments.

No command into which to pipe on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a '|' at the end of the command line, so it doesn't know whither you want to pipe the output from this command.

No DB::DB routine defined

(F) The currently executing code was compiled with the **-d** switch, but for some reason the perl5db.pl file (or some facsimile thereof) didn't define a routine to be called at the beginning of each statement. Which is odd, because the file should have been required automatically, and should have blown up the require if it didn't parse right.

No dbm on this machine

(P) This is counted as an internal error, because every machine should supply dbm nowadays, since Perl comes with SDBM. See *SDBM_File* .

No DBsub routine

(F) The currently executing code was compiled with the **-d** switch, but for some reason the perl5db.pl file (or some facsimile thereof) didn't define a DB::sub routine to be called at the beginning of each ordinary subroutine call.

No error file after 2> or >> on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a '2>' or a '>>' on the command line, but can't find the name of the file to which to write data destined for stderr.

No input file after< on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a <' on the command line, but can't find the name of the file from which to read data for stdin.

No output file after > on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a lone '>' at the end of the command line, so it doesn't know whither you wanted to redirect stdout.

No output file after > or>> on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a '>' or a >>' on the command line, but can't find the name of the file to which to write data destined for stdout.

No Perl script found in input

(F) You called **perl -x** , but no line was found in the file beginning with #! and containing the word "perl".

No setregid available

(F) Configure didn't find anything resembling the *setregid()* call for your system.

No setreuid available

(F) Configure didn't find anything resembling the *setreuid()* call for your system.

No space allowed after -I

(F) The argument to **-I** must follow the **-I** immediately with no intervening space.

No such pipe open

(P) An error peculiar to VMS. The internal routine *my_pclose()* tried to close a pipe which hadn't been opened. This should have been caught earlier as an attempt to close an unopened filehandle.

No such signal: SIG %s

(W) You specified a signal name as a subscript to **%SIG** that was not recognized. Say **kill -l** in your shell to see the valid signal names on your system.

Not a CODE reference

(F) Perl was trying to evaluate a reference to a code value (that is, a subroutine), but found a reference to something else instead. You can use the *ref()* function to find out what kind of ref it really was. See also the *perlref* manpage .

Not a format reference

(F) I'm not sure how you managed to generate a reference to an anonymous format, but this indicates you did, and that it didn't exist.

Not a GLOB reference

(F) Perl was trying to evaluate a reference to a "type glob" (that is, a symbol table entry that looks like ***foo**), but found a reference to something else instead. You can use the *ref()* function to find out what kind of ref it really was. See the *perlref* manpage .

Not a HASH reference

(F) Perl was trying to evaluate a reference to a hash value, but found a reference to something else instead. You can use the *ref()* function to find out what kind of ref it really was. See the *perlref* manpage .

Not a perl script

(F) The setuid emulator requires that scripts have a well-formed #! line even on machines that don't support the #! construct. The line must mention perl.

Not a SCALAR reference

(F) Perl was trying to evaluate a reference to a scalar value, but found a reference to something else instead. You can use the *ref()* function to find out what kind of ref it really was. See the *perlref* manpage .

Not a subroutine reference

(F) Perl was trying to evaluate a reference to a code value (that is, a subroutine), but found a reference to something else instead. You can use the *ref()* function to find out what kind of ref it really was. See also the *perlref* manpage .

Not a subroutine reference in %OVERLOAD

(F) An attempt was made to specify an entry in an overloading table that doesn't somehow point to

a valid subroutine. See the *perlovl* manpage .

Not an ARRAY reference

(F) Perl was trying to evaluate a reference to an array value, but found a reference to something else instead. You can use the *ref()* function to find out what kind of ref it really was. See the *perlref* manpage .

Not enough arguments for %s

(F) The function requires more arguments than you specified.

Not enough format arguments

(W) A format specified more picture fields than the next line supplied. See the *perlform* manpage .

Null filename used

(F) You can't require the null filename, especially since on many machines that means the current directory! See *require* .

NULL OP IN RUN

(P) Some internal routine called *run()* with a null opcode pointer.

Null realloc

(P) An attempt was made to realloc NULL.

NULL regexp argument

(P) The internal pattern matching routines blew it bigtime.

NULL regexp parameter

(P) The internal pattern matching routines are out of their gourd.

Odd number of elements in hash list

(S) You specified an odd number of elements to a hash list, which is odd, since hash lists come in key/value pairs.

oops: oopsAV

(S) An internal warning that the grammar is screwed up.

oops: oopsHV

(S) An internal warning that the grammar is screwed up.

Operation ` %s ' %s: no method found,

(F) An attempt was made to use an entry in an overloading table that somehow no longer points to a valid method. See the *perlovl* manpage .

Operator or semicolon missing before %s

(S) You used a variable or subroutine call where the parser was expecting an operator. The parser has assumed you really meant to use an operator, but this is highly likely to be incorrect. For example, if you say "*foo *foo" it will be interpreted as if you said "*foo * 'foo'".

Out of memory for yacc stack

(F) The yacc parser wanted to grow its stack so it could continue parsing, but *realloc()* wouldn't give it more memory, virtual or otherwise.

Out of memory!

(X) The *malloc()* function returned 0, indicating there was insufficient remaining memory (or virtual memory) to satisfy the request.

page overflow

(W) A single call to *write()* produced more lines than can fit on a page. See the *perlform* manpage .

panic: ck_grep

(P) Failed an internal consistency check trying to compile a grep.

panic: ck_split

(P) Failed an internal consistency check trying to compile a split.

panic: corrupt saved stack index

(P) The savestack was requested to restore more localized values than there are in the savestack.

panic: die %s

(P) We popped the context stack to an eval context, and then discovered it wasn't an eval context.

panic: do_match

(P) The internal *pp_match()* routine was called with invalid operational data.

panic: do_split

(P) Something terrible went wrong in setting up for the split.

panic: do_subst

(P) The internal *pp_subst()* routine was called with invalid operational data.

panic: do_trans

(P) The internal *do_trans()* routine was called with invalid operational data.

panic: goto

(P) We popped the context stack to a context with the specified label, and then discovered it wasn't a context we know how to do a goto in.

panic: INTERPCASEMOD

(P) The lexer got into a bad state at a case modifier.

panic: INTERPCONCAT

(P) The lexer got into a bad state parsing a string with brackets.

panic: last

(P) We popped the context stack to a block context, and then discovered it wasn't a block context.

panic: leave_scope clearsv

(P) A writable lexical variable became readonly somehow within the scope.

panic: leave_scope inconsistency

(P) The savestack probably got out of sync. At least, there was an invalid enum on the top of it.

panic: malloc

(P) Something requested a negative number of bytes of malloc.

panic: mapstart

(P) The compiler is screwed up with respect to the *map()* function.

panic: null array

(P) One of the internal array routines was passed a null AV pointer.

panic: pad_alloc

(P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_free curpad

(P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_free po

(P) An invalid scratch pad offset was detected internally.

panic: pad_reset curpad

(P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_sv po

(P) An invalid scratch pad offset was detected internally.

panic: pad_swipe curpad

(P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_swipe po

(P) An invalid scratch pad offset was detected internally.

panic: pp_iter

(P) The foreach iterator got called in a non-loop context frame.

panic: realloc

(P) Something requested a negative number of bytes of realloc.

panic: restartop

(P) Some internal routine requested a goto (or something like it), and didn't supply the destination.

panic: return

(P) We popped the context stack to a subroutine or eval context, and then discovered it wasn't a subroutine or eval context.

panic: scan_num

(P) *scan_num()* got called on something that wasn't a number.

panic: sv_insert

(P) The *sv_insert()* routine was told to remove more string than there was string.

panic: top_env

(P) The compiler attempted to do a goto, or something weird like that.

panic: yylex

(P) The lexer got into a bad state while processing a case modifier.

Parens missing around " %s " list

(W) You said something like

my $foo, $bar = @_;

when you meant

my ($foo, $bar) = @_;

Remember that "my" and "local" bind closer than comma.

Perl %3 .3f required--this is only version %s , stopped

(F) The module in question uses features of a version of Perl more recent than the currently running version. How long has it been since you upgraded, anyway? See *require* .

Permission denied

(F) The setuid emulator in suidperl decided you were up to no good.

pid %d not a child

(W) A warning peculiar to VMS. *Waitpid()* was asked to wait for a process which isn't a subprocess of the current process. While this is fine from VMS' perspective, it's probably not what you intended.

POSIX getpgrp can't take an argument

(F) Your C compiler uses POSIX *getpgrp()* , which takes no argument, unlike the BSD version, which takes a pid.

Possible memory corruption: %s overflowed 3rd argument

(F) An ioctl() or fcntl() returned more than Perl was bargaining for. Perl guesses a reasonable buffer size, but puts a sentinel byte at the end of the buffer just in case. This sentinel byte got clobbered, and Perl assumes that memory is now corrupted. See *ioctl* .

Precedence problem: open %s should be open(%s)

(S) The old irregular construct open FOO || die;

is now misinterpreted as

open(FOO || die);

because of the strict regularization of Perl 5's grammar into unary and list operators. (The old open was a little of both.) You must put parens around the filehandle, or use the new "or" operator instead of "||".

print on closed filehandle %s

(W) The filehandle you're printing on got itself closed sometime before now. Check your logic flow.

printf on closed filehandle %s

(W) The filehandle you're writing to got itself closed sometime before now. Check your logic flow.

Probable precedence problem on %s

(W) The compiler found a bare word where it expected a conditional, which often indicates that an

|| or && was parsed as part of the last argument of the previous construct, for example:

open FOO || die;

Read on closed filehandle< %s >

(W) The filehandle you're reading from got itself closed sometime before now. Check your logic flow.

Reallocation too large: %lx

(F) You can't allocate more than 64K on an MSDOS machine.

Recompile perl with -D DEBUGGING to use -D switch

(F) You can't use the **-D** option unless the code to produce the desired output is compiled into Perl, which entails some overhead, which is why it's currently left out of your copy.

Recursive inheritance detected

(F) More than 100 levels of inheritance were used. Probably indicates an unintended loop in your inheritance hierarchy.

Reference miscount in *sv_replace()*

(W) The internal *sv_replace()* function was handed a new SV with a reference count of other than 1.

regexp memory corruption

(P) The regular expression engine got confused by what the regular expression compiler gave it.

regexp out of space

(P) A "can't happen" error, because *safemalloc()* should have caught it earlier.

regexp too big

(F) The current implementation of regular expression uses shorts as address offsets within a string. Unfortunately this means that if the regular expression compiles to longer than 32767, it'll blow up. Usually when you want a regular expression this big, there is a better way to do it with multiple statements. See the *perlre* manpage .

Reversed %s = operator

(W) You wrote your assignment operator backwards. The = must always comes last, to avoid ambiguity with subsequent unary operators.

Runaway format

(F) Your format contained the ~~ repeat-until-blank sequence, but it produced 200 lines at once, and the 200th line looked exactly like the 199th line. Apparently you didn't arrange for the arguments to exhaust themselves, either by using ^ instead of @ (for scalar variables), or by shifting or popping (for array variables). See the *perlform* manpage .

Scalar value @% s[%s] better written as $% s[%s]

(W) You've used an array slice (indicated by @) to select a single value of an array. Generally it's better to ask for a scalar value (indicated by $). The difference is that **$foo** [&bar] always behaves like a scalar, both when assigning to it and when evaluating its argument, while **@foo** [&bar] behaves like a list when you assign to it, and provides a list context to its subscript, which can do weird things if you're only expecting one subscript.

On the other hand, if you were actually hoping to treat the array element as a list, you need to l
into how references work, since Perl will not magically convert between scalars and lists for y
See the *perlref* manpage .

Script is not setuid/setgid in suidperl

(F) Oddly, the suidperl program was invoked on a script with its setuid or setgid bit set. This doesn't make much sense.

Search pattern not terminated

(F) The lexer couldn't find the final delimiter of a // or m{} construct. Remember that bracketing delimiters count nesting level.

seek() on unopened file

(W) You tried to use the seek() function on a filehandle that was either never opened or has been closed since.

select not implemented

(F) This machine doesn't implement the select() system call.

sem %s not implemented

(F) You don't have System V semaphore IPC on your system.

semi-panic: attempt to dup freed string

(S) The internal *newSVsv()* routine was called to duplicate a scalar that had previously been marked as free.

Semicolon seems to be missing

(W) A nearby syntax error was probably caused by a missing semicolon, or possibly some other missing operator, such as a comma.

Send on closed socket

(W) The filehandle you're sending to got itself closed sometime before now. Check your logic flow.

Sequence (?#... not terminated

(F) A regular expression comment must be terminated by a closing parenthesis. Embedded parens aren't allowed. See the *perlre* manpage .

Sequence (? %s ...) not implemented

(F) A proposed regular expression extension has the character reserved but has not yet been written. See the *perlre* manpage .

Sequence (? %s ...) not recognized

(F) You used a regular expression extension that doesn't make sense. See the *perlre* manpage .

setegid() not implemented

(F) You tried to assign to $), and your operating system doesn't support the setegid() system call (or equivalent), or at least Configure didn't think so.

seteuid() not implemented

(F) You tried to assign to $>, and your operating system doesn't support the seteuid() system call (or equivalent), or at least Configure didn't think so.

setrgid() not implemented

(F) You tried to assign to $(, and your operating system doesn't support the setrgid() system call (or equivalent), or at least Configure didn't think so.

setruid() not implemented

(F) You tried to assign to <, and your operating system doesn't support the setruid() system call (or equivalent), or at least Configure didn't think so.

Setuid/gid script is writable by world

(F) The setuid emulator won't run a script that is writable by the world, because the world might have written on it already.

shm %s not implemented

(F) You don't have System V shared memory IPC on your system.

shutdown() on closed fd

(W) You tried to do a shutdown on a closed socket. Seems a bit superfluous.

SIG %s handler " %s " not defined.

(W) The signal handler named in **%SIG** doesn't, in fact, exist. Perhaps you put it into the wrong package?

sort is now a reserved word

(F) An ancient error message that almost nobody ever runs into anymore. But before sort was a keyword, people sometimes used it as a filehandle.

Sort subroutine didn't return a numeric value

(F) A sort comparison routine must return a number. You probably blew it by not using **<=> or C, or by not using them correctly. See *sort* .**

Sort subroutine didn't return single value

(F) A sort comparison subroutine may not return a list value with more or less than one element. See *sort* .

Split loop

(P) The split was looping infinitely. (Obviously, a split shouldn't iterate more times than there are characters of input, which is what happened.) See *split* .

Stat on unopened file< %s >

(W) You tried to use the *stat()* function (or an equivalent file test) on a filehandle that was either never opened or has been closed since.

Statement unlikely to be reached

(W) You did an *exec()* with some statement after it other than a *die()* . This is almost always an error, because *exec()* never returns unless there was a failure. You probably wanted to use *system()* instead, which does return. To suppress this warning, put the *exec()* in a block by itself.

Subroutine %s redefined

(W) You redefined a subroutine. To suppress this warning, say

{ local $^W = 0; eval "sub name { ... }"; }

Substitution loop

(P) The substitution was looping infinitely. (Obviously, a substitution shouldn't iterate more times than there are characters of input, which is what happened.) See the discussion of substitution in *Quote and Quotelike Operators* .

Substitution pattern not terminated

(F) The lexer couldn't find the interior delimiter of a s/// or s{}{} construct. Remember that bracketing delimiters count nesting level.

Substitution replacement not terminated

(F) The lexer couldn't find the final delimiter of a s/// or s{}{} construct. Remember that bracketing delimiters count nesting level.

substr outside of string

(W) You tried to reference a substr() that pointed outside of a string. That is, the absolute value of the offset was larger than the length of the string. See *substr* .

suidperl is no longer needed since...

(F) Your Perl was compiled with -D SETUID_SCRIPTS_ARE_SECURE_NOW, but a version of the setuid emulator somehow got run anyway.

syntax error

(F) Probably means you had a syntax error. Common reasons include:

A keyword is misspelled. A semicolon is missing. A comma is missing. An opening or closing parenthesis is missing. An opening or closing brace is missing. A closing quote is missing.

Often there will be another error message associated with the syntax error giving more information. (Sometimes it helps to turn on -w .) The error message itself often tells you where it was in the line when it decided to give up. Sometimes the actual error is several tokens before this, since Perl is good at understanding random input. Occasionally the line number may be misleading, and once in a blue moon the only way to figure out what's triggering the error is to call perl -c repeatedly, chopping away half the program each time to see if the error went away. Sort of the cybernetic version of S<20 questions>.

System V IPC is not implemented on this machine

(F) You tried to do something with a function beginning with "sem", "shm" or "msg". See *semctl* , for example.

Syswrite on closed filehandle

(W) The filehandle you're writing to got itself closed sometime before now. Check your logic flow.

tell() on unopened file

(W) You tried to use the tell() function on a filehandle that was either never opened or has been closed since.

Test on unopened file< %s >

(W) You tried to invoke a file test operator on a filehandle that isn't open. Check your logic.

See also -*X* .

That use of $[is unsupported

(F) Assignment to $[is now strictly circumscribed, and interpreted as a compiler directive. You may only say one of

$[= 0; $[= 1; ... local $[= 0; local $[= 1; ...

This is to prevent the problem of one module changing the array base out from under another module inadvertently. See *$[* .

The %s function is unimplemented

The function indicated isn't implemented on this architecture, according to the probings of Configure.

The *crypt()* function is unimplemented due to excessive paranoia.

(F) Configure couldn't find the *crypt()* function on your machine, probably because your vendor didn't supply it, probably because they think the U.S. Govermnment thinks it's a secret, or at least that they will continue to pretend that it is. And if you quote me on that, I will deny it.

The stat preceding -l _ wasn't an lstat

(F) It makes no sense to test the current stat buffer for symbolic linkhood if the last stat that wrote to the stat buffer already went past the symlink to get to the real file. Use an actual filename instead.

times not implemented

(F) Your version of the C library apparently doesn't do times() . I suspect you're not running on Unix.

Too few args to syscall

(F) There has to be at least one argument to *syscall()* to specify the system call to call, silly dilly.

Too many args to syscall

(F) Perl only supports a maximum of 14 args to *syscall()* .

Too many arguments for %s

(F) The function requires fewer arguments than you specified.

trailing \ in regexp

(F) The regular expression ends with an unbackslashed backslash. Backslash it. See the *perlre* manpage .

Translation pattern not terminated

(F) The lexer couldn't find the interior delimiter of a tr/// or tr[][] construct.

Translation replacement not terminated

(F) The lexer couldn't find the final delimiter of a tr/// or tr[][] construct.

truncate not implemented

(F) Your machine doesn't implement a file truncation mechanism that Configure knows

about.

Type of arg %d to %s must be %s (not %s)

(F) This function requires the argument in that position to be of a certain type. Arrays must be @NAME or @{ EXPR}. Hashes must be %NAME or %{ EXPR}. No implicit dereferencing is allowed--use the {EXPR} forms as an explicit dereference. See the *perlref* manpage .

umask: argument is missing initial 0

(W) A umask of 222 is incorrect. It should be 0222, since octal literals always start with 0 in Perl, as in C.

Unbalanced context: %d more PUSHes than POPs

(W) The exit code detected an internal inconsistency in how many execution contexts were entered and left.

Unbalanced saves: %d more saves than restores

(W) The exit code detected an internal inconsistency in how many values were temporarily localized.

Unbalanced scopes: %d more ENTERs than LEAVEs

(W) The exit code detected an internal inconsistency in how many blocks were entered and left.

Unbalanced tmps: %d more allocs than frees

(W) The exit code detected an internal inconsistency in how many mortal scalars were allocated and freed.

Undefined format " %s " called

(F) The format indicated doesn't seem to exist. Perhaps it's really in another package? See the *perlform* manpage .

Undefined sort subroutine " %s " called

(F) The sort comparison routine specified doesn't seem to exist. Perhaps it's in a different package? See *sort* .

Undefined subroutine & %s called

(F) The subroutine indicated hasn't been defined, or if it was, it has since been undefined.

Undefined subroutine called

(F) The anonymous subroutine you're trying to call hasn't been defined, or if it was, it has since been undefined.

Undefined subroutine in sort

(F) The sort comparison routine specified is declared but doesn't seem to have been defined yet. See *sort* .

unexec of %s into %s failed!

(F) The unexec() routine failed for some reason. See your local FSF representative, who probably put it there in the first place.

Unknown BYTEORDER

(F) There are no byteswapping functions for a machine with this byte order.

unmatched () in regexp

(F) Unbackslashed parentheses must always be balanced in regular expressions. If you're a vi user, the % key is valuable for finding the matching paren. See the *perlre* manpage .

Unmatched right bracket

(F) The lexer counted more closing curly brackets (braces) than opening ones, so you're probably missing an opening bracket. As a general rule, you'll find the missing one (so to speak) near the place you were last editing.

unmatched [] in regexp

(F) The brackets around a character class must match. If you wish to include a closing bracket in a character class, backslash it or put it first. See the *perlre* manpage .

Unquoted string " %s " may clash with future reserved word

(W) You used a bare word that might someday be claimed as a reserved word. It's best to put such a word in quotes, or capitalize it somehow, or insert an underbar into it. You might also declare it as a subroutine.

Unrecognized character \ %03o ignored

(S) A garbage character was found in the input, and ignored, in case it's a weird control character on an EBCDIC machine, or some such.

Unrecognized signal name " %s "

(F) You specified a signal name to the *kill()* function that was not recognized. Say kill -l in your shell to see the valid signal names on your system.

Unrecognized switch: - %s

(F) You specified an illegal option to Perl. Don't do that. (If you think you didn't do that, check the #! line to see if it's supplying the bad switch on your behalf.)

Unsuccessful %s on filename containing newline

(W) A file operation was attempted on a filename, and that operation failed, PROBABLY because the filename contained a newline, PROBABLY because you forgot to *chop()* or *chomp()* it off. See *chop* .

Unsupported directory function " %s " called

(F) Your machine doesn't support *opendir()* and *readdir()* .

Unsupported function %s

(F) This machines doesn't implement the indicated function, apparently. At least, Configure doesn't think so.

Unsupported socket function " %s " called

(F) Your machine doesn't support the Berkeley socket mechanism, or at least that's what Configure thought.

Unterminated<> operator

(F) The lexer saw a left angle bracket in a place where it was expecting a term, so it's looking for the corresponding right angle bracket, and not finding it. Chances are you left some needed parentheses out earlier in the line, and you really meant a "less than".

Use of $# is deprecated

(D) This was an ill-advised attempt to emulate a poorly defined awk feature. Use an explicit printf() or *sprintf()* instead.

Use of $* is deprecated

(D) This variable magically turned on multiline pattern matching, both for you and for any luckless subroutine that you happen to call. You should use the new //m and //s modifiers now to do that without the dangerous action-at-a-distance effects of $* .

Use of %s in printf format not supported

(F) You attempted to use a feature of printf that is accessible only from C. This usually means there's a better way to do it in Perl.

Use of %s is deprecated

(D) The construct indicated is no longer recommended for use, generally because there's a better way to do it, and also because the old way has bad side effects.

Use of implicit split to @_ is deprecated

(D) It makes a lot of work for the compiler when you clobber a subroutine's argument list, so it's better if you assign the results of a *split()* explicitly to an array (or list).

Use of uninitialized value

(W) An undefined value was used as if it were already defined. It was interpreted as a "" or a 0, but maybe it was a mistake. To suppress this warning assign an initial value to your variables.

Useless use of %s in void context

(W) You did something without a side effect in a context that does nothing with the return value, such as a statement that doesn't return a value from a block, or the left side of a scalar comma operator. Very often this points not to stupidity on your part, but a failure of Perl to parse your program the way you thought it would. For example, you'd get this if you mixed up your C precedence with Python precedence and said

$one, $two = 1, 2;

when you meant to say

($one, $two) = (1, 2);

Another common error is to use ordinary parentheses to construct a list reference when you should be using square or curly brackets, for example, if you say

$array = (1,2);

when you should have said

$array = [1,2];

The square brackets explicitly turn a list value into a scalar value, while parentheses do not. So when a parenthesized list is evaluated in a scalar context, the comma is treated like C's comma operator, which throws away the left argument, which is not what you want. See the

perlref manpage for more on this.

Warning: unable to close filehandle %s properly.

(S) The implicit _close()_ done by an _open()_ got an error indication on the close(0. This usually indicates your filesystem ran out of disk space.

Warning: Use of " %s " without parens is ambiguous

(S) You wrote a unary operator followed by something that looks like a binary operator that could also have been interpreted as a term or unary operator. For instance, if you know that the rand function has a default argument of 1.0, and you write

rand + 5;

you may THINK you wrote the same thing as

rand() + 5;

but in actual fact, you got

rand(+5);

So put in parens to say what you really mean.

Write on closed filehandle

(W) The filehandle you're writing to got itself closed sometime before now. Check your logic flow.

X outside of string

(F) You had a pack template that specified a relative position before the beginning of the string being unpacked. See _pack_ .

x outside of string

(F) You had a pack template that specified a relative position after the end of the string being unpacked. See _pack_ .

Xsub " %s " called in sort

(F) The use of an external subroutine as a sort comparison is not yet supported.

Xsub called in sort

(F) The use of an external subroutine as a sort comparison is not yet supported.

You can't use -l on a filehandle

(F) A filehandle represents an opened file, and when you opened the file it already went past any symlink you are presumably trying to look for. Use a filename instead.

YOU HAVEN'T DISABLED SET-ID SCRIPTS IN THE KERNEL YET!

(F) And you probably never will, since you probably don't have the sources to your kernel, and your vendor probably doesn't give a rip about what you want. Your best bet is to use the wrapsuid script in the eg directory to put a setuid C wrapper around your script.

You need to quote " %s "

(W) You assigned a bareword as a signal handler name. Unfortunately, you already have a subroutine of that name declared, which means that Perl 5 will try to call the subroutine

when the assignment is executed, which is probably not what you want. (If it IS what you want, put an & in front.)

[gs] *etsockopt()* on closed fd

(W) You tried to get or set a socket option on a closed socket. Did you forget to check the return value of your *socket()* call? See *getsockopt* .

\1 better written as $1

(W) Outside of patterns, backreferences live on as variables. The use of backslashes is grandfathered on the righthand side of a substitution, but stylistically it's better to use the variable form because other Perl programmers will expect it, and it works better if there are more than 9 backreferences.

'|' and <' may not both be specified on command line

(F) An error peculiar to VMS. Perl does its own command line redirection, and found that STDIN was a pipe, and that you also tried to redirect STDIN using <'. Only one STDIN stream to a customer, please.

'|' and '>' may not both be specified on command line

(F) An error peculiar to VMS. Perl does its own command line redirection, and thinks you tried to redirect stdout both to a file and into a pipe to another command. You need to choose one or the other, though nothing's stopping you from piping into a program or Perl script which 'splits' output into two streams, such as

open(OUT,">$ARGV[0]") or die "Can't write to $ARGV[0]: $!"; while (<STDIN>) { print; print OUT; } close OUT;

.

NAME

perlform - Perl formats

DESCRIPTION

Perl has a mechanism to help you generate simple reports and charts. To facilitate this, Perl helps you lay out your output page in your code in a fashion that's close to how it will look when it's printed. It can keep track of things like how many lines on a page, what page you're, when to print page headers, etc. Keywords are borrowed from FORTRAN: *format()* to declare and *write()* to execute; see their entries in the *perlfunc* manpage . Fortunately, the layout is much more legible, more like BASIC's PRINT USING statement. Think of it as a poor man's nroff(1).

Formats, like packages and subroutines, are declared rather than executed, so they may occur at any point in your program. (Usually it's best to keep them all together though.) They have their own namespace apart from all the other "types" in Perl. This means that if you have a function named "Foo", it is not the same thing as having a format named "Foo". However, the default name for the format associated with a given filehandle is the same as the name of the filehandle. Thus, the default format for STDOUT is name "STDOUT", and the default format for filehandle TEMP is name "TEMP". They just look the same. They aren't.

Output record formats are declared as follows:

format NAME = FORMLIST .

If name is omitted, format "STDOUT" is defined. FORMLIST consists of a sequence of lines, each of which may be of one of three types:

1. A comment, indicated by putting a '#' in the first column.
2. A "picture" line giving the format for one output line.
3. An argument line supplying values to plug into the previous picture line.

Picture lines are printed exactly as they look, except for certain fields.that substitute values into the line. Each field in a picture line starts with either "@" (at) or "^" (caret). These lines do not undergo any kind of variable interpolation. The at field (not to be confused with the array marker @) is the normal kind of field; the other kind, caret fields, are used to do rudimentary multi-line text block filling. The length of the field is supplied by padding out the field with multiple <", ">", or "|" characters to specify, respectively, left justification, right justification, or centering. If the variable would exceed the width specified, it is truncated.

As an alternate form of right justification, you may also use "#" characters (with an optional ".") to specify a numeric field. This way you can line up the decimal points. If any value supplied for these fields contains a newline, only the text up to the newline is printed. Finally, the special field "@*" can be

used for printing multi-line, non-truncated values; it should appear by itself on a line.

The values are specified on the following line in the same order as the picture fields. The expressions providing the values should be separated by commas. The expressions are all evaluated in a list context before the line is processed, so a single list expression could produce multiple list elements. The expressions may be spread out to more than one line if enclosed in braces. If so, the opening brace must be the first token on the first line.

Picture fields that begin with ^ rather than **@** are treated specially. With a # field, the field is blanked out if the value is undefined. For other field types, the caret enables a kind of fill mode. Instead of an arbitrary expression, the value supplied must be a scalar variable name that contains a text string. Perl puts as much text as it can into the field, and then chops off the front of the string so that the next time the variable is referenced, more of the text can be printed. (Yes, this means that the variable itself is altered during execution of the *write()* call, and is not returned.) Normally you would use a sequence of fields in a vertical stack to print out a block of text. You might wish to end the final field with the text "...", which will appear in the output if the text was too long to appear in its entirety. You can change which characters are legal to break on by changing the variable **$:** (that's **$FORMAT_LINE_BREAK_CHARACTERS** if you're using the English module) to a list of the desired characters.

Using caret fields can produce variable length records. If the text to be formatted is short, you can suppress blank lines by putting a "~" (tilde) character anywhere in the line. The tilde will be translated to a space upon output. If you put a second tilde contiguous to the first, the line will be repeated until all the fields on the line are exhausted. (If you use a field of the at variety, the expression you supply had better not give the same value every time forever!)

Top-of-form processing is by default handled by a format with the same name as the current filehandle with "_TOP" concatenated to it. It's triggered at the top of each page. See<perlfunc/ *write()* >.

Examples:

```
# a report on the /etc/passwd file
format STDOUT_TOP =
                        Passwd File
Name                Login    Office   Uid   Gid Home
------------------------------------------------------------------
.
format STDOUT =
@<<<<<<<<<<<<<<<<<< @||||||| @<<<<<<>>>> >>>> @<<<<<<<<<<<<<<<<<
 $name ,              $login ,   $office , $uid , $gid ,  $home
.
# a report from a bug report form
format STDOUT_TOP =
                        Bug Reports
@<<<<<<<<<<<<<<<<<<<<<<     @|||         >>>>>>>>>>>>>>>>
 $system ,                     $%,           $date
------------------------------------------------------------------
```

```
.
format STDOUT =
Subject: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
          $subject
Index: @<<<<<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
        $index ,                        $description
Priority: @<<<<<<<<<< Date: @<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
           $priority ,         $date ,    $description
From: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
       $from ,                          $description
Assigned to: @<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
              $programmer ,             $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                      $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                      $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                      $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                      $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<...
                                      $description
.
```

It is possible to intermix *print()* s with *write()* s on the same output channel, but you'll have to handle $- (**$FORMAT_LINES_LEFT**) yourself.

Format Variables

The current format name is stored in the variable **$~** (**$FORMAT_NAME**), and the current top of form format name is in **$^** (**$FORMAT_TOP_NAME**). The current output page number is stored in **$%** (**$FORMAT_PAGE_NUMBER**), and the number of lines on the page is in **$=** (**$FORMAT_LINES_PER_PAGE**). Whether to autoflush output on this handle is stored in **$|** (**$OUTPUT_AUTOFLUSH**). The string output before each top of page (except the first) is stored in **$^ L** (**$FORMAT_FORMFEED**). These variables are set on a per-filehandle basis, so you'll need to *select()* into a different one to affect them:

select((select(OUTF), $~ = "My_Other_Format", $^ = "My_Top_Format")[0]);

Pretty ugly, eh? It's a common idiom though, so don't be too surprised when you see it. You can at least use a temporary variable to hold the previous filehandle: (this is a much better approach in general, because not only does legibility improve, you now have intermediary stage in the expression to single-step the debugger through):

$ofh = select(OUTF); $~ = "My_Other_Format"; $^ = "My_Top_Format"; select($ofh);

If you use the English module, you can even read the variable names:

use English; $ofh = select(OUTF); $FORMAT_NAME = "My_Other_Format"; $FORMAT_TOP_NAME = "My_Top_Format"; select($ofh);

But you still have those funny *select()* s. So just use the FileHandle module. Now, you can access these special variables using lower-case method names instead:

use FileHandle; format_name OUTF "My_Other_Format"; format_top_name OUTF "My_Top_Format";

Much better!

NOTES

Since the values line may contain arbitrary expressions (for at fields, not caret fields), you can farm out more sophisticated processing to other functions, like *sprintf()* or one of your own. For example:

```
    format Ident =
        @<<<<<<<<<<<<<<<
        &commify( $n )
    .
```

To get a real at or caret into the field, do this:

format Ident = I have an @ here. "@" .

To center a whole line of text, do something like this:

format Ident = @||||||||||||||||||||||||||||||||||||||| "Some text line" .

There is no builtin way to say "float this to the right hand side of the page, however wide it is." You have to specify where it goes. The truly desperate can generate their own format on the fly, based on the current number of columns, and then *eval()* it:

$format = "format STDOUT = \n"; . '^' . '<' x $cols . "\n"; . '$entry' . "\n"; . "\t^" . "<" x ($cols-8) . "~~\n"; . '$entry' . "\n"; . ".\n"; print $format if $Debugging; eval $format; die $@ if $@;

Which would generate a format looking something like this:

```
 format STDOUT =
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
 $entry
        ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<~~
 $entry
 .
```

Here's a little program that's somewhat like fmt(1):

```
format =
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ~~
 $_
.
$/ = '';
while (<>) {
    s/\s*\n\s*/ /g;
    write;
}
```

Footers

While **$FORMAT_TOP_NAME** contains the name of the current header format, there is no corresponding mechanism to automatically do the same thing for a footer. Not knowing how big a format is going to be until you evaluate it is one of the major problems. It's on the TODO list.

Here's one strategy: If you have a fixed-size footer, you can get footers by checking **$FORMAT_LINES_LEFT** before each *write()* and print the footer yourself if necessary.

Here's another strategy; open a pipe to yourself, using **open(MESELF, "|-")** (see *open*) and always *write()* to MESELF instead of STDOUT. Have your child process postprocesses its STDIN to rearrange headers and footers however you like. Not very convenient, but doable.

Accessing Formatting Internals

For low-level access to the formatting mechanism. you may use *formline()* and access **$^ A** (the **$ACCUMULATOR** variable) directly.

For example:

```
 $str  = formline <<'END', 1,2,3;
@<<<  @|||  >>>
END
print "Wow, I just stored `$^A' in the accumulator!\n";
```

Or to make an *swrite()* subroutine which is to *write()* what *sprintf()* is to *printf()* , do this:

use Carp; sub swrite { croak "usage: swrite PICTURE ARGS" unless @_; my $format = shift; $^A = ""; formline($format,@_); return $^A; }

```
$string  = swrite(<<'END', 1, 2, 3);
```

```
Check me out
@<<<  @|||  >>>
END
   print  $string ;
```

TOC

WARNING

Lexical variables (declared with "my") are not visible within a format unless the format is declared within the scope of the lexical variable. (They weren't visiblie at all before version 5.001.)

TOC

NAME

perlipc - Perl interprocess communication

DESCRIPTION

The IPC facilities of Perl are built on the Berkeley socket mechanism. If you don't have sockets, you can ignore this section. The calls have the same names as the corresponding system calls, but the arguments tend to differ, for two reasons. First, Perl file handles work differently than C file descriptors. Second, Perl already knows the length of its strings, so you don't need to pass that information.

Client/Server Communication

Here's a sample TCP client.

($them,$port) = @ARGV; $port = 2345 unless $port; $them = 'localhost' unless $them; $SIG{'INT'} = 'dokill'; sub dokill { kill 9,$child if $child; } use Socket; $sockaddr = 'S n a4 x8'; chop($hostname = `hostname`); ($name, $aliases, $proto) = getprotobyname('tcp'); ($name, $aliases, $port) = getservbyname($port, 'tcp') unless $port =~ /^\d+$/; ($name, $aliases, $type, $len, $thisaddr) = gethostbyname($hostname); ($name, $aliases, $type, $len, $thataddr) = gethostbyname($them); $this = pack($sockaddr, AF_INET, 0, $thisaddr); $that = pack($sockaddr, AF_INET, $port, $thataddr); socket(S, PF_INET, SOCK_STREAM, $proto) || die "socket: $!"; bind(S, $this) || die "bind: $!"; connect(S, $that) || die "connect: $!"; select(S); $| = 1; select(stdout); if ($child = fork) { while (<>) { print S; } sleep 3; do dokill(); } else { while (<S>) { print; } }

And here's a server:

($port) = @ARGV; $port = 2345 unless $port; use Socket; $sockaddr = 'S n a4 x8'; ($name, $aliases, $proto) = getprotobyname('tcp'); ($name, $aliases, $port) = getservbyname($port, 'tcp') unless $port =~ /^\d+$/; $this = pack($sockaddr, AF_INET, $port, "\0\0\0\0"); select(NS); $| = 1; select(stdout); socket(S, PF_INET, SOCK_STREAM, $proto) || die "socket: $!"; bind(S, $this) || die "bind: $!"; listen(S, 5) || die "connect: $!"; select(S); $| = 1; select(stdout); for (;;) { print "Listening again\n"; ($addr = accept(NS,S)) || die $!; print "accept ok\n"; ($af,$port,$inetaddr) = unpack($sockaddr,$addr); @inetaddr = unpack('C4',$inetaddr); print "$af $port @inetaddr\n"; while (<NS>) { print; print NS; } }

SysV IPC

Here's a small example showing shared memory usage:

$IPC_PRIVATE = 0; $IPC_RMID = 0; $size = 2000; $key = shmget($IPC_PRIVATE, $size , 0777);

die if !defined($key); $message = "Message #1"; shmwrite($key, $message, 0, 60) || die "$!"; shmread($key,$buff,0,60) || die "$!"; print $buff,"\n"; print "deleting $key\n"; shmctl($key ,$IPC_RMID, 0) || die "$!";

Here's an example of a semaphore:

$IPC_KEY = 1234; $IPC_RMID = 0; $IPC_CREATE = 0001000; $key = semget($IPC_KEY, $nsems , 0666 | $IPC_CREATE); die if !defined($key); print "$key\n";

Put this code in a separate file to be run in more that one process Call the file *take* :

create a semaphore $IPC_KEY = 1234; $key = semget($IPC_KEY, 0 , 0); die if !defined($key); $semnum = 0; $semflag = 0; # 'take' semaphore # wait for semaphore to be zero $semop = 0; $opstring1 = pack("sss", $semnum, $semop, $semflag); # Increment the semaphore count $semop = 1; $opstring2 = pack("sss", $semnum, $semop, $semflag); $opstring = $opstring1 . $opstring2; semop($key,$opstring) || die "$!";

Put this code in a separate file to be run in more that one process Call this file *give* :

#'give' the semaphore # run this in the original process and you will see # that the second process continues $IPC_KEY = 1234; $key = semget($IPC_KEY, 0, 0); die if !defined($key); $semnum = 0; $semflag = 0; # Decrement the semaphore count $semop = -1; $opstring = pack("sss", $semnum, $semop, $semflag); semop($key,$opstring) || die "$!";

NAME

perlsec - Perl security

DESCRIPTION

Perl is designed to make it easy to write secure setuid and setgid scripts. Unlike shells, which are based on multiple substitution passes on each line of the script, Perl uses a more conventional evaluation scheme with fewer hidden "gotchas". Additionally, since the language has more built-in functionality, it has to rely less upon external (and possibly untrustworthy) programs to accomplish its purposes.

Beyond the obvious problems that stem from giving special privileges to such flexible systems as scripts, on many operating systems, setuid scripts are inherently insecure right from the start. This is because that between the time that the kernel opens up the file to see what to run, and when the now setuid interpreter it ran turns around and reopens the file so it can interpret it, things may have changed, especially if you have symbolic links on your system.

Fortunately, sometimes this kernel "feature" can be disabled. Unfortunately, there are two ways to disable it. The system can simply outlaw scripts with the setuid bit set, which doesn't help much. Alternately, it can simply ignore the setuid bit on scripts. If the latter is true, Perl can emulate the setuid and setgid mechanism when it notices the otherwise useless setuid/gid bits on Perl scripts. It does this via a special executable called **suidperl** that is automatically invoked for you if it's needed.

If, however, the kernel setuid script feature isn't disabled, Perl will complain loudly that your setuid script is insecure. You'll need to either disable the kernel setuid script feature, or put a C wrapper around the script. See the program **wrapsuid** in the *eg* directory of your Perl distribution for how to go about doing this.

There are some systems on which setuid scripts are free of this inherent security bug. For example, recent releases of Solaris are like this. On such systems, when the kernel passes the name of the setuid script to open to the interpreter, rather than using a pathname subject to mettling, it instead passes /dev/fd/3. This is a special file already opened on the script, so that there can be no race condition for evil scripts to exploit. On these systems, Perl should be compiled with **-DSETUID_SCRIPTS_ARE_SECURE_NOW** . The **Configure** program that builds Perl tries to figure this out for itself.

When Perl is executing a setuid script, it takes special precautions to prevent you from falling into any obvious traps. (In some ways, a Perl script is more secure than the corresponding C program.) Any command line argument, environment variable, or input is marked as "tainted", and may not be used, directly or indirectly, in any command that invokes a subshell, or in any command that modifies files, directories, or processes. Any variable that is set within an expression that has previously referenced a tainted value also becomes tainted (even if it is logically impossible for the tainted value to influence the

TOC

variable). For example:

$foo = shift; # $foo is tainted $bar = $foo,'bar'; # $bar is also tainted $xxx = <>; # Tainted $path = $ENV{'PATH'}; # Tainted, but see below $abc = 'abc'; # Not tainted system "echo $foo"; # Insecure system "/bin/echo", $foo; # Secure (doesn't use sh) system "echo $bar"; # Insecure system "echo $abc"; # Insecure until PATH set $ENV{'PATH'} = '/bin:/usr/bin'; $ENV{'IFS'} = " if $ENV{'IFS'} ne "; $path = $ENV{'PATH'}; # Not tainted system "echo $abc"; # Is secure now! open(FOO,"$foo"); # OK open(FOO,">$foo"); # Not OK open(FOO,"echo $foo|"); # Not OK, but... open(FOO,"-|") || exec 'echo', $foo; # OK $zzz = `echo $foo`; # Insecure, zzz tainted unlink $abc,$foo; # Insecure umask $foo; # Insecure exec "echo $foo"; # Insecure exec "echo", $foo; # Secure (doesn't use sh) exec "sh", '-c', $foo; # Considered secure, alas

The taintedness is associated with each scalar value, so some elements of an array can be tainted, and others not.

If you try to do something insecure, you will get a fatal error saying something like "Insecure dependency" or "Insecure PATH". Note that you can still write an insecure system call or exec, but only by explicitly doing something like the last example above. You can also bypass the tainting mechanism by referencing subpatterns--Perl presumes that if you reference a substring using **$1** , **$2** , etc, you knew what you were doing when you wrote the pattern:

$ARGV[0] =~ /^-P(\w+)$/; $printer = $1; # Not tainted

This is fairly secure since **\w+** doesn't match shell metacharacters. Use of **/.+/** would have been insecure, but Perl doesn't check for that, so you must be careful with your patterns. This is the *ONLY* mechanism for untainting user supplied filenames if you want to do file operations on them (unless you make **$& gt;** equal to **$& lt;**).

For "Insecure **$ENV** {PATH}" messages, you need to set **$ENV {'PATH'}** to a known value, and each directory in the path must be non-writable by the world. A frequently voiced gripe is that you can get this message even if the pathname to an executable is fully qualified. But Perl can't know that the executable in question isn't going to execute some other program depending on the PATH.

It's also possible to get into trouble with other operations that don't care whether they use tainted values. Make judicious use of the file tests in dealing with any user-supplied filenames. When possible, do opens and such after setting **$& gt; = $& lt;** . (Remember group IDs, too!) Perl doesn't prevent you from opening tainted filenames for reading, so be careful what you print out. The tainting mechanism is intended to prevent stupid mistakes, not to remove the need for thought.

TOC

NAME

perltrap - Perl traps for the unwary

DESCRIPTION

The biggest trap of all is forgetting to use the **-w** switch; see the *perlrun* manpage . Making your entire program runnable under

use strict;

can help make your program more bullet-proof, but sometimes it's too annoying for quick throw-away programs.

Awk Traps

Accustomed **awk** users should take special note of the following:

- * The English module, loaded via

  use English;

  allows you to refer to special variables (like **$RS**) as though they were in **awk** ; see the *perlvar* manpage for details.
- * Semicolons are required after all simple statements in Perl (except at the end of a block). Newline is not a statement delimiter.
- * Curly brackets are required on **if** s and **while** s.
- * Variables begin with "$" or "@" in Perl.
- * Arrays index from 0. Likewise string positions in *substr()* and *index()* .
- * You have to decide whether your array has numeric or string indices.
- * Associative array values do not spring into existence upon mere reference.
- * You have to decide whether you want to use string or numeric comparisons.
- * Reading an input line does not split it for you. You get to split it yourself to an array. And *split()* operator has different arguments.
- * The current input line is normally in **$_** , not **$0** . It generally does not have the newline stripped. (**$0** is the name of the program executed.) See the *perlvar* manpage .
- * < *digit* > does not refer to fields--it refers to substrings matched by the last match pattern.
- * The *print()* statement does not add field and record separators unless you set **$,** and **$.** . You can set **$OFS** and **$ORS** if you're using the English module.

- * You must open your files before you print to them.
- * The range operator is "..", not comma. The comma operator works as in C.
- * The match operator is "=~", not "~". ("~" is the one's complement operator, as in C.)
- * The exponentiation operator is "**", not "^". "^" is the XOR operator, as in C. (You know, one could get the feeling that **awk** is basically incompatible with C.)
- * The concatenation operator is ".", not the null string. (Using the null string would render **/pat/ /pat/** unparsable, since the third slash would be interpreted as a division operator--the tokener is in fact slightly context sensitive for operators like "/", "?", and ">". And in fact, "." itself can be the beginning of a number.)
- * The **next** , **exit** , and **continue** keywords work differently.
- *

  The following variables work differently:

  - Awk Perl
  - ARGC $#ARGV or scalar @ARGV
  - ARGV[0] $0
  - FILENAME $ARGV
  - FNR $. - something
  - FS (whatever you like)
  - NF $#Fld, or some such
  - NR $.
  - OFMT $#
  - OFS $,
  - ORS $\
  - RLENGTH length($&)
  - RS $/
  - RSTART length($`)
  - SUBSEP $;
- * You cannot set **$RS** to a pattern, only a string.
- * When in doubt, run the **awk** construct through **a2p** and see what it gives you.

.

C Traps

Cerebral C programmers should take note of the following:

- * Curly brackets are required on **if** 's and **while** 's.
- * You must use **elsif** rather than **else if** .
- * The **break** and **continue** keywords from C become in Perl **last** and **next** , respectively. Unlike in C, these do *NOT* work within a **do { } while** construct.

- * There's no switch statement. (But it's easy to build one on the fly.)
- * Variables begin with "$" or "@" in Perl.
- * *printf()* does not implement the "*" format for interpolating field widths, but it's trivial to use interpolation of double-quoted strings to achieve the same effect.
- * Comments begin with "#", not "/*".
- * You can't take the address of anything, although a similar operator in Perl 5 is the backslash, which creates a reference.
- * **ARGV** must be capitalized.
- * System calls such as *link()* , *unlink()* , *rename()* , etc. return nonzero for success, not 0.
- * Signal handlers deal with signal names, not numbers. Use **kill -l** to find their names on your system.

.

Sed Traps

Seasoned **sed** programmers should take note of the following:

- * Backreferences in substitutions use "$" rather than "\".
- * The pattern matching metacharacters "(", ")", and "|" do not have backslashes in front.
- * The range operator is **...** , rather than comma.

.

Shell Traps

Sharp shell programmers should take note of the following:

- * The backtick operator does variable interpretation without regard to the presence of single quotes in the command.
- * The backtick operator does no translation of the return value, unlike **csh** .
- * Shells (especially **csh**) do several levels of substitution on each command line. Perl does substitution only in certain constructs such as double quotes, backticks, angle brackets, and search patterns.
- * Shells interpret scripts a little bit at a time. Perl compiles the entire program before executing it (except for **BEGIN** blocks, which execute at compile time).
- * The arguments are available via **@ARGV** , not **$1** , **$2** , etc.
- * The environment is not automatically made available as separate scalar variables.

.

Perl Traps

Practicing Perl Programmers should take note of the following:

- * Remember that many operations behave differently in a list context than they do in a scalar one. See the *perldata* manpage for details.
- * Avoid barewords if you can, especially all lower-case ones. You can't tell just by looking at it whether a bareword is a function or a string. By using quotes on strings and parens on function calls, you won't ever get them confused.
- * You cannot discern from mere inspection which built-ins are unary operators (like *chop()* and *chdir()*) and which are list operators (like *print()* and *unlink()*). (User-defined subroutines can **only** be list operators, never unary ones.) See the *perlop* manpage .
- * People have a hard time remembering that some functions default to **$_** , or **@ARGV** , or whatever, but that others which you might expect to do not.
- * The<FH> construct is not the name of the filehandle, it is a readline operation on that handle. The data read is only assigned to **$_** if the file read is the sole condition in a while loop:

  while (<FH>) { } while ($_ = <FH>) { }.. <FH>; # data discarded!
- * Remember not to use " = " when you need " =~ "; these two constructs are quite different:

  $x = /foo/; $x =~ /foo/;
- * The **do {}** construct isn't a real loop that you can use loop control on.
- * Use *my()* for local variables whenever you can get away with it (but see the *perlform* manpage for where you can't). Using *local()* actually gives a local value to a global variable, which leaves you open to unforeseen side-effects of dynamic scoping.

.

Perl4 Traps

Penitent Perl 4 Programmers should take note of the following incompatible changes that occurred between release 4 and release 5:

- * **@** now always interpolates an array in double-quotish strings. Some programs may now need to use backslash to protect any **@** that shouldn't interpolate.
- * Barewords that used to look like strings to Perl will now look like subroutine calls if a subroutine by that name is defined before the compiler sees them. For example:

  sub SeeYa { die "Hasta la vista, baby!" } $SIG{'QUIT'} = SeeYa;

  In Perl 4, that set the signal handler; in Perl 5, it actually calls the function! You may use the **-w** switch to find such places.
- * Symbols starting with **_** are no longer forced into package **main** , except for **$_** itself (and **@_** , etc.).
- * **s' $lhs ' $rhs '** now does no interpolation on either side. It used to interpolate **$lhs** but not **$rhs** .

- * The second and third arguments of *splice()* are now evaluated in scalar context (as the book s rather than list context.
- * These are now semantic errors because of precedence:

  shift @list + 20; $n = keys %map + 20;

  Because if that were to work, then this couldn't:

  sleep $dormancy + 20;
- * **open FOO || die** is now incorrect. You need parens around the filehandle. While temporarily supported, using such a construct will generate a non-fatal (but non-suppressible) warning.
- * The elements of argument lists for formats are now evaluated in list context. This means you can interpolate list values now.
- * You can't do a **goto** into a block that is optimized away. Darn.
- * It is no longer syntactically legal to use whitespace as the name of a variable, or as a delimiter for any kind of quote construct. Double darn.
- * The *caller()* function now returns a false value in a scalar context if there is no caller. This lets library files determine if they're being required.
- * **m//g** now attaches its state to the searched string rather than the regular expression.
- * **reverse** is no longer allowed as the name of a sort subroutine.
- * **taintperl** is no longer a separate executable. There is now a **-T** switch to turn on tainting when it isn't turned on automatically.
- * Double-quoted strings may no longer end with an unescaped **$** or **@** .
- * The archaic **while/if** BLOCK BLOCK syntax is no longer supported.
- * Negative array subscripts now count from the end of the array.
- * The comma operator in a scalar context is now guaranteed to give a scalar context to its arguments.
- * The ****** operator now binds more tightly than unary minus. It was documented to work this way before, but didn't.
- * Setting **$# array** lower now discards array elements.
- * *delete()* is not guaranteed to return the old value for *tie()* d arrays, since this capability may be onerous for some modules to implement.
- * The construct "this is **$$** x" used to interpolate the pid at that point, but now tries to dereference **$x** . **$$** by itself still works fine, however.
- * Some error messages will be different.
- * Some bugs may have been inadvertently removed.

.

NAME

perlstyle - Perl style guide

DESCRIPTION

Style

Each programmer will, of course, have his or her own preferences in regards to formatting, but there are some general guidelines that will make your programs easier to read, understand, and maintain.

Regarding aesthetics of code lay out, about the only thing Larry cares strongly about is that the closing curly brace of a multi-line BLOCK should line up with the keyword that started the construct. Beyond that, he has other preferences that aren't so strong:

- * 4-column indent.
- * Opening curly on same line as keyword, if possible, otherwise line up.
- * Space before the opening curly of a multiline BLOCK.
- * One-line BLOCK may be put on one line, including curlies.
- * No space before the semicolon.
- * Semicolon omitted in "short" one-line BLOCK.
- * Space around most operators.
- * Space around a "complex" subscript (inside brackets).
- * Blank lines between chunks that do different things.
- * Uncuddled elses.
- * No space between function name and its opening paren.
- * Space after each comma.
- * Long lines broken after an operator (except "and" and "or").
- * Space after last paren matching on current line.
- * Line up corresponding items vertically.
- * Omit redundant punctuation as long as clarity doesn't suffer.

Larry has his reasons for each of these things, but he doen't claim that.everyone else's mind works the same as his does.

Here are some other more substantive style issues to think about:

- * Just because you *CAN* do something a particular way doesn't mean that you *SHOULD* do it that way. Perl is designed to give you several ways to do anything, so consider picking the most

readable one. For instance

open(FOO,$foo) || die "Can't open $foo: $!";

is better than

die "Can't open $foo: $!" unless open(FOO,$foo);

because the second way hides the main point of the statement in a modifier. On the other hand

print "Starting analysis\n" if $verbose;

is better than

$verbose && print "Starting analysis\n";

since the main point isn't whether the user typed **-v** or not.

Similarly, just because an operator lets you assume default arguments doesn't mean that you have to make use of the defaults. The defaults are there for lazy systems programmers writing one-shot programs. If you want your program to be readable, consider supplying the argument.

Along the same lines, just because you *CAN* omit parentheses in many places doesn't mean that you ought to:

return print reverse sort num values %array; return print(reverse(sort num (values(%array))));

When in doubt, parenthesize. At the very least it will let some poor schmuck bounce on the **%** key in **vi** .

Even if you aren't in doubt, consider the mental welfare of the person who has to maintain the code after you, and who will probably put parens in the wrong place.

- * Don't go through silly contortions to exit a loop at the top or the bottom, when Perl provides the **last** operator so you can exit in the middle. Just "outdent" it a little to make it more visible:

  LINE: for (;;) { statements; last LINE if $foo; next LINE if /^#/; statements; }
- * Don't be afraid to use loop labels--they're there to enhance readability as well as to allow multi-level loop breaks. See the previous example.
- * For portability, when using features that may not be implemented on every machine, test the construct in an eval to see if it fails. If you know what version or patchlevel a particular feature was implemented, you can test **$]** (**$PERL_VERSION** in **English**) to see if it will be there. The **Config** module will also let you interrogate values determined by the **Configure** program when Perl was installed.
- * Choose mnemonic identifiers. If you can't remember what mnemonic means, you've got a problem.
- * If you have a really hairy regular expression, use the **/x** modifier and put in some whitespace to make it look a little less like line noise. Don't use slash as a delimiter when your regexp has slashes or backslashes.
- * Use the new "and" and "or" operators to avoid having to parenthesize list operators so much, and

to reduce the incidence of punctuational operators like **&&** and || . Call your subroutines as if they were functions or list operators to avoid excessive ampersands and parens.

- * Use here documents instead of repeated *print()* statements.
- * Line up corresponding things vertically, especially if it'd be too long to fit on one line anyway.

  $IDX = $ST_MTIME; $IDX = $ST_ATIME if $opt_u; $IDX = $ST_CTIME if $opt_c; $IDX = $ST_SIZE if $opt_s; mkdir $tmpdir, 0700 or die "can't mkdir $tmpdir: $!"; chdir($tmpdir) or die "can't chdir $tmpdir: $!"; mkdir 'tmp', 0777 or die "can't mkdir $tmpdir/tmp: $!";
- * Line up your translations when it makes sense:

  tr [abc] [xyz];
- * Think about reusability. Why waste brainpower on a one-shot when you might want to do something like it again? Consider generalizing your code. Consider writing a module or object class. Consider making your code run cleanly with **use strict** and **-w** in effect. Consider giving away your code. Consider changing your whole world view. Consider... oh, never mind.
- * Be consistent.
- * Be nice.

.

TOC

NAME

perlguts - Perl's Internal Functions

DESCRIPTION

This document attempts to describe some of the internal functions of the Perl executable. It is far from complete and probably contains many errors. Please refer any questions or comments to the author below.

Datatypes

Perl has three typedefs that handle Perl's three main data types:

SV Scalar Value AV Array Value HV Hash Value

Each typedef has specific routines that manipulate the various data type.

What is an "IV"?

Perl uses a special typedef IV which is large enough to hold either an integer or a pointer.

Perl also uses a special typedef I32 which will always be a 32-bit integer.

Working with SV's

An SV can be created and loaded with one command. There are four types of values that can be loaded: an integer value (IV), a double (NV), a string, (PV), and another scalar (SV).

The four routines are:

SV* newSViv(IV); SV* newSVnv(double); SV* newSVpv(char*, int); SV* newSVsv(SV*);

To change the value of an *already-existing* scalar, there are five routines:

void sv_setiv(SV*, IV); void sv_setnv(SV*, double); void sv_setpvn(SV*, char*, int) void sv_setpv(SV*, char*); void sv_setsv(SV*, SV*);

Notice that you can choose to specify the length of the string to be assigned by using **sv_setpvn** , or

allow Perl to calculate the length by using **sv_setpv** . Be warned, though, that **sv_setpv** determines th string's length by using **strlen** , which depends on the string terminating with a NUL character.

To access the actual value that an SV points to, you can use the macros:

SvIV(SV*) SvNV(SV*) SvPV(SV*, STRLEN len)

which will automatically coerce the actual scalar type into an IV, double, or string.

In the **SvPV** macro, the length of the string returned is placed into the variable **len** (this is a macro, so you do *not* use **&len**). If you do not care what the length of the data is, use the global variable **na** . Remember, however, that Perl allows arbitrary strings of data that may both contain NUL's and not be terminated by a NUL.

If you simply want to know if the scalar value is TRUE, you can use:

SvTRUE(SV*)

Although Perl will automatically grow strings for you, if you need to force Perl to allocate more memory for your SV, you can use the macro

SvGROW(SV*, STRLEN newlen)

which will determine if more memory needs to be allocated. If so, it will call the function **sv_grow** . Note that **SvGROW** can only increase, not decrease, the allocated memory of an SV.

If you have an SV and want to know what kind of data Perl thinks is stored in it, you can use the following macros to check the type of SV you have.

SvIOK(SV*) SvNOK(SV*) SvPOK(SV*)

You can get and set the current length of the string stored in an SV with the following macros:

SvCUR(SV*) SvCUR_set(SV*, I32 val)

But note that these are valid only if ***SvPOK()*** is true.

If you know the name of a scalar variable, you can get a pointer to its SV by using the following:

SV* perl_get_sv("varname", FALSE);

This returns NULL if the variable does not exist.

If you want to know if this variable (or any other SV) is actually defined, you can call:

SvOK(SV*)

The scalar **undef** value is stored in an SV instance called **sv_undef** . Its address can be used whenever an **SV*** is needed.

There are also the two values **sv_yes** and **sv_no** , which contain Boolean TRUE and FALSE values, respectively. Like **sv_undef** , their addresses can be used whenever an **SV*** is needed.

Do not be fooled into thinking that **(SV *) 0** is the same as **&sv_undef** . Take this code:

SV* sv = (SV*) 0; if (I-am-to-return-a-real-value) { sv = sv_2mortal(newSViv(42)); } sv_setsv(ST(0), sv);

This code tries to return a new SV (which contains the value 42) if it should return a real value, or undef otherwise. Instead it has returned a null pointer which, somewhere down the line, will cause a segmentation violation, or just weird results. Change the zero to **&sv_undef** in the first line and all will be well.

To free an SV that you've created, call **SvREFCNT_dec(SV*)** . Normally this call is not necessary. See the section on **MORTALITY** .

Private and Public Values

Recall that the usual method of determining the type of scalar you have is to use **Sv[INP]OK** macros. Since a scalar can be both a number and a string, usually these macros will always return TRUE and calling the **Sv[INP]V** macros will do the appropriate conversion of string to integer/double or integer/double to string.

If you *really* need to know if you have an integer, double, or string pointer in an SV, you can use the following three macros instead:

SvIOKp(SV*) SvNOKp(SV*) SvPOKp(SV*)

These will tell you if you truly have an integer, double, or string pointer stored in your SV.

In general, though, it's best to just use the **Sv[INP]V** macros.

Working with AV's

There are two ways to create and load an AV. The first method just creates an empty AV:

AV* newAV();

The second method both creates the AV and initially populates it with SV's:

AV* av_make(I32 num, SV **ptr);

The second argument points to an array containing **num SV*** 's.

Once the AV has been created, the following operations are possible on AV's:

void av_push(AV*, SV*); SV* av_pop(AV*); SV* av_shift(AV*); void av_unshift(AV*, I32 num);

These should be familiar operations, with the exception of **av_unshift** . This routine adds **num** elements at the front of the array with the **undef** value. You must then use **av_store** (described below) to assign values to these new elements.

Here are some other functions:

I32 av_len(AV*); /* Returns length of array */ SV** av_fetch(AV*, I32 key, I32 lval); /* Fetches va at key offset, but it seems to set the value to lval if lval is non-zero */ SV** av_store(AV*, I32 key, S val); /* Stores val at offset key */ void av_clear(AV*); /* Clear out all elements, but leave the array */ void av_undef(AV*); /* Undefines the array, removing all elements */

If you know the name of an array variable, you can get a pointer to its AV by using the following:

AV* perl_get_av("varname", FALSE);

This returns NULL if the variable does not exist.

Working with HV's

To create an HV, you use the following routine:

HV* newHV();

Once the HV has been created, the following operations are possible on HV's:

SV** hv_store(HV*, char* key, U32 klen, SV* val, U32 hash); SV** hv_fetch(HV*, char* key, U32 klen, I32 lval);

The **klen** parameter is the length of the key being passed in. The **val** argument contains the SV pointer to the scalar being stored, and **hash** is the pre-computed hash value (zero if you want **hv_store** to calculate it for you). The **lval** parameter indicates whether this fetch is actually a part of a store operation.

Remember that **hv_store** and **hv_fetch** return **SV**** 's and not just **SV*** . In order to access the scalar value, you must first dereference the return value. However, you should check to make sure that the return value is not NULL before dereferencing it.

These two functions check if a hash table entry exists, and deletes it.

bool hv_exists(HV*, char* key, U32 klen); SV* hv_delete(HV*, char* key, U32 klen);

And more miscellaneous functions:

void hv_clear(HV*); /* Clears all entries in hash table */ void hv_undef(HV*); /* Undefines the hash table */ I32 hv_iterinit(HV*); /* Prepares starting point to traverse hash table */ HE* hv_iternext(HV*); /* Get the next entry, and return a pointer to a structure that has both the key and value */ char* hv_iterkey(HE* entry, I32* retlen); /* Get the key from an HE structure and also return the length of the key string */ SV* hv_iterval(HV*, HE* entry); /* Return a SV pointer to the value of the HE structure */

If you know the name of a hash variable, you can get a pointer to its HV by using the following:

HV* perl_get_hv("varname", FALSE);

This returns NULL if the variable does not exist.

The hash algorithm, for those who are interested, is:

i = klen; hash = 0; s = key; while (i--) hash = hash * 33 + *s++;

References

References are a special type of scalar that point to other scalar types (including references). To treat an AV or HV as a scalar, it is simply a matter of casting an AV or HV to an SV.

To create a reference, use the following command:

SV* newRV((SV*) pointer);

Once you have a reference, you can use the following macro with a cast to the appropriate typedef (SV, AV, HV):

SvRV(SV*)

then call the appropriate routines, casting the returned **SV*** to either an **AV*** or **HV*** .

To determine, after dereferencing a reference, if you still have a reference, you can use the following macro:

SvROK(SV*)

XSUB'S and the Argument Stack

The XSUB mechanism is a simple way for Perl programs to access C subroutines. An XSUB routine will have a stack that contains the arguments from the Perl program, and a way to map from the Perl data structures to a C equivalent.

The stack arguments are accessible through the **ST(n)** macro, which returns the **n** 'th stack argument. Argument 0 is the first argument passed in the Perl subroutine call. These arguments are **SV*** , and can be used anywhere an **SV*** is used.

Most of the time, output from the C routine can be handled through use of the RETVAL and OUTPUT directives. However, there are some cases where the argument stack is not already long enough to handle all the return values. An example is the POSIX *tzname()* call, which takes no arguments, but returns two, the local timezone's standard and summer time abbreviations.

To handle this situation, the PPCODE directive is used and the stack is extended using the macro:

EXTEND(sp, num);

where **sp** is the stack pointer, and **num** is the number of elements the stack should be extended by.

Now that there is room on the stack, values can be pushed on it using the macros to push IV's, doubles, strings, and SV pointers respectively:

PUSHi(IV) PUSHn(double) PUSHp(char*, I32) PUSHs(SV*)

And now the Perl program calling **tzname** , the two values will be assigned as in:

($standard_abbrev, $summer_abbrev) = POSIX::tzname;

An alternate (and possibly simpler) method to pushing values on the stack is to use the macros:

XPUSHi(IV) XPUSHn(double) XPUSHp(char*, I32) XPUSHs(SV*)

These macros automatically adjust the stack for you, if needed.

Mortality

In Perl, values are normally "immortal" -- that is, they are not freed unless explicitly done so (via the Perl **undef** call or other routines in Perl itself).

In the above example with **tzname** , we needed to create two new SV's to push onto the argument stack, that being the two strings. However, we don't want these new SV's to stick around forever because they will eventually be copied into the SV's that hold the two scalar variables.

An SV (or AV or HV) that is "mortal" acts in all ways as a normal "immortal" SV, AV, or HV, but is only valid in the "current context". When the Perl interpreter leaves the current context, the mortal SV, AV, or HV is automatically freed. Generally the "current context" means a single Perl statement.

To create a mortal variable, use the functions:

SV* sv_newmortal() SV* sv_2mortal(SV*) SV* sv_mortalcopy(SV*)

The first call creates a mortal SV, the second converts an existing SV to a mortal SV, the third creates a mortal copy of an existing SV.

The mortal routines are not just for SV's -- AV's and HV's can be made mortal by passing their address (and casting them to **SV***) to the **sv_2mortal** or **sv_mortalcopy** routines.

Creating New Variables

To create a new Perl variable, which can be accessed from your Perl script, use the following routines, depending on the variable type.

SV* perl_get_sv("varname", TRUE); AV* perl_get_av("varname", TRUE); HV* perl_get_hv("varname", TRUE);

Notice the use of TRUE as the second parameter. The new variable can now be set, using the routines appropriate to the data type.

Stashes and Objects

A stash is a hash table (associative array) that contains all of the different objects that are contained within a package. Each key of the hash table is a symbol name (shared by all the different types of objects that have the same name), and each value in the hash table is called a GV (for Glob Value). The GV in turn contains references to the various objects of that name, including (but not limited to) the following: Scalar Value Array Value Hash Value File Handle Directory Handle Format Subroutine

Perl stores various stashes in a GV structure (for global variable) but represents them with an HV structure.

To get the HV pointer for a particular package, use the function:

HV* gv_stashpv(char* name, I32 create) HV* gv_stashsv(SV*, I32 create)

The first function takes a literal string, the second uses the string stored in the SV.

The name that **gv_stash*v** wants is the name of the package whose symbol table you want. The default package is called **main** . If you have multiply nested packages, it is legal to pass their names to **gv_stash*v** , separated by **::** as in the Perl language itself.

Alternately, if you have an SV that is a blessed reference, you can find out the stash pointer by using:

HV* SvSTASH(SvRV(SV*));

then use the following to get the package name itself:

char* HvNAME(HV* stash);

If you need to return a blessed value to your Perl script, you can use the following function:

SV* sv_bless(SV*, HV* stash)

where the first argument, an **SV*** , must be a reference, and the second argument is a stash. The returned **SV*** can now be used in the same way as any other SV.

Magic

[This section under construction]

Double-Typed SV's

Scalar variables normally contain only one type of value, an integer, double, pointer, or reference. Perl will automatically convert the actual scalar data from the stored type into the requested type.

Some scalar variables contain more than one type of scalar data. For example, the variable **$!** contains either the numeric value of **errno** or its string equivalent from **sys_errlist[]** .

To force multiple data values into an SV, you must do two things: use the **sv_set*v** routines to add the additional scalar type, then set a flag so that Perl will believe it contains more than one type of data. The four macros to set the flags are:

SvIOK_on SvNOK_on SvPOK_on SvROK_on

The particular macro you must use depends on which **sv_set*v** routine you called first. This is because every **sv_set*v** routine turns on only the bit for the particular type of data being set, and turns off all the rest.

For example, to create a new Perl variable called "dberror" that contains both the numeric and descriptive string error values, you could use the following code:

extern int dberror; extern char *dberror_list; SV* sv = perl_get_sv("dberror", TRUE); sv_setiv(sv, (IV) dberror); sv_setpv(sv, dberror_list[dberror]); SvIOK_on(sv);

If the order of **sv_setiv** and **sv_setpv** had been reversed, then the macro **SvPOK_on** would need to be called instead of **SvIOK_on** .

Calling Perl Routines from within C Programs

There are four routines that can be used to call a Perl subroutine from within a C program. These four are:

I32 perl_call_sv(SV*, I32); I32 perl_call_pv(char*, I32); I32 perl_call_method(char*, I32); I32 perl_call_argv(char*, I32, register char**);

The routine most often used should be **perl_call_sv** . The **SV*** argument contains either the name of the Perl subroutine to be called, or a reference to the subroutine. The second argument tells the appropriate routine what, if any, variables are being returned by the Perl subroutine.

All four routines return the number of arguments that the subroutine returned on the Perl stack.

When using these four routines, the programmer must manipulate the Perl stack. These include the following macros and functions:

dSP PUSHMARK() PUTBACK SPAGAIN ENTER SAVETMPS FREETMPS LEAVE XPUSH*()

For more information, consult the *perlcall* manpage .

Memory Allocation

[This section under construction]

AUTHOR

Jeff Okamoto<okamoto **@corp** .hp.com>

With lots of help and suggestions from Dean Roehrich, Malcolm Beattie, Andreas Koenig, Paul Hudson, Ilya Zakharevich, Paul Marquess, and Neil Bowers.

DATE

Version 12: 1994/10/16

NAME

perlcall - Perl calling conventions from C

DESCRIPTION

BThe purpose of this document is to show you how to write *callbacks* , i.e. how to call Perl from C. The main focus is on how to interface back to Perl from a bit of C code that has itself been run by Perl, i.e. the 'main' program is a Perl script; you are using it to execute a section of code written in C; that bit of C code wants you to do something with a particular event, so you want a Perl sub to be executed whenever it happens.

Examples where this is necessary include

- * You have created an XSUB interface to an application's C API.

  A fairly common feature in applications is to allow you to define a C function that will get called whenever something nasty occurs. What we would like is for a Perl sub to be called instead.
- * The classic example of where callbacks are used is in an event driven program like for X-windows. In this case your register functions to be called whenever a specific events occur, e.g. a mouse button is pressed.

Although the techniques described are applicable to embedding Perl.in a C program, this is not the primary goal of this document. For details on embedding Perl in C refer to the *perlembed* manpage (currently unwritten).

Before you launch yourself head first into the rest of this document, it would be a good idea to have read the following two documents - the *perlapi* manpage and the *perlguts* manpage .

This stuff is easier to explain using examples. But first here are a few definitions anyway.

Definitions

Perl has a number of C functions which allow you to call Perl subs. They are

I32 perl_call_sv(SV* sv, I32 flags) ; I32 perl_call_pv(char *subname, I32 flags) ; I32 perl_call_method(char *methname, I32 flags) ; I32 perl_call_argv(char *subname, I32 flags, register char **argv) ;

The key function is *perl_call_sv* . All the other functions make use of *perl_call_sv* to do what they do.

perl_call_sv takes two parameters, the first is an SV*. This allows you to specify the Perl sub to be called either as a C string (which has first been converted to an SV) or a reference to a sub. Example 7, shows

you how you can make use of *perl_call_sv* . The second parameter, **flags** , is a general purpose optio command. This parameter is common to all the *perl_call_** functions. It is discussed in the next sect

The function, *perl_call_pv* , is similar as *perl_call_sv* except it expects it's first parameter has to be a C char* which identifies the Perl sub you want to call, e.g. **perl_call_pv("fred", 0)** .

The function *perl_call_method* expects its first argument to contain a blessed reference to a class. Using that reference it looks up and calls **methname** from that class. See example 9.

perl_call_argv calls the Perl sub specified by the **subname** parameter. It also takes the usual **flags** parameter. The final parameter, **argv** , consists of a list of C strings to be sent to the Perl sub. See example 8.

All the functions return a number. This is a count of the number of items returned by the Perl sub on the stack.

As a general rule you should *always* check the return value from these functions. Even if you are only expecting a particular number of values to be returned from the Perl sub, there is nothing to stop someone from doing something unexpected - don't say you havn't been warned.

Flag Values

The **flags** parameter in all the *perl_call_** functions consists of any combination of the symbols defined below, OR'ed together.

G_SCALAR

Calls the Perl sub in a scalar context.

Whatever the Perl sub actually returns, we only want a scalar. If the perl sub does return a scalar, the return value from the *perl_call_** function will be 1 or 0. If 1, then the value actually returned by the Perl sub will be contained on the top of the stack. If 0, then the sub has probably called *die* or you have used the G_DISCARD flag.

If the Perl sub returns a list, the *perl_call_** function will still only return 1 or 0. If 1, then the number of elements in the list will be stored on top of the stack. The actual values of the list will not be accessable.

G_SCALAR is the default flag setting for all the functions.

G_ARRAY

Calls the Perl sub in a list context.

The return code from the *perl_call_** functions will indicate how many elements of the stack are used to store the array.

G_DISCARD

If you are not interested in the values returned by the Perl sub then setting this flag will make Perl get rid of them automatically for you. This will take precedence to either G_SCALAR or G_ARRAY.

If you do not set this flag then you may need to explicitly get rid of temporary values. See exa
3 for details.

G_NOARGS

If you are not passing any parameters to the Perl sub, you can save a bit of time by setting this flag. It has the effect of of not creating the **@_** array for the Perl sub.

A point worth noting is that if this flag is specified the Perl sub called can still access an **@_** array from a previous Perl sub. This functionality can be illustrated with the perl code below

sub fred { print "@_\n" } sub joe { &fred } &joe(1,2,3) ;

This will print

1 2 3

What has happened is that **fred** accesses the **@_** array which belongs to **joe** .

G_EVAL

If the Perl sub you are calling has the ability to terminate abnormally, e.g. by calling *die* or by not actually existing, and you want to catch this type of event, specify this flag setting. It will put an *eval { }* around the sub call.

Whenever control returns from the *perl_call_** function you need to check the **$@** variable as you would in a normal Perl script. See example 6 for details of how to do this.

.

EXAMPLES

Enough of the definition talk, let's have a few examples.

Perl provides many macros to assist in accessing the Perl stack. These macros should always be used when interfacing to Perl internals. Hopefully this should make the code less vulnerable to changes made to Perl in the future.

Another point worth noting is that in the first series of examples I have only made use of the *perl_call_pv* function. This has only been done to ease you into the topic. Wherever possible, if the choice is between using *perl_call_pv* and *perl_call_sv* , I would always try to use *perl_call_sv* .

The code for these examples is stored in the file *perlcall.tar* . (Once this document settles down, all the example code will be available in the file).

Example1: No Parameters, Nothing returned

This first trivial example will call a Perl sub, *PrintUID* , to print out the UID of the process.

sub PrintUID { print "UID is $<\n" ; }

and here is the C to call it

```
void call_PrintUID() { dSP ; PUSHMARK(sp) ; perl_call_pv("PrintUID", G_DISCARD|G_NOARGS) ;
}
```

Simple, eh.

A few points to note about this example.

1. We aren't passing any parameters to *PrintUID* so G_NOARGS can be specified.
2. Ignore **dSP** and **PUSHMARK(sp)** for now. They will be discussed in the next example.
3. We aren't interested in anything returned from *PrintUID* , so G_DISCARD is specified. Even if *PrintUID* was changed to actually return some value(s), having specified G_DISCARD will mean that they will be wiped by the time control returns from *perl_call_pv* .
4. Because we specified G_DISCARD, it is not necessary to check the value returned from *perl_call_sv* . It will always be 0.
5. As *perl_call_pv* is being used, the Perl sub is specified as a C string.

.

Example 2: Passing Parameters

Now let's make a slightly more complex example. This time we want to call a Perl sub which will take 2 parameters - a string (**$s**) and an integer (**$n**). The sub will simply print the first **$n** characters of the string.

So the Perl sub would look like this

```
sub LeftString { my($s, $n) = @_ ; print substr($s, 0, $n), "\n" ; }
```

The C function required to call *LeftString* would look like this.

```
static void call_LeftString(a, b) char * a ; int b ; { dSP ; PUSHMARK(sp) ;
XPUSHs(sv_2mortal(newSVpv(a, 0))); XPUSHs(sv_2mortal(newSViv(b))); PUTBACK ;
perl_call_pv("LeftString", G_DISCARD); }
```

Here are a few notes on the C function *call_LeftString* .

1. The only flag specified this time is G_DISCARD. As we are passing 2 parameters to the Perl sub this time, we have not specified G_NOARGS.
2. Parameters are passed to the Perl sub using the Perl stack. This is the purpose of the code beginning with the line **dSP** and ending with the line **PUTBACK** .
3. If you are going to put something onto the Perl stack, you need to know where to put it. This is the purpose of the macro **dSP** - it declares and initialises a local copy of the Perl stack pointer.

   All the other macros which will be used in this example require you to have used this macro.

   If you are calling a Perl sub directly from an XSUB function, it is not necessary to explicitly use the **dSP** macro - it will be declared for you.

4. Any parameters to be pushed onto the stack should be bracketed by the **PUSHMARK** and **PUTBACK** macros. The purpose of these two macros, in this context, is to automatically count the number of parameters you are pushing. Then whenever Perl is creating the **@_** array for the sub, it knows how big to make it.

   The **PUSHMARK** macro tells Perl to make a mental note of the current stack pointer. Even if you aren't passing any parameters (like in Example 1) you must still call the **PUSHMARK** macro before you can call any of the *perl_call_** functions - Perl still needs to know that there are no parameters.

   The **PUTBACK** macro sets the global copy of the stack pointer to be the same as our local copy. If we didn't do this *perl_call_pv* wouldn't know where the two parameters we pushed were - remember that up to now all the stack pointer manipulation we have done is with our local copy, *not* the global copy.
5. Next, we come to XPUSHs. This is where the parameters actually get pushed onto the stack. In this case we are pushing a string and an integer.

   See the section *XSUB's AND THE ARGUMENT STACK* in the *perlguts* manpage for details on how the XPUSH macros work.
6. Finally, *LeftString* can now be called via the *perl_call_pv* function.

.

Example 3: Returning a Scalar

Now for an example of dealing with the values returned from a Perl sub.

Here is a Perl sub, *Adder* , which takes 2 integer parameters and simply returns their sum.

sub Adder { my($a, $b) = @_ ; $a + $b ; }

As we are now concerned with the return value from *Adder* , the C function is now a bit more complex.

static void call_Adder(a, b) int a ; int b ; { dSP ; int count ; ENTER ; SAVETMPS; PUSHMARK(sp) ; XPUSHs(sv_2mortal(newSViv(a))); XPUSHs(sv_2mortal(newSViv(b))); PUTBACK ; count = perl_call_pv("Adder", G_SCALAR); SPAGAIN ; if (count != 1) croak("Big trouble\n") ; printf ("The sum of %d and %d is %d\n", a, b, POPi) ; PUTBACK ; FREETMPS ; LEAVE ; }

Points to note this time are

1. The only flag specified this time was G_SCALAR. That means the **@_** array will be created and that the value returned by *Adder* will still exist after the call to *perl_call_pv* .
2. Because we are interested in what is returned from *Adder* we cannot specify G_DISCARD. This means that we will have to tidy up the Perl stack and dispose of any temporary values ourselves. This is the purpose of

   ENTER ; SAVETMPS ;

   at the start of the function, and

FREETMPS ; LEAVE ;

at the end. The **ENTER** / **SAVETMPS** pair creates a boundary for any temporaries we create. This means that the temporaries we get rid of will be limited to those which were created after these calls.

The **FREETMPS** / **LEAVE** pair will get rid of any values returned by the Perl sub, plus it will also dump the mortal SV's we created. Having **ENTER** / **SAVETMPS** at the beginning of the code makes sure that no other mortals are destroyed.

3. The purpose of the macro **SPAGAIN** is to refresh the local copy of the stack pointer. This is necessary because it is possible that the memory allocated to the Perl stack has been re-allocated whilst in the *perl_call_pv* call.

   If you are making use of the Perl stack pointer in your code you must always refresh the your local copy using SPAGAIN whenever you make use of of the *perl_call_** functions or any other Perl internal function.

4. Although only a single value was expected to be returned from *Adder* , it is still good practice to check the return code from *perl_call_pv* anyway.

   Expecting a single value is not quite the same as knowing that there will be one. If someone modified *Adder* to return a list and we didn't check for that possibility and take appropriate action the Perl stack would end up in an inconsistant state. That is something you *really* don't want to ever happen.

5. The **POPi** macro is used here to pop the return value from the stack. In this case we wanted an integer, so **POPi** was used.

   Here is the complete list of POP macros available, along with the types they return.

   - POPs SV
   - POPp pointer
   - POPn double
   - POPi integer
   - POPl long

6. The final **PUTBACK** is used to leave the Perl stack in a consistant state before exiting the function. This is necessary because when we popped the return value from the stack with **POPi** it only updated our local copy of the stack pointer. Remember, **PUTBACK** sets the global stack pointer to be the same as our local copy.

.

Example 4: Returning a list of values

Now, let's extend the previous example to return both the sum of the parameters and the difference.

Here is the Perl sub

sub AddSubtract { my($a, $b) = @_ ; ($a+$b, $a-$b) ; }

and this is the C function

static void call_AddSubtract(a, b) int a ; int b ; { dSP ; int count ; ENTER ; SAVETMPS; PUSHMARK(sp) ; XPUSHs(sv_2mortal(newSViv(a))); XPUSHs(sv_2mortal(newSViv(b))); PUTBACK ; count = perl_call_pv("AddSubtract", G_ARRAY); SPAGAIN ; if (count != 2) croak("Big trouble\n") ; printf ("%d - %d = %d\n", a, b, POPi) ; printf ("%d + %d = %d\n", a, b, POPi) ; PUTBACK ; FREETMPS ; LEAVE ; }

Notes

1. We wanted array context, so we used G_ARRAY.
2. Not surprisingly there are 2 POPi's this time because we were retrieving 2 values from the stack. The main point to note is that they came off the stack in reverse order.

.

Example 5: Returning Data from Perl via the parameter list

It is also possible to return values directly via the parameter list - whether it is actually desirable to do it is another matter entirely.

The Perl sub, *Inc* , below takes 2 parameters and increments each.

sub Inc { ++ $_[0] ; ++ $_[1] ; }

and here is a C function to call it.

static void call_Inc(a, b) int a ; int b ; { dSP ; int count ; SV * sva ; SV * svb ; ENTER ; SAVETMPS; sva = sv_2mortal(newSViv(a)) ; svb = sv_2mortal(newSViv(b)) ; PUSHMARK(sp) ; XPUSHs(sva); XPUSHs(svb); PUTBACK ; count = perl_call_pv("Inc", G_DISCARD); if (count != 0) croak ("call_Inc : expected 0 return value from 'Inc', got %d\n", count) ; printf ("%d + 1 = %d\n", a, SvIV(sva)) ; printf ("%d + 1 = %d\n", b, SvIV(svb)) ; FREETMPS ; LEAVE ; }

To be able to access the two parameters that were pushed onto the stack after they return from *perl_call_pv* it is necessary to make a note of their addresses - thus the two variables **sva** and **svb** .

The reason this is necessary is that the area of the Perl stack which held them will very likely have been overwritten by something else by the time control returns from *perl_call_pv* .

Example 6: Using G_EVAL

Now an example using G_EVAL. Below is a Perl sub which computes the difference of its 2 parameters. If this would result in a negative result, the sub calls *die* .

sub Subtract { my ($a, $b) = @_ ; die "death can be fatal\n" if $a < $b ; $a - $b ; }

and some C to call it

static void call_Subtract(a, b) int a ; int b ; { dSP ; int count ; SV * sv ; ENTER ; SAVETMPS; PUSHMARK(sp) ; XPUSHs(sv_2mortal(newSViv(a))); XPUSHs(sv_2mortal(newSViv(b))); PUTBACK ; count = perl_call_pv("Subtract", G_EVAL|G_SCALAR); /* Check the eval first */ sv = GvSV(gv_fetchpv("@", TRUE, SVt_PV)); if (SvTRUE(sv)) printf ("Uh oh - %s\n", SvPV(sv, na)) ; SPAGAIN ; if (count != 1) croak ("call_Subtract : expected 1 return value from 'Subtract', got %d\n", count) ; printf ("%d - %d = %d\n", a, b, POPi) ; PUTBACK ; FREETMPS ; LEAVE ; }

If *call_Subtract* is called thus

call_Subtract(4, 5)

the following will be printed

Uh oh - death can be fatal

Notes

1. We want to be able to catch the *die* so we have used the G_EVAL flag. Not specifying this flag would mean that the program would terminate.
2. The code

   sv = GvSV(gv_fetchpv("@", TRUE, SVt_PV)); if (SvTRUE(sv)) printf ("Uh oh - %s\n", SvPVx(sv, na)) ;

   is the equivalent of this bit of Perl

   print "Uh oh - $@\n" if $@ ;

.

Example 7: Using perl_call_sv

In all the previous examples I have 'hard-wried' the name of the Perl sub to be called from C. Sometimes though, it is necessary to be able to specify the name of the Perl sub from within the Perl script.

Consider the Perl code below

sub fred { print "Hello there\n" ; } CallSub("fred") ;

here is a snippet of XSUB which defines *CallSub* .

void CallSub(name) char * name CODE: PUSHMARK(sp) ; perl_call_pv(name, G_DISCARD|G_NOARGS) ;

That is fine as far as it goes. The thing is, it only allows the Perl sub to be specified as a string. For perl 4 this was adequate, but Perl 5 allows references to subs and anonymous subs. This is where *perl_call_sv* is useful.

The code below for *CallSub* is identical to the previous time except that the **name** parameter is now defined as an SV* and we use *perl_call_sv* instead of *perl_call_pv* .

void CallSub(name) SV* name CODE: PUSHMARK(sp) ; perl_call_sv(name,

G_DISCARD|G_NOARGS) ;

As we are using an SV to call *fred* the following can all be used

CallSub("fred") ; Callsub(\&fred) ; $ref = \&fred ; CallSub($ref) ; CallSub(sub { print "Hello there\n" }) ;

As you can see, *perl_call_sv* gives you greater flexibility in how you can specify the Perl sub.

Example 8: Using perl_call_argv

Here is a Perl sub which prints whatever parameters are passed to it.

sub PrintList { my(@list) = @_ ; foreach (@list) { print "$_\n" } }

and here is an example of *perl_call_argv* which will call *PrintList* .

call_PrintList { dSP ; char * words[] = {"alpha", "beta", "gamma", "delta", NULL } ; perl_call_argv("PrintList", words, G_DISCARD) ; }

Note that it is not necessary to call **PUSHMARK** in this instance. This is because *perl_call_argv* will do it for you.

Example 9: Using perl_call_method

[This section is under construction]

Consider the following Perl code

{ package Mine ; sub new { bless [@_] } sub Display { print $_[0][1], "\n" } } $a = new Mine ('red', 'green', 'blue') ; call_Display($a, 'Display') ;

The method **Display** just prints out the first element of the list. Here is a XSUB implementation of *call_Display* .

void call_Display(ref, method) SV * ref char * method CODE: PUSHMARK(sp); XPUSHs(ref); PUTBACK; perl_call_method(method, G_DISCARD) ;

Strategies for storing Context Information

[This section is under construction]

One of the trickiest problems to overcome when designing a callback interface is figuring out how to store the mapping between the C callback functions and the Perl equivalent.

Consider the following example.

Alternate Stack Manipulation

TOC

[This section is under construction]

Although I have only made use of the POP* macros to access values returned from Perl subs, it is also possible to bypass these macros and read the stack directly.

The code below is example 4 recoded to

SEE ALSO

the *perlapi* manpage , the *perlguts* manpage , the *perlembed* manpage

---

AUTHOR

Paul Marquess<pmarquess **@bfsec** .bt.co.uk>

Special thanks to the following people who assisted in the creation of the document.

Jeff Okamoto, Tim Bunce.

DATE

Version 0.4, 17th October 1994

TOC

NAME

perlovl - perl overloading semantics

SYNOPSIS

package SomeThing;

%OVERLOAD = ('+' => \&myadd, '-' => \&mysub, # etc); ... package main; $a = new SomeThing 57; $b=5+$a;

CAVEAT SCRIPTOR

Overloading of operators is a subject not to be taken lightly. Neither its precise implementation, syntax, nor semantics are 100 **%** endorsed by Larry Wall. So any of these may be changed at some point in the future.

DESCRIPTION

Declaration of overloaded functions

package Number; **%OVERLOAD** = ("+" => \&add, "*=" => "muas");

declares function Number:: *add()* for addition, and method *muas()* in the "class" **Number** (or one of its base classes) for the assignment form ***=** of multiplication. Legal values of this hash array are values legal inside **&{ ... }** call, so the name of a subroutine, a reference to a subroutine, or an anonymous subroutine will all work.

The subroutine **$OVERLOAD {"+"}** will be called to execute **$a + $b** if **$a** is a reference to an object blessed into the package **Number** , or **$a** is not an object from a package with defined mathemagic addition, but **$b** is a reference to a **Number** . It can be called also in other situations, like **$a +=7** , or **$a ++** . See *MAGIC AUTOGENERATION* . (Mathemagical methods refer to methods triggered by an overloaded mathematical operator.)

Calling Conventions for Binary Operations

The functions in **values %OVERLOAD** are called with three (in one particular case with four, see *Last Resort*) arguments. If the corresponding operation is binary, then the first two arguments are the two arguments of the operation. However, due to general object calling conventions, the first argument should be always an object in the package, so in the situation of **7+ $a** , the order of arguments is interchanged. Most probably it does not matter for implementation of the addition method, but whether the arguments are reversed is vital for the subtraction method. The subroutine can query this information by examining the third argument, which can take three different values:

FALSE

the order of arguments is as in the current operation.

TRUE

the arguments are reversed.

undef

the current operation is an assignment variant (as in **$a +=7**), but the usual function is called instead. This additional information can be used to generate some optimizations.

.

Calling Conventions for Unary Operations

Unary operation are considered binary operations with the second argument being undef . Thus **$OVERLOAD {"++"}** is called with arguments **($a ,undef,'')** when **$a** ++ is executed.

Overloadable Operations

The following keys of **%OVERLOAD** are recognized:

*** *Arithmetic operations***

"+", "+=", "-", "-=", "*", "*=", "/", "/=", "%", "%=", "**", "**=", "<<", "<<=", >>", >>=", "x", "x=", ".", ".=",

For these operations a substituted non-assignment variant can be called if the assignment variant is not available. Methods for operations " + ", " - ", " += ", and " -= " can be called to automatically generate increment and decrement methods. The operations " - " can be used to autogenerate missing methods for unary minus or **abs** .

*** Comparison operations**

<", <=", ">", ">=", "==", "!=", <=>", "lt", "le", "gt", "ge", "eq", "ne", "cmp",

If the corresponding "spaceship" variant is available, it can be used to substitute for the missing operation. During **sort** ing arrays, **cmp** is used to compare values subject to **%OVERLOAD** .

*** *Bit operations***

"&", "^", "|", "&=", "^=", "|=", "neg", "!", "~",

" **neg** " stands for unary minus. If the method for **neg** is not specified, it can be autogenerated using on the method for subtraction.

*** Increment and decrement**

"++", "--",

If undefined, addition and subtraction methods can be used instead. These operations are called both in prefix and postfix form.

**** Transcendental functions***

"atan2", "cos", "sin", "exp", "abs", "log", "sqrt",

If **abs** is unavailable, it can be autogenerated using methods for <" or <=>" combined with either unary minus or subtraction.

**** Boolean, string and numeric conversion***

"bool", "\"\"", "0+",

If one or two of these operations are unavailable, the remaining ones can be used instead. **bool** is used in the flow control operators (like **while**) and for the ternary " **?:** " operation. These functions can return any arbitrary Perl value. If the corresponding operation for this value is overloaded too, that operation will be called again with this value.

**** Special***

"nomethod", "fallback", "=",

see *SPECIAL KEYS OF %OVERLOAD* .

See *Fallback* for an explanation of when a missing method can be autogenerated..

SPECIAL KEYS OF %OVERLOAD

Three keys are recognized by Perl that are not covered by the above description.

Last Resort

$OVERLOAD {"nomethod"} is a reference to a function of four parameters. If defined, it is called when the overloading mechanism cannot find a method for some operation. The first three arguments of this function coincide with arguments for the corresponding method if it were found, the fourth argument is the key of **%OVERLOAD** corresponding to the missing method. If several methods are tried, the last one is used. Say, **1- $a** can be equivalent to

&{ $Pack::OVERLOAD{"nomethod"} }($a,1,1,"-").

If some operation cannot be resolved, and there is no **$OVERLOAD {"nomethod"}** , then an exception will be raised via *die()* -- unless **$OVERLOAD {"fallback"}** is true.

Fallback

$OVERLOAD {"fallback"}** governs what to do if a method for a particular operation is not found. Three different cases are possible depending on value of **$OVERLOAD {"fallback"} :

*** undef**

Perl tries to use a substituted method (see *MAGIC AUTOGENERATION*). If this fails, it then tries to calls **$OVERLOAD {"nomethod"}** ; if missing, an exception will be raised.

*** TRUE**

The same as for the undef value, but no exception is raised. Instead, it silently reverts to what it would have done were there no **%OVERLOAD** is present.

*** defined, but FALSE**

No autogeneration is tried. Perl tries to call **$OVERLOAD {"nomethod"}** , and if this is missing, raises an exception.

.

Copy Constructor

$OVERLOAD {"="} is a reference to a function with three arguments, i.e., it looks like a usual value of **%OVERLOAD** . This operation is called in the situations when a mutator is applied to a reference that shares its object with some other reference, such as

$a=$b; $a++;

To make this change to **$a** and not to change **$b** , a freshly made copy of **$$ a** is made, and **$a** is assigned a reference to this object. This operation is executed during **$a ++** , (so before this **$$ a** coincides with **$$ b**), and only if ++ is expressed via **$OPERATOR {'++'}** or **$OPERATOR {'+='}** . Note that if this operation is expressed via ' + ', i.e., as

$a=$b; $a=$a+1;

then **$$ a** and **$$ b** do not appear as lvalues.

If the copy constructor is required during execution of some mutator, but **$OPERATOR {'='}** is missing, it can be autogenerated as a string copy if an object of the package is a plain scalar.

MAGIC AUTOGENERATION

If a method for an operation is not found, and **$OVERLOAD {"fallback"}** is TRUE or undefined, Perl tries to to autogenerate a substitute method for the missing operation based on defined operations. Autogenerated method substitutions are possible for the following operations:

Assignment forms of arithmetic operations

$a =+ $b can use the **$OVERLOAD {"+"}** method if **$OVERLOAD {"+="}** is not defined.

Conversion operations

String, numeric, and boolean conversion are calculated in terms of one another if not all of them are defined.

Increment and decrement

The **++ $a** operation can be expressed in terms of **$a +=1** or **$a +1** , and **$a --** in terms of **$a -=1** and **$a -1** .

abs($a)

can be expressed in terms of **$a <0** and **- $a** (or **0- $a**).

Unary minus

can be expressed in terms of subtraction.

Concatenation

can be expressed in terms of string conversion.

Comparison operations

can be expressed in terms of its "spaceship" counterpart: either **<=>** or **cmp** : <, >,<=, >=, ==, != in terms of<=> lt, gt, le, ge, eq, ne in terms of cmp

Copy operator

can be expressed in terms of assignment to the dereferenced value, if this value is scalar but not a reference.

.

WARNING

The restriction for the comparison operation is that even if, for example, ` **cmp** ' should return a reference to a blessed object, the autogenerated ` **lt** ' function will produce only a standard logical value based on the numerical value of the result of ` **cmp** '. In particular, a working numeric conversion is needed in this case (possibly expressed in terms of other conversions).

Similarly, **.=** and **x=** operators lose their mathemagical properties if the string conversion substitution is applied.

When you *chop()* a mathemagical object, it becomes promoted to a string first, and its mathemagical

qualities is lost. The same can happen with other operations as well.

TOC

IMPLEMENTATION

The table of methods for all operations is cached as a magic for the symbol table hash of the package. It is rechecked for changes of **%OVERLOAD** and **@ISA** only during **bless** ing; so if it is changed dynamically, you'll need an additional fake **bless** ing to update the table.

(Every SVish thing has a magic queue, and a magic is an entry in that queue. This is how a single variable may participate in multiple forms of magic simultaneously. For instance, environment variables regularly have two forms at once: their **%ENV** magic and their taint magic.)

If an object belongs to a package with **%OVERLOAD** , it carries a special flag. Thus the only speed penalty during arithmetic operations without overload is the check of this flag.

In fact, if no **%OVERLOAD** is ever accessed, there is almost no overhead for overloadable operations, so most programs should not suffer measurable performance penalties. Considerable effort was made minimize overhead when **%OVERLOAD** is accessed and the current operation is overloadable but the arguments in question do not belong to packages with **%OVERLOAD** . When in doubt, test your speed with **%OVERLOAD** and without it. So far there have been no reports of substantial speed degradation if Perl is compiled with optimization turned on.

There is no size penalty for data if there is no **%OVERLOAD** .

The copying like **$a = $b** is shallow; however, a one-level-deep copying is carried out before any operation that can imply an assignment to the object **$b** (or **$a**) refers to, like **$b ++** . You can override this behavior by defining your copy constructor (see *Copy Constructor*).

It is expected that arguments to methods that are not explicitly supposed to be changed are constant (but this is not enforced).

AUTHOR

Ilya Zakharevich< *ilya* ***@math*** *.mps.ohio-state.edu* >.

DIAGNOSTICS

When Perl is run with the **-Do** switch or its equivalent, overloading induces diagnostic messages.

BUGS

Because it's used for overloading, the per-package associative array **%OVERLOAD** now has a special meaning in Perl.

As shipped, **%OVERLOAD** is not inherited via the **@ISA** tree. A patch for this is available from the author.

This document is confusing.

TOC

NAME

perlbook - Perl book information

DESCRIPTION

You can order Perl books from O'Reilly & Associates, 1-800-998-9938. Local/overseas is +1 707 829 0515. If you can locate an O'Reilly order form, you can also fax to +1 707 829 0104. *Programming Perl* is a reference work that covers nearly all of Perl (version 4, alas), while *Learning Perl* is a tutorial that covers the most frequently used subset of the language.

Programming Perl (the Camel Book): ISBN 0-937175-64-1 (English) ISBN 4-89052-384-7 (Japanese)
Learning Perl (the Llama Book): ISBN 1-56592-042-2 (English)

Python library reference

Guido van Rossum

Go to the next section.

Introduction

The "Python library" contains several different kinds of components.

It contains data types that would normally be considered part of the "core" of a language, such as numbers and lists. For these types, the Python language core defines the form of literals and places some constraints on their semantics, but does not fully define the semantics. (On the other hand, the language core does define syntactic properties like the spelling and priorities of operators.)

The library also contains built-in functions and exceptions --- objects that can be used by all Python code without the need of an `import` statement. Some of these are defined by the core language, but many are not essential for the core semantics and are only described here.

The bulk of the library, however, consists of a collection of modules. There are many ways to dissect this collection. Some modules are written in C and built in to the Python interpreter; others are written in Python and imported in source form. Some modules provide interfaces that are highly specific to Python, like printing a stack trace; some provide interfaces that are specific to particular operating systems, like socket I/O; others provide interfaces that are specific to a particular application domain, like the World-Wide Web. Some modules are avaiable in all versions and ports of Python; others are only available when the underlying system supports or requires them; yet others are available only when a particular configuration option was chosen at the time when Python was compiled and installed.

This manual is organized "from the inside out": it first describes the built-in data types, then the built-in functions and exceptions, and finally the modules, grouped in chapters of related modules. The ordering of the chapters as well as the ordering of the modules within each chapter is roughly from most relevant to least important.

This means that if you start reading this manual from the start, and skip to the next chapter when you get bored, you will get a reasonable overview of the available modules and application areas that are supported by the Python library. Of course, you don't *have* to read it like a novel -- you can also browse the table of contents (in front of the manual), or look for a specific function, module or term in the index (in the back). And finally, if you enjoy learning about random subjects, you choose a random page number (see module `rand`) and read a section or two.

Let the show begin!

Go to the next section.

Go to the previous, next section.

Built-in Types, Exceptions and Functions

Names for built-in exceptions and functions are found in a separate symbol table. This table is searched last when the interpreter looks up the meaning of a name, so local and global user-defined names can override built-in names. Built-in types are described together here for easy reference.(1)

The tables in this chapter document the priorities of operators by listing them in order of ascending priority (within a table) and grouping operators that have the same priority in the same box. Binary operators of the same priority group from left to right. (Unary operators group from right to left, but there you have no real choice.) See Chapter 5 of the Python Reference Manual for the complete picture on operator priorities.

Built-in Types

The following sections describe the standard types that are built into the interpreter. These are the numeric types, sequence types, and several others, including types themselves. There is no explicit Boolean type; use integers instead.

Some operations are supported by several object types; in particular, all objects can be compared, tested for truth value, and converted to a string (with the `` `...` `` notation). The latter conversion is implicitly used when an object is written by the `print` statement.

Truth Value Testing

Any object can be tested for truth value, for use in an `if` or `while` condition or as operand of the Boolean operations below. The following values are considered false:

- `None`
- zero of any numeric type, e.g., `0`, `0L`, `0.0`.
- any empty sequence, e.g., `''`, `()`, `[]`.
- any empty mapping, e.g., `{}`.
- instances of user-defined classes, if the class defines a `__nonzero__()` or `__len__()` method, when that method returns zero.

All other values are considered true -- so objects of many types are always true.

Operations and built-in functions that have a Boolean result always return `0` for false and `1` for true, unless otherwise stated. (Important exception: the Boolean operations `or' and `and' always return one of their operands.)

Boolean Operations

These are the Boolean operations, ordered by ascending priority:

Operation

Result -- *Notes*

`x or y`

if x is false, then y, else x -- (1) @hline

`x and y`

if x is false, then x, else y -- (1) @hline

`not x`

if x is false, then `1`, else `0` -- (2)

Notes:

(1)

These only evaluate their second argument if needed for their outcome.

(2)

`not' has a lower priority than non-Boolean operators, so e.g. `not a == b` is interpreted as `not(a == b)`, and `a == not b` is a syntax error.

Comparisons

Comparison operations are supported by all objects. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily, e.g. `x < y <= z` is equivalent to `x < y and y <= z`, except that `y` is evaluated only once (but in both cases `z` is not evaluated at all when `x < y` is found to be false).

This table summarizes the comparison operations:

Operation

Meaning -- *Notes*

`<`

strictly less than

`<=`

less than or equal

`>`

strictly greater than

`>=`

greater than or equal

`==`

equal

`<>`

not equal -- (1)

`!=`

not equal -- (1)

`is`

object identity

`is not`

negated object identity

Notes:

(1)

<> and `!=` are alternate spellings for the same operator. (I couldn't choose between ABC and C! :-)

Objects of different types, except different numeric types, never compare equal; such objects are ordered consistently but arbitrarily (so that sorting a heterogeneous array yields a consistent result). Furthermore, some types (e.g., windows) support only a degenerate notion of comparison where any two objects of that type are unequal. Again, such objects are ordered arbitrarily but consistently.

(Implementation note: objects of different types except numbers are ordered by their type names; objects of the same types that don't support proper comparison are ordered by their address.)

Two more operations with the same syntactic priority, `in` and `not in`, are supported only by sequence types (below).

Numeric Types

There are three numeric types: plain integers, long integers, and floating point numbers. Plain integers (also just called integers) are implemented using `long` in C, which gives them at least 32 bits of precision. Long integers have unlimited precision. Floating point numbers are implemented using `double` in C. All bets on their precision are off unless you happen to know the machine you are working with.

Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals (including hex and octal numbers) yield plain integers. Integer literals with an `L' or `l' suffix yield long integers (`L' is preferred because `1l` looks too much like eleven!). Numeric literals containing a decimal point or an exponent sign yield floating point numbers.

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the "smaller" type is converted to that of the other, where plain integer is smaller than long integer is smaller than floating point. Comparisons between numbers of mixed type use the same rule.(2) The functions `int()`, `long()` and `float()` can be used to coerce numbers to a specific type.

All numeric types support the following operations, sorted by ascending priority (operations in the same box have the same priority; all numeric operations have a higher priority than comparison operations):

Operation
: *Result -- Notes*

`x + y`
: sum of x and y

`x - y`
: difference of x and y @hline

`x * y`
: product of x and y

`x / y`
: quotient of x and y -- (1)

`x % y`
: remainder of `x / y` @hline

`-x`
: x negated

`+x`
: x unchanged @hline

`abs(x)`
: absolute value of x

`int(x)`
: x converted to integer -- (2)

`long(x)`
: x converted to long integer -- (2)

`float(x)`
: x converted to floating point

`divmod(x, y)`
: the pair `(x / y, x % y)` -- (3)

`pow(x, y)`
: x to the power y

Notes:

(1)
: For (plain or long) integer division, the result is an integer; it always truncates towards zero.

(2)
: Conversion from floating point to (long or plain) integer may round or truncate as in C; see functions `floor()` and `ceil()` in module `math` for well-defined conversions.

(3)
: See the section on built-in functions for an exact definition.

Bit-string Operations on Integer Types

Plain and long integer types support additional operations that make sense only for bit-strings. Negative numbers are treated as their 2's complement value (for long integers, this assumes a sufficiently large number of bits that no overflow occurs during the operation).

The priorities of the binary bit-wise operations are all lower than the numeric operations and higher than the comparisons; the unary operation `~' has the same priority as the other unary numeric operations (`+' and `-').

This table lists the bit-string operations sorted in ascending priority (operations in the same box have the same priority):

Operation

Result -- Notes

`x | y`

bitwise or of x and y @hline

`x ^ y`

bitwise exclusive or of x and y @hline

`x & y`

bitwise and of x and y @hline

`x << n`

x shifted left by n bits -- (1), (2)

`x >> n`

x shifted right by n bits -- (1), (3) @hline@hline

`~x`

the bits of x inverted

Notes:

(1)

Negative shift counts are illegal.

(2)

A left shift by n bits is equivalent to multiplication by `pow(2, n)` without overflow check.

(3)

A right shift by n bits is equivalent to division by `pow(2, n)` without overflow check.

Sequence Types

There are three sequence types: strings, lists and tuples.

Strings literals are written in single or double quotes: `'xyzzy'`, `"frobozz"`. See Chapter 2 of the Python Reference Manual for more about string literals. Lists are constructed with square brackets, separating items with commas: `[a, b, c]`. Tuples are constructed by the comma operator (not within

square brackets), with or without enclosing parentheses, but an empty tuple must have the enclosing parentheses, e.g., `a, b, c` or `()`. A single item tuple must have a trailing comma, e.g., `(d,)`.

Sequence types support the following operations. The `in' and `not@,in' operations have the same priorities as the comparison operations. The `+' and `*' operations have the same priority as the corresponding numeric operations.(3)

This table lists the sequence operations sorted in ascending priority (operations in the same box have the same priority). In the table, s and t are sequences of the same type; n, i and j are integers:

Operation

Result -- Notes

`x in s`

`1` if an item of s is equal to x, else `0`

`x not in s`

`0` if an item of s is equal to x, else `1` @hline

`s + t`

the concatenation of s and t @hline

`s * n, n * s`

n copies of s concatenated @hline

`s[i]`

i'th item of s, origin 0 -- (1)

`s[i:j]`

slice of s from i to j -- (1), (2) @hline

`len(s)`

length of s

`min(s)`

smallest item of s

`max(s)`

largest item of s

Notes:

(1)

If i or j is negative, the index is relative to the end of the string, i.e., `len(s) + i` or `len(s) + j` is substituted. But note that `-0` is still `0`.

(2)

The slice of s from i to j is defined as the sequence of items with index k such that `i <= k < j`. If i or j is greater than `len(s)`, use `len(s)`. If i is omitted, use `0`. If j is omitted, use `len(s)`. If i is greater than or equal to j, the slice is empty.

More String Operations

String objects have one unique built-in operation: the % operator (modulo) with a string left argument interprets this string as a C sprintf format string to be applied to the right argument, and returns the string resulting from this formatting operation.

The right argument should be a tuple with one item for each argument required by the format string; if the string requires a single argument, the right argument may also be a single non-tuple object.(4) The following format characters are understood: %, c, s, i, d, u, o, x, X, e, E, f, g, G. Width and precision may be a * to specify that an integer argument specifies the actual width or precision. The flag characters -, +, blank, # and 0 are understood. The size specifiers h, l or L may be present but are ignored. The `%s` conversion takes any Python object and converts it to a string using `str()` before formatting it. The ANSI features `%p` and `%n` are not supported. Since Python strings have an explicit length, `%s` conversions don't assume that `'\0'` is the end of the string.

For safety reasons, floating point precisions are clipped to 50; `%f` conversions for numbers whose absolute value is over 1e25 are replaced by `%g` conversions.(5) All other errors raise exceptions.

If the right argument is a dictionary (or any kind of mapping), then the formats in the string must have a parenthesized key into that dictionary inserted immediately after the % character, and each format formats the corresponding entry from the mapping. E.g.

```
>>> count = 2
>>> language = 'Python'
>>> print '%(language)s has %(count)03d quote types.' % vars()
Python has 002 quote types.
>>>
```

In this case no * specifiers may occur in a format (since they a require sequential parameter list).

Additional string operations are defined in standard module `string` and in built-in module `regex`.

Mutable Sequence Types

List objects support additional operations that allow in-place modification of the object. These operations would be supported by other mutable sequence types (when added to the language) as well. Strings and tuples are immutable sequence types and such objects cannot be modified once created. The following operations are defined on mutable sequence types (where x is an arbitrary object):

Operation

Result -- Notes

`s[i] = x`

item i of s is replaced by x

`s[i:j] = t`

slice of s from i to j is replaced by t

`del s[i:j]`

same as `s[i:j] = []`

`s.append(x)`

same as `s[len(s):len(s)] = [x]`

`s.count(x)`

return number of i's for which `s[i] == x`

`s.index(x)`

return smallest i such that `s[i] == x` -- (1)

`s.insert(i, x)`

same as `s[i:i] = [x]`

`s.remove(x)`

same as `del s[s.index(x)]` -- (1)

`s.reverse()`

reverses the items of s in place

`s.sort()`

permutes the items of s to satisfy `s[i] <= s[j]`, for `i < j` -- (2)

Notes:

(1)

Raises an exception when x is not found in s.

(2)

The `sort()` method takes an optional argument specifying a comparison function of two arguments (list items) which should return `-1`, `0` or `1` depending on whether the first argument is considered smaller than, equal to, or larger than the second argument. Note that this slows the sorting process down considerably; e.g. to sort a list in reverse order it is much faster to use calls to `sort()` and `reverse()` than to use `sort()` with a comparison function that reverses the ordering of the elements.

Mapping Types

A mapping object maps values of one type (the key type) to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the dictionary. A dictionary's keys are almost arbitrary values. The only types of values not acceptable as keys are values containing lists or dictionaries or other mutable types that are compared by value rather than by object identity. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (e.g. 1 and 1.0) then they can be used interchangeably to index the same dictionary entry.

Dictionaries are created by placing a comma-separated list of `key:@,var{value}` pairs within braces, for example: `{'jack':@,4098, 'sjoerd':@,4127}` or `{4098:@,'jack', 4127:@,'sjoerd'}`.

The following operations are defined on mappings (where a is a mapping, k is a key and x is an arbitrary object):

Operation

Result -- *Notes*

`len(a)`
: the number of items in a

`a[k]`
: the item of a with key k -- (1)

`a[k] = x`
: set `a[k]` to x

`del a[k]`
: remove `a[k]` from a -- (1)

`a.items()`
: a copy of a's list of (key, item) pairs -- (2)

`a.keys()`
: a copy of a's list of keys -- (2)

`a.values()`
: a copy of a's list of values -- (2)

`a.has_key(k)`
: `1` if a has a key k, else `0`

Notes:

(1)
: Raises an exception if k is not in the map.

(2)
: Keys and values are listed in random order.

Other Built-in Types

The interpreter supports several other kinds of objects. Most of these support only one or two operations.

Modules

The only special operation on a module is attribute access: `m.name`, where m is a module and name accesses a name defined in m's symbol table. Module attributes can be assigned to. (Note that the `import` statement is not, strictly spoken, an operation on a module object; `import foo` does not require a module object named foo to exist, rather it requires an (external) *definition* for a module named foo somewhere.)

A special member of every module is `__dict__`. This is the dictionary containing the module's symbol table. Modifying this dictionary will actually change the module's symbol table, but direct assignment to the `__dict__` attribute is not possible (i.e., you can write `m.__dict__['a'] = 1`, which defines `m.a` to be `1`, but you can't write `m.__dict__ = {}`.

Modules are written like this: `<module 'sys'>`.

Classes and Class Instances

(See Chapters 3 and 7 of the Python Reference Manual for these.)

Functions

Function objects are created by function definitions. The only operation on a function object is to call it: `func(argument-list)`.

There are really two flavors of function objects: built-in functions and user-defined functions. Both support the same operation (to call the function), but the implementation is different, hence the different object types.

The implementation adds two special read-only attributes: `f.func_code` is a function's code object (see below) and `f.func_globals` is the dictionary used as the function's global name space (this is the same as `m.__dict__` where m is the module in which the function f was defined).

Methods

Methods are functions that are called using the attribute notation. There are two flavors: built-in methods (such as `append()` on lists) and class instance methods. Built-in methods are described with the types that support them.

The implementation adds two special read-only attributes to class instance methods: `m.im_self` is the object whose method this is, and `m.im_func` is the function implementing the method. Calling `m(arg-1, arg-2, ..., arg-n)` is completely equivalent to calling `m.im_func(m.im_self, arg-1, arg-2, ..., arg-n)`.

(See the Python Reference Manual for more info.)

Code Objects

Code objects are used by the implementation to represent "pseudo-compiled" executable Python code such as a function body. They differ from function objects because they don't contain a reference to their global execution environment. Code objects are returned by the built-in `compile()` function and can be extracted from function objects through their `func_code` attribute.

A code object can be executed or evaluated by passing it (instead of a source string) to the `exec` statement or the built-in `eval()` function.

(See the Python Reference Manual for more info.)

Type Objects

Type objects represent the various object types. An object's type is accessed by the built-in function `type()`. There are no special operations on types. The standard module `types` defines names for all standard built-in types.

Types are written like this: `<type 'int'>`.

The Null Object

This object is returned by functions that don't explicitly return a value. It supports no special operations. There is exactly one null object, named `None` (a built-in name).

It is written as `None`.

File Objects

File objects are implemented using C's `stdio` package and can be created with the built-in function `open()` described under Built-in Functions below. They are also returned by some other built-in functions and methods, e.g. `posix.popen()` and `posix.fdopen()` and the `makefile()` method of socket objects.

When a file operation fails for an I/O-related reason, the exception `IOError` is raised. This includes situations where the operation is not defined for some reason, like `seek()` on a tty device or writing a file opened for reading.

Files have the following methods:

Method: file **close** *()*

Close the file. A closed file cannot be read or written anymore.

Method: file **flush** *()*

Flush the internal buffer, like `stdio`'s `fflush()`.

Method: file **isatty** *()*

Return `1` if the file is connected to a tty(-like) device, else `0`.

Method: file **read** *(size)*

Read at most size bytes from the file (less if the read hits EOF or no more data is immediately available on a pipe, tty or similar device). If the size argument is omitted, read all data until EOF is reached. The bytes are returned as a string object. An empty string is returned when EOF is encountered immediately. (For certain files, like ttys, it makes sense to continue reading after an EOF is hit.)

Method: file **readline** *()*

Read one entire line from the file. A trailing newline character is kept in the string(6) (but may be absent when a file ends with an incomplete line). An empty string is returned when EOF is hit immediately. Note: unlike `stdio`'s `fgets()`, the returned string contains null characters (`'\0'`) if they occurred in the input.

Method: file **readlines** *()*

Read until EOF using `readline()` and return a list containing the lines thus read.

Method: file **seek** *(offset, whence)*

Set the file's current position, like `stdio`'s `fseek()`. The whence argument is optional and default `0` (absolute file positioning); other values are `1` (seek relative to the current position) and `2` (seek relative to the file's end). There is no return value.

Method: file **tell** *()*

Return the file's current position, like `stdio`'s `ftell()`.

Method: file **write** *(str)*

Write a string to the file. There is no return value.

Method: file **writelines** *(list)*

Write a list of strings to the file. There is no return value. (The name is intended to match `readlines`; `writelines` does not add line separators.)

Internal Objects

(See the Python Reference Manual for these.)

Special Attributes

The implementation adds a few special read-only attributes to several object types, where they are relevant:

- `x.__dict__` is a dictionary of some sort used to store an object's (writable) attributes;
- `x.__methods__` lists the methods of many built-in object types, e.g., `[].__methods__` yields `['append', 'count', 'index', 'insert', 'remove', 'reverse', 'sort']`;
- `x.__members__` lists data attributes;
- `x.__class__` is the class to which a class instance belongs;
- `x.__bases__` is the tuple of base classes of a class object.

Built-in Exceptions

Exceptions are string objects. Two distinct string objects with the same value are different exceptions. This is done to force programmers to use exception names rather than their string value when specifying exception handlers. The string value of all built-in exceptions is their name, but this is not a requirement for user-defined exceptions or exceptions defined by library modules.

The following exceptions can be generated by the interpreter or built-in functions. Except where mentioned, they have an `associated value' indicating the detailed cause of the error. This may be a string or a tuple containing several items of information (e.g., an error code and a string explaining the code).

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition `just like' the situation in which the interpreter raises the same exception; but beware that there

is nothing to prevent user code from raising an inappropriate error.

built-in exception: **AttributeError**

Raised when an attribute reference or assignment fails. (When an object does not support attribute references or attribute assignments at all, `TypeError` is raised.)

built-in exception: **EOFError**

Raised when one of the built-in functions (`input()` or `raw_input()`) hits an end-of-file condition (EOF) without reading any data. (N.B.: the `read()` and `readline()` methods of file objects return an empty string when they hit EOF.) No associated value.

built-in exception: **IOError**

Raised when an I/O operation (such as a `print` statement, the built-in `open()` function or a method of a file object) fails for an I/O-related reason, e.g., `file not found', `disk full'.

built-in exception: **ImportError**

Raised when an `import` statement fails to find the module definition or when a `from ... import` fails to find a name that is to be imported.

built-in exception: **IndexError**

Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not a plain integer, `TypeError` is raised.)

built-in exception: **KeyError**

Raised when a mapping (dictionary) key is not found in the set of existing keys.

built-in exception: **KeyboardInterrupt**

Raised when the user hits the interrupt key (normally Control-C or DEL). During execution, a check for interrupts is made regularly. Interrupts typed when a built-in function `input()` or `raw_input()`) is waiting for input also raise this exception. No associated value.

built-in exception: **MemoryError**

Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating what kind of (internal) operation ran out of memory. Note that because of the underlying memory management architecture (C's `malloc()` function), the interpreter may not always be able to completely recover from this situation; it nevertheless raises an exception so that a stack traceback can be printed, in case a run-away program was the cause.

built-in exception: **NameError**

Raised when a local or global name is not found. This applies only to unqualified names. The associated value is the name that could not be found.

built-in exception: **OverflowError**

Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for long integers (which would rather raise `MemoryError` than give up). Because of the lack of standardization of floating point exception handling in C, most floating point operations also aren't checked. For plain integers, all operations that can overflow are checked except left shift, where typical applications prefer to drop bits than raise an exception.

built-in exception: **RuntimeError**

Raised when an error is detected that doesn't fall in any of the other categories. The associated value is a string indicating what precisely went wrong. (This exception is a relic from a previous version of the interpreter; it is not used any more except by some extension modules that haven't been converted to define their own exceptions yet.)

built-in exception: **SyntaxError**

Raised when the parser encounters a syntax error. This may occur in an `import` statement, in an `exec` statement, in a call to the built-in function `eval()` or `input()`, or when reading the initial script or standard input (also interactively).

built-in exception: **SystemError**

Raised when the interpreter finds an internal error, but the situation does not look so serious to cause it to abandon all hope. The associated value is a string indicating what went wrong (in low-level terms).

You should report this to the author or maintainer of your Python interpreter. Be sure to report the version string of the Python interpreter (`sys.version`; it is also printed at the start of an interactive Python session), the exact error message (the exception's associated value) and if possible the source of the program that triggered the error.

built-in exception: **SystemExit**

This exception is raised by the `sys.exit()` function. When it is not handled, the Python interpreter exits; no stack traceback is printed. If the associated value is a plain integer, it specifies the system exit status (passed to C's `exit()` function); if it is `None`, the exit status is zero; if it has another type (such as a string), the object's value is printed and the exit status is one.

A call to `sys.exit` is translated into an exception so that clean-up handlers (`finally` clauses of `try` statements) can be executed, and so that a debugger can execute a script without running the risk of losing control. The `posix._exit()` function can be used if it is absolutely positively necessary to exit immediately (e.g., after a `fork()` in the child process).

built-in exception: **TypeError**

Raised when a built-in operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

built-in exception: **ValueError**

Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as `IndexError`.

built-in exception: **ZeroDivisionError**

Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

Built-in Functions

The Python interpreter has a number of functions built into it that are always available. They are listed here in alphabetical order.

built-in function: **abs** *(x)*

Return the absolute value of a number. The argument may be a plain or long integer or a floating point number.

built-in function: **apply** *(function, args)*

The function argument must be a callable object (a user-defined or built-in function or method, or a class object) and the args argument must be a tuple. The function is called with args as argument list; the number of arguments is the the length of the tuple. (This is different from just calling `func(args)`, since in that case there is always exactly one argument.)

built-in function: **chr** *(i)*

Return a string of one character whose ASCII code is the integer i, e.g., `chr(97)` returns the string `'a'`. This is the inverse of `ord()`. The argument must be in the range [0..255], inclusive.

built-in function: **cmp** *(x, y)*

Compare the two objects x and y and return an integer according to the outcome. The return value is negative if `x < y`, zero if `x == y` and strictly positive if `x > y`.

built-in function: **coerce** *(x, y)*

Return a tuple consisting of the two numeric arguments converted to a common type, using the same rules as used by arithmetic operations.

built-in function: **compile** *(string, filename, kind)*

Compile the string into a code object. Code objects can be executed by an `exec` statement or evaluated by a call to `eval()`. The filename argument should give the file from which the code was read; pass e.g. `'<string>'` if it wasn't read from a file. The kind argument specifies what kind of code must be compiled; it can be `'exec'` if string consists of a sequence of statements, or `'eval'` if it consists of a single expression.

built-in function: **delattr** *(object, name)*

This is a relative of `setattr`. The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, `delattr(x, 'foobar')` is equivalent to `del x.foobar`.

built-in function: **dir** *()*

Without arguments, return the list of names in the current local symbol table. With a module, class or class instance object as argument (or anything else that has a `__dict__` attribute), returns the list of names in that object's attribute dictionary. The resulting list is sorted. For example:

```
>>> import sys
>>> dir()
['sys']
>>> dir(sys)
['argv', 'exit', 'modules', 'path', 'stderr', 'stdin', 'stdout']
>>>
```

built-in function: **divmod** *(a, b)*

Take two numbers as arguments and return a pair of integers consisting of their integer quotient and remainder. With mixed operand types, the rules for binary arithmetic operators apply. For plain and long integers, the result is the same as `(a / b, a % b)`. For floating point numbers the result is the same as `(math.floor(a / b), a % b)`.

built-in function: **eval** *(expression[, globals[, locals]])*

The arguments are a string and two optional dictionaries. The expression argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the globals and locals dictionaries as global and local name space. If the locals dictionary is omitted it defaults to the globals dictionary. If both dictionaries are omitted, the expression is executed in the environment where `eval` is called. The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```
>>> x = 1
>>> print eval('x+1')
2
>>>
```

This function can also be used to execute arbitrary code objects (e.g. created by `compile()`). In this case pass a code object instead of a string. The code object must have been compiled passing `'eval'` to the kind argument.

Hints: dynamic execution of statements is supported by the `exec` statement. Execution of statements from a file is supported by the `execfile()` function. The `vars()` function returns the current local dictionary, which may be useful to pass around for use by `eval()` or `execfile()`.

built-in function: **execfile** *(file[, globals[, locals]])*

This function is similar to the `exec` statement, but parses a file instead of a string. It is different from the `import` statement in that it does not use the module administration -- it reads the file unconditionally and does not create a new module.(7)

The arguments are a file name and two optional dictionaries. The file is parsed and evaluated as a sequence of Python statements (similarly to a module) using the globals and locals dictionaries as gl and local name space. If the locals dictionary is omitted it defaults to the globals dictionary. If both dictionaries are omitted, the expression is executed in the environment where `execfile()` is called. The return value is `None`.

built-in function: **filter** *(function, list)*

Construct a list from those elements of list for which function returns true. If list is a string or a tuple, the result also has that type; otherwise it is always a list. If function is `None`, the identity function is assumed, i.e. all elements of list that are false (zero or empty) are removed.

built-in function: **float** *(x)*

Convert a number to floating point. The argument may be a plain or long integer or a floating point number.

built-in function: **getattr** *(object, name)*

The arguments are an object and a string. The string must be the name of one of the object's attributes. The result is the value of that attribute. For example, `getattr(x, 'foobar')` is equivalent to `x.foobar`.

built-in function: **hasattr** *(object, name)*

The arguments are an object and a string. The result is 1 if the string is the name of one of the object's attributes, 0 if not. (This is implemented by calling `getattr(object, name)` and seeing whether it raises an exception or not.)

built-in function: **hash** *(object)*

Return the hash value of the object (if it has one). Hash values are 32-bit integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, e.g. 1 and 1.0).

built-in function: **hex** *(x)*

Convert an integer number (of any size) to a hexadecimal string. The result is a valid Python expression.

built-in function: **id** *(object)*

Return the `identity' of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. (Two objects whose lifetimes are disjunct may have the same id() value.) (Implementation note: this is the address of the object.)

built-in function: **input** *([prompt])*

Almost equivalent to `eval(raw_input(prompt))`. Like `raw_input()`, the prompt argument is optional. The difference is that a long input expression may be broken over multiple lines using the backslash convention.

built-in function: **int** *(x)*

Convert a number to a plain integer. The argument may be a plain or long integer or a floating point number. Conversion of floating point numbers to integers is defined by the C semantics; normally the conversion truncates towards zero.(8)

built-in function: **len** *(s)*

Return the length (the number of items) of an object. The argument may be a sequence (string, tuple or list) or a mapping (dictionary).

built-in function: **long** *(x)*

Convert a number to a long integer. The argument may be a plain or long integer or a floating point number.

built-in function: **map** *(function, list, ...)*

Apply function to every item of list and return a list of the results. If additional list arguments are passed, function must take that many arguments and is applied to the items of all lists in parallel; if a list is shorter than another it is assumed to be extended with `None` items. If function is `None`, the identity function is assumed; if there are multiple list arguments, `map` returns a list consisting of tuples containing the corresponding items from all lists (i.e. a kind of transpose operation). The list arguments may be any kind of sequence; the result is always a list.

built-in function: **max** *(s)*

Return the largest item of a non-empty sequence (string, tuple or list).

built-in function: **min** *(s)*

Return the smallest item of a non-empty sequence (string, tuple or list).

built-in function: **oct** *(x)*

Convert an integer number (of any size) to an octal string. The result is a valid Python expression.

built-in function: **open** *(filename[, mode[, bufsize]])*

Return a new file object (described earlier under Built-in Types). The first two arguments are the same as for `stdio`'s `fopen()`: filename is the file name to be opened, mode indicates how the file is to be opened: `'r'` for reading, `'w'` for writing (truncating an existing file), and `'a'` opens it for appending. Modes `'r+'`, `'w+'` and `'a+'` open the file for updating, provided the underlying `stdio` library understands this. On systems that differentiate between binary and text files, `'b'` appended to the mode opens the file in binary mode. If the file cannot be opened, `IOError` is raised. If mode is omitted, it defaults to `'r'`. The optional bufsize argument specifies the file's desired buffer size: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size. A negative bufsize means to use the system default, which is usually line buffered for for tty devices and fully buffered for other files.(9)

built-in function: **ord** *(c)*

Return the ASCII value of a string of one character. E.g., `ord('a')` returns the integer `97`. This is the

inverse of `chr()`.

built-in function: **pow** *(x, y[, z])*

Return x to the power y; if z is present, return x to the power y, modulo z (computed more efficiently than `pow(x, y) % z`). The arguments must have numeric types. With mixed operand types, the rules for binary arithmetic operators apply. The effective operand type is also the type of the result; if the result is not expressible in this type, the function raises an exception; e.g., `pow(2, -1)` or `pow(2, 35000)` is not allowed.

built-in function: **range** *([start,] end[, step])*

This is a versatile function to create lists containing arithmetic progressions. It is most often used in `for` loops. The arguments must be plain integers. If the step argument is omitted, it defaults to `1`. If the start argument is omitted, it defaults to `0`. The full form returns a list of plain integers `[start, start + step, start + 2 * step, ...]`. If step is positive, the last element is the largest `start + i * step` less than end; if step is negative, the last element is the largest `start + i * step` greater than end. step must not be zero (or else an exception is raised). Example:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
>>> range(1, 0)
[]
>>>
```

built-in function: **raw_input** *([prompt])*

If the prompt argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, `EOFError` is raised. Example:

```
>>> s = raw_input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
>>>
```

built-in function: **reduce** *(function, list[, initializer])*

Apply the binary function to the items of list so as to reduce the list to a single value. E.g., `reduce(lambda x, y: x*y, list, 1)` returns the product of the elements of list. The optional initializer can be thought of as being prepended to list so as to allow reduction of an empty list. The list arguments may be any kind of sequence.

built-in function: **reload** *(module)*

Re-parse and re-initialize an already imported module. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (i.e. the same as the module argument).

There are a number of caveats:

If a module is syntactically correct but its initialization fails, the first `import` statement for it does not bind its name locally, but does store a (partially initialized) module object in `sys.modules`. To reload the module you must first `import` it again (this will bind the name to the partially initialized module object) before you can `reload()` it.

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module's advantage if it maintains a global table or cache of objects --- with a `try` statement it can test for the table's presence and skip its initialization if desired.

It is legal though generally not very useful to reload built-in or dynamically loaded modules, except for `sys`, `__main__` and `__builtin__`. In certain cases, however, extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from` {...} `import` {...}, calling `reload()` for the other module does not redefine the objects imported from it -- one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (module.name) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances -- they continue to use the old class definition. The same is true for derived classes.

built-in function: **repr** *(object)*

Return a string containing a printable representation of an object. This is the same value yielded by conversions (reverse quotes). It is sometimes useful to be able to access this operation as an ordinary function. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `eval()`.

built-in function: **round** *(x, n)*

Return the floating point value x rounded to n digits after the decimal point. If n is omitted, it defaults to zero. The result is a floating point number. Values are rounded to the closest multiple of 10 to the power minus n; if two multiples are equally close, rounding is done away from 0 (so e.g. `round(0.5)` is `1.0`

and `round(-0.5)` is `-1.0`).

built-in function: **setattr** *(object, name, value)*

This is the counterpart of `getattr`. The arguments are an object, a string and an arbitrary value. The string must be the name of one of the object's attributes. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is equivalent to `x.foobar = 123`.

built-in function: **str** *(object)*

Return a string containing a nicely printable representation of an object. For strings, this returns the string itself. The difference with `repr(object)` is that `str(object)` does not always attempt to return a string that is acceptable to `eval()`; its goal is to return a printable string.

built-in function: **tuple** *(sequence)*

Return a tuple whose items are the same and in the same order as sequence's items. If sequence is alread a tuple, it is returned unchanged. For instance, `tuple('abc')` returns returns `('a', 'b', 'c')` and `tuple([1, 2, 3])` returns `(1, 2, 3)`.

built-in function: **type** *(object)*

Return the type of an object. The return value is a type object. The standard module `types` defines names for all built-in types. For instance:

```
>>> import types
>>> if type(x) == types.StringType: print "It's a string"
```

built-in function: **vars** *([object])*

Without arguments, return a dictionary corresponding to the current local symbol table. With a module, class or class instance object as argument (or anything else that has a `__dict__` attribute), returns a dictionary corresponding to the object's symbol table. The returned dictionary should not be modified: the effects on the corresponding symbol table are undefined.(10)

built-in function: **xrange** *([start,] end[, step])*

This function is very similar to `range()`, but returns an "xrange object" instead of a list. This is an opaque sequence type which yields the same values as the corresponding list, without actually storing them all simultaneously. The advantage of `xrange()` over `range()` is minimal (since `xrange()` still has to create the values when asked for them) except when a very large range is used on a memory-starved machine (e.g. MS-DOS) or when all of the range's elements are never used (e.g. when the loop is usually terminated with `break`).

Go to the previous, next section.

Go to the previous, next section.

TOC

Python Services

The modules described in this chapter provide a wide range of services related to the Python interpreter and its interaction with its environment. Here's an overview:

sys
--- Access system specific parameters and functions.

types
--- Names for all built-in types.

traceback
--- Print or retrieve a stack traceback.

pickle
--- Convert Python objects to streams of bytes and back.

shelve
--- Python object persistency.

copy
--- Shallow and deep copy operations.

marshal
--- Convert Python objects to streams of bytes and back (with different constraints).

imp
--- Access the implementation of the `import` statement.

__builtin__
--- The set of built-in functions.

__main__
--- The environment where the top-level script is run.

Built-in Module sys

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available.

data: module sys **argv**

The list of command line arguments passed to a Python script. `sys.argv[0]` is the script name (it is operating system dependent whether this is a full pathname or not). If the command was executed using the `-c' command line option to the interpreter, `sys.argv[0]` is set to the string `"-c"`. If no script name was passed to the Python interpreter, `sys.argv` has zero length.

data: module sys **builtin_module_names**

A list of strings giving the names of all modules that are compiled into this Python interpreter. (This information is not available in any other way -- `sys.modules.keys()` only lists the imported modules.)

data: module sys **exc_type**

data: module sys **exc_value**

data: module sys **exc_traceback**

These three variables are not always defined; they are set when an exception handler (an `except` clause of a `try` statement) is invoked. Their meaning is: `exc_type` gets the exception type of the exception being handled; `exc_value` gets the exception parameter (its associated value or the second argument to `raise`); `exc_traceback` gets a traceback object (see the Reference Manual) which encapsulates the call stack at the point where the exception originally occurred.

function of module sys: **exit** *(n)*

Exit from Python with numeric exit status n. This is implemented by raising the `SystemExit` exception, so cleanup actions specified by `finally` clauses of `try` statements are honored, and it is possible to catch the exit attempt at an outer level.

data: module sys **exitfunc**

This value is not actually defined by the module, but can be set by the user (or by a program) to specify a clean-up action at program exit. When set, it should be a parameterless function. This function will be called when the interpreter exits in any way (except not when a fatal error occurs: in that case the interpreter's internal state cannot be trusted).

data: module sys **last_type**

data: module sys **last_value**

data: module sys **last_traceback**

These three variables are not always defined; they are set when an exception is not handled and the interpreter prints an error message and a stack traceback. Their intended use is to allow an interactive user to import a debugger module and engage in post-mortem debugging without having to re-execute the command that caused the error (which may be hard to reproduce). The meaning of the variables is the same as that of `exc_type`, `exc_value` and `exc_tracaback`, respectively.

data: module sys **modules**

Gives the list of modules that have already been loaded. This can be manipulated to force reloading of modules and other tricks.

data: module sys **path**

A list of strings that specifies the search path for modules. Initialized from the environment variable `PYTHONPATH`, or an installation-dependent default.

data: module sys **ps1**

data: module sys **ps2**

Strings specifying the primary and secondary prompt of the interpreter. These are only defined if the interpreter is in interactive mode. Their initial values in this case are `'>>> '` and `'... '`.

function of module sys: **setcheckinterval** *(interval)*

Set the interpreter's "check interval". This integer value determines how often the interpreter checks for periodic things such as thread switches and signal handlers. The default is 10, meaning the check is performed every 10 Python virtual instructions. Setting it to a larger value may increase performance for programs using threads. Setting it to a value checks every virtual instruction, maximizing responsiveness as well as overhead.

function of module sys: **settrace** *(tracefunc)*

Set the system's trace function, which allows you to implement a Python source code debugger in Python. See section "How It Works" in the chapter on the Python Debugger.

function of module sys: **setprofile** *(profilefunc)*

Set the system's profile function, which allows you to implement a Python source code profiler in Python. See the chapter on the Python Profiler. The system's profile function is called similarly to the system's trace function (see `sys.settrace`), but it isn't called for each executed line of code (only on call and return and when an exception occurs). Also, its return value is not used, so it can just return `None`.

data: module sys **stdin**

data: module sys **stdout**

data: module sys **stderr**

File objects corresponding to the interpreter's standard input, output and error streams. `sys.stdin` is used for all interpreter input except for scripts but including calls to `input()` and `raw_input()`. `sys.stdout` is used for the output of `print` and expression statements and for the prompts of `input()` and `raw_input()`. The interpreter's own prompts and (almost all of) its error messages go to `sys.stderr`. `sys.stdout` and `sys.stderr` needn't be built-in file objects: any object is acceptable as long as it has a `write` method that takes a string argument. (Changing these objects doesn't affect the standard I/O streams of processes executed by `popen()`, `system()` or the `exec*()` family of functions in the `os` module.)

data: module sys **tracebacklimit**

When this variable is set to an integer value, it determines the maximum number of levels of traceback information printed when an unhandled exception occurs. The default is 1000. When set to 0 or less, all traceback information is suppressed and only the exception type and value are printed.

Standard Module types

This module defines names for all object types that are used by the standard Python interpreter (but not for the types defined by various extension modules). It is safe to use "`from types import *`" --- the module does not export any other names besides the ones listed here. New names exported by future versions of this module will all end in `Type`.

Typical use is for functions that do different things depending on their argument types, like the following:

```
from types import *
def delete(list, item):
    if type(item) is IntType:
       del list[item]
    else:
       list.remove(item)
```

The module defines the following names:

data: module types **NoneType**

The type of `None`.

data: module types **TypeType**

The type of type objects (such as returned by `type()`).

data: module types **IntType**

The type of integers (e.g. `1`).

data: module types **LongType**

The type of long integers (e.g. `1L`).

data: module types **FloatType**

The type of floating point numbers (e.g. `1.0`).

data: module types **StringType**

The type of character strings (e.g. `'Spam'`).

data: module types **TupleType**

The type of tuples (e.g. `(1, 2, 3, 'Spam')`).

data: module types **ListType**

The type of lists (e.g. `[0, 1, 2, 3]`).

data: module types **DictType**

The type of dictionaries (e.g. `{'Bacon': 1, 'Ham': 0}`).

data: module types **DictionaryType**

An alternative name for `DictType`.

data: module types **FunctionType**

The type of user-defined functions and lambdas.

data: module types **LambdaType**

An alternative name for `FunctionType`.

data: module types **CodeType**

The type for code objects such as returned by `compile()`.

data: module types **ClassType**

The type of user-defined classes.

data: module types **InstanceType**

The type of instances of user-defined classes.

data: module types **MethodType**

The type of methods of user-defined class instances.

data: module types **UnboundMethodType**

An alternative name for `MethodType`.

data: module types **BuiltinFunctionType**

The type of built-in functions like `len` or `sys.exit`.

data: module types **BuiltinMethodType**

An alternative name for `BuiltinFunction`.

data: module types **ModuleType**

The type of modules.

data: module types **FileType**

The type of open file objects such as `sys.stdout`.

data: module types **XRangeType**

The type of range objects returned by `xrange()`.

data: module types **TracebackType**

The type of traceback objects such as found in `sys.exc_traceback`.

data: module types **FrameType**

The type of frame objects such as found in `tb.tb_frame` if `tb` is a traceback object.

Standard Module `traceback`

This module provides a standard interface to format and print stack traces of Python programs. It exactly mimics the behavior of the Python interpreter when it prints a stack trace. This is useful when you want to print stack traces under program control, e.g. in a "wrapper" around the interpreter.

The module uses traceback objects -- this is the object type that is stored in the variables `sys.exc_traceback` and `sys.last_traceback`.

The module defines the following functions:

function of module traceback: **print_tb** *(traceback[, limit])*

Print up to limit stack trace entries from traceback. If limit is omitted or `None`, all entries are printed.

function of module traceback: **extract_tb** *(traceback[, limit])*

Return a list of up to limit "pre-processed" stack trace entries extracted from traceback. It is useful for alternate formatting of stack traces. If limit is omitted or `None`, all entries are extracted. A "pre-processed" stack trace entry is a quadruple (filename, line number, function name, line text) representing the information that is usually printed for a stack trace. The line text is a string with leading and trailing whitespace stripped; if the source is not available it is `None`.

function of module traceback: **print_exception** *(type, value, traceback[, limit])*

Print exception information and up to limit stack trace entries from traceback. This differs from `print_tb` in the following ways: (1) if traceback is not `None`, it prints a header "`Traceback (innermost last):`"; (2) it prints the exception type and value after the stack trace; (3) if type is `SyntaxError` and value has the appropriate format, it prints the line where the syntax error occurred with a caret indication the approximate position of the error.

function of module traceback: **print_exc** *([limit])*

This is a shorthand for `print_exception(sys.exc_type, sys.exc_value, sys.exc_traceback, limit)`.

function of module traceback: **print_last** *([limit])*

This is a shorthand for `print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit)`.

Standard Module `pickle`

The `pickle` module implements a basic but powerful algorithm for "pickling" (a.k.a. serializing, marshalling or flattening) nearly arbitrary Python objects. This is the act of converting objects to a stream of bytes (and back: "unpickling"). This is a more primitive notion than persistency -- although `pickle` reads and writes file objects, it does not handle the issue of naming persistent objects, nor the (even more complicated) area of concurrent access to persistent objects. The `pickle` module can transform a complex object into a byte stream and it can transform the byte stream into an object with the same internal structure. The most obvious thing to do with these byte streams is to write them onto a file, but it is also conceivable to send them across a network or store them in a database. The module `shelve` provides a simple interface to pickle and unpickle objects on "dbm"-style database files.

Unlike the built-in module `marshal`, `pickle` handles the following correctly:

- recursive objects (objects containing references to themselves)
- object sharing (references to the same object in different places)
- user-defined classes and their instances

The data format used by `pickle` is Python-specific. This has the advantage that there are no restrictions imposed by external standards such as CORBA (which probably can't represent pointer sharing or recursive objects); however it means that non-Python programs may not be able to reconstruct pickled Python objects.

The `pickle` data format uses a printable ASCII representation. This is slightly more voluminous than a binary representation. However, small integers actually take *less* space when represented as minimal-size decimal strings than when represented as 32-bit binary numbers, and strings are only much longer if they contain many control characters or 8-bit characters. The big advantage of using printable ASCII (and of some other characteristics of `pickle`'s representation) is that for debugging or recovery purposes it is possible for a human to read the pickled file with a standard text editor. (I could have gone a step further and used a notation like S-expressions, but the parser (currently written in Python) would have been considerably more complicated and slower, and the files would probably have become much larger.)

The `pickle` module doesn't handle code objects, which the `marshal` module does. I suppose `pickle` could, and maybe it should, but there's probably no great need for it right now (as long as `marshal` continues to be used for reading and writing code objects), and at least this avoids the possibility of smuggling Trojan horses into a program.

For the benefit of persistency modules written using `pickle`, it supports the notion of a reference to an object outside the pickled data stream. Such objects are referenced by a name, which is an arbitrary string of printable ASCII characters. The resolution of such names is not defined by the `pickle` module -- the persistent object module will have to implement a method `persistent_load`. To write references to persistent objects, the persistent module must define a method `persistent_id` which returns either `None` or the persistent ID of the object.

There are some restrictions on the pickling of class instances.

First of all, the class must be defined at the top level in a module.

Next, it must normally be possible to create class instances by calling the class without arguments. If is undesirable, the class can define a method `__getinitargs__()`, which should return a *tuple* containing the arguments to be passed to the class constructor (`__init__()`).

Classes can further influence how their instances are pickled -- if the class defines the method `__getstate__()`, it is called and the return state is pickled as the contents for the instance, and if the class defines the method `__setstate__()`, it is called with the unpickled state. (Note that these methods can also be used to implement copying class instances.) If there is no `__getstate__()` method, the instance's `__dict__` is pickled. If there is no `__setstate__()` method, the pickled object must be a dictionary and its items are assigned to the new instance's dictionary. (If a class defines both `__getstate__()` and `__setstate__()`, the state object needn't be a dictionary --- these methods can do what they want.) This protocol is also used by the shallow and deep copying operations defined in the `copy` module.

Note that when class instances are pickled, their class's code and data are not pickled along with them. Only the instance data are pickled. This is done on purpose, so you can fix bugs in a class or add methods and still load objects that were created with an earlier version of the class. If you plan to have long-lived objects that will see many versions of a class, it may be worthwhile to put a version number in the objects so that suitable conversions can be made by the class's `__setstate__()` method.

When a class itself is pickled, only its name is pickled -- the class definition is not pickled, but re-imported by the unpickling process. Therefore, the restriction that the class must be defined at the top level in a module applies to pickled classes as well.

The interface can be summarized as follows.

To pickle an object `x` onto a file `f`, open for writing:

```
p = pickle.Pickler(f)
p.dump(x)
```

A shorthand for this is:

```
pickle.dump(x, f)
```

To unpickle an object `x` from a file `f`, open for reading:

```
u = pickle.Unpickler(f)
x = u.load()
```

A shorthand is:

```
x = pickle.load(f)
```

The `Pickler` class only calls the method `f.write` with a string argument. The `Unpickler` calls the methods `f.read` (with an integer argument) and `f.readline` (without argument), both returning a string. It is explicitly allowed to pass non-file objects here, as long as they have the right methods.

The following types can be pickled:

- `None`
- integers, long integers, floating point numbers
- strings
- tuples, lists and dictionaries containing only picklable objects
- classes that are defined at the top level in a module
- instances of such classes whose `__dict__` or `__setstate__()` is picklable

Attempts to pickle unpicklable objects will raise the `PicklingError` exception; when this happens, an unspecified number of bytes may have been written to the file.

It is possible to make multiple calls to the `dump()` method of the same `Pickler` instance. These must then be matched to the same number of calls to the `load()` instance of the corresponding `Unpickler` instance. If the same object is pickled by multiple `dump()` calls, the `load()` will all yield references to the same object. *Warning*: this is intended for pickling multiple objects without intervening modifications to the objects or their parts. If you modify an object and then pickle it again using the same `Pickler` instance, the object is not pickled again -- a reference to it is pickled and the `Unpickler` will return the old value, not the modified one. (There are two problems here: (a) detecting changes, and (b) marshalling a minimal set of changes. I have no answers. Garbage Collection may also become a problem here.)

Apart from the `Pickler` and `Unpickler` classes, the module defines the following functions, and an exception:

function of module pickle: **dump** *(object, file)*

Write a pickled representation of obect to the open file object file. This is equivalent to `Pickler(file).dump(object)`.

function of module pickle: **load** *(file)*

Read a pickled object from the open file object file. This is equivalent to `Unpickler(file).load()`.

function of module pickle: **dumps** *(object)*

Return the pickled representation of the object as a string, instead of writing it to a file.

function of module pickle: **loads** *(string)*

Read a pickled object from a string instead of a file. Characters in the string past the pickled object's representation are ignored.

exception: module pickle **PicklingError**

This exception is raised when an unpicklable object is passed to `Pickler.dump()`.

Standard Module shelve

A "shelf" is a persistent, dictionary-like object. The difference with "dbm" databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects -- anything that the `pickle` module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys are ordinary strings.

To summarize the interface (`key` is a string, `data` is an arbitrary object):

```
import shelve

d = shelve.open(filename) # open, with (g)dbm filename -- no suffix

d[key] = data   # store data at key (overwrites old data if
                # using an existing key)
data = d[key]   # retrieve data at key (raise KeyError if no
                # such key)
del d[key]      # delete data stored at key (raises KeyError
                # if no such key)
flag = d.has_key(key)   # true if the key exists
list = d.keys() # a list of all existing keys (slow!)

d.close()       # close it
```

Restrictions:

- The choice of which database package will be used (e.g. dbm or gdbm) depends on which interface is available. Therefore it isn't safe to open the database directly using dbm. The database is also (unfortunately) subject to the limitations of dbm, if it is used --- this means that (the pickled representation of) the objects stored in the database should be fairly small, and in rare cases key collisions may cause the database to refuse updates.
- Dependent on the implementation, closing a persistent dictionary may or may not be necessary to flush changes to disk.
- The `shelve` module does not support *concurrent* read/write access to shelved objects. (Multiple simultaneous read accesses are safe.) When a program has a shelf open for writing, no other program should have it open for reading or writing. UNIX file locking can be used to solve this, but this differs across UNIX versions and requires knowledge about the database implementation used.

Standard Module copy

This module provides generic (shallow and deep) copying operations.

Interface summary:


```
import copy

x = copy.copy(y)        # make a shallow copy of y
x = copy.deepcopy(y)    # make a deep copy of y
```

For module specific errors, `copy.error` is raised.

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A *shallow copy* constructs a new compound object and then (to the extent possible) inserts *references* into it to the objects found in the original.
- A *deep copy* constructs a new compound object and then, recursively, inserts *copies* into it of the objects found in the original.

Two problems often exist with deep copy operations that don't exist with shallow copy operations:

- Recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop.
- Because deep copy copies *everything* it may copy too much, e.g. administrative data structures that should be shared even between copies.

Python's `deepcopy()` operation avoids these problems by:

- keeping a table of objects already copied during the current copying pass; and
- letting user-defined classes override the copying operation or the set of components copied.

This version does not copy types like module, class, function, method, nor stack trace, stack frame, nor file, socket, window, nor array, nor any similar types.

Classes can use the same interfaces to control copying that they use to control pickling: they can define methods called `__getinitargs__()`, `__getstate__()` and `__setstate__()`. See the description of module `pickle` for information on these methods.

Built-in Module `marshal`

This module contains functions that can read and write Python values in a binary format. The format is specific to Python, but independent of machine architecture issues (e.g., you can write a Python value to a file on a PC, transport the file to a Sun, and read it back there). Details of the format are undocumented on purpose; it may change between Python versions (although it rarely does).(11)

This is not a general "persistency" module. For general persistency and transfer of Python objects through RPC calls, see the modules `pickle` and `shelve`. The `marshal` module exists mainly to support reading and writing the "pseudo-compiled" code for Python modules of `.pyc' files.

Not all Python object types are supported; in general, only objects whose value is independent from a particular invocation of Python can be written and read by this module. The following types are supported: `None`, integers, long integers, floating point numbers, strings, tuples, lists, dictionaries, and code objects, where it should be understood that tuples, lists and dictionaries are only supported as long

as the values contained therein are themselves supported; and recursive lists and dictionaries should not be written (they will cause infinite loops).

Caveat: On machines where C's `long int` type has more than 32 bits (such as the DEC Alpha or the HP Precision Architecture), it is possible to create plain Python integers that are longer than 32 bits. Since the current `marshal` module uses 32 bits to transfer plain Python integers, such values are silently truncated. This particularly affects the use of very long integer literals in Python modules -- these will be accepted by the parser on such machines, but will be silently be truncated when the module is read from the `.pyc` instead.(12)

There are functions that read/write files as well as functions operating on strings.

The module defines these functions:

function of module marshal: **dump** *(value, file)*

Write the value on the open file. The value must be a supported type. The file must be an open file object such as `sys.stdout` or returned by `open()` or `posix.popen()`.

If the value has an unsupported type, garbage is written which cannot be read back by `load()`.

function of module marshal: **load** *(file)*

Read one value from the open file and return it. If no valid value is read, raise `EOFError`, `ValueError` or `TypeError`. The file must be an open file object.

function of module marshal: **dumps** *(value)*

Return the string that would be written to a file by `dump(value, file)`. The value must be a supported type.

function of module marshal: **loads** *(string)*

Convert the string to a value. If no valid value is found, raise `EOFError`, `ValueError` or `TypeError`. Extra characters in the string are ignored.

Built-in Module `imp`

This module provides an interface to the mechanisms used to implement the `import` statement. It defines the following constants and functions:

function of module struct: **get_magic** *()*

Return the magic string value used to recognize byte-compiled code files ("`.pyc` files").

function of module struct: **get_suffixes** *()*

Return a list of triples, each describing a particular type of file. Each triple has the form `(suffix, mode, type)`, where suffix is a string to be appended to the module name to form the filename to search for, mode is the mode string to pass to the built-in `open` function to open the file (this can be `'r'` for text files or `'rb'` for binary files), and type is the file type, which has one of the values

`PY_SOURCE`, `PY_COMPILED` or `C_EXTENSION`, defined below. (System-dependent values may be returned.)

function of module struct: **find_module** *(name, [path])*

Try to find the module name on the search path path. The default path is `sys.path`. The return value is a triple `(file, pathname, description)` where file is an open file object positioned at the beginning, pathname is the pathname of the file found, and description is a triple as contained in the list returned by `get_suffixes` describing the kind of file found.

function of module struct: **init_builtin** *(name)*

Initialize the built-in module called name and return its module object. If the module was already initialized, it will be initialized *again*. A few modules cannot be initialized twice -- attempting to initialize these again will raise an `ImportError` exception. If there is no built-in module called name, `None` is returned.

function of module struct: **init_frozen** *(name)*

Initialize the frozen module called name and return its module object. If the module was already initialized, it will be initialized *again*. If there is no frozen module called name, `None` is returned. (Frozen modules are modules written in Python whose compiled byte-code object is incorporated into a custom-built Python interpreter by Python's `freeze` utility. See `Tools/freeze` for now.)

function of module struct: **is_builtin** *(name)*

Return `1` if there is a built-in module called name which can be initialized again. Return `-1` if there is a built-in module called name which cannot be initialized again (see `init_builtin`). Return `0` if there is no built-in module called name.

function of module struct: **is_frozen** *(name)*

Return `1` if there is a frozen module (see `init_frozen`) called name, `0` if there is no such module.

function of module struct: **load_compiled** *(name, pathname, [file])*

Load and initialize a module implemented as a byte-compiled code file and return its module object. If the module was already initialized, it will be initialized *again*. The name argument is used to create or access a module object. The pathname argument points to the byte-compiled code file. The optional file argument is the byte-compiled code file, open for reading in binary mode, from the beginning -- if not given, the function opens pathname. It must currently be a real file object, not a user-defined class emulating a file.

function of module struct: **load_dynamic** *(name, pathname, [file])*

Load and initialize a module implemented as a dynamically loadable shared library and return its module object. If the module was already initialized, it will be initialized *again*. Some modules don't like that and may raise an exception. The pathname argument must point to the shared library. The name argument is used to construct the name of the initialization function: an external C function called `initname()` in the shared library is called. The optional file argment is ignored. (Note: using shared libraries is highly system dependent, and not all systems support it.)

function of module struct: **load_source** *(name, pathname, [file])*

Load and initialize a module implemented as a Python source file and return its module object. If the module was already initialized, it will be initialized *again*. The name argument is used to create or access a module object. The pathname argument points to the source file. The optional file argument is the source file, open for reading as text, from the beginning -- if not given, the function opens pathname. It must currently be a real file object, not a user-defined class emulating a file. Note that if a properly matching byte-compiled file (with suffix `.pyc`) exists, it will be used instead of parsing the given source file.

function of module struct: **new_module** *(name)*

Return a new empty module object called name. This object is *not* inserted in `sys.modules`.

The following constants with integer values, defined in the module, are used to indicate the search result of `imp.find_module`.

data: module struct **SEARCH_ERROR**

The module was not found.

data: module struct **PY_SOURCE**

The module was found as a source file.

data: module struct **PY_COMPILED**

The module was found as a compiled code object file.

data: module struct **C_EXTENSION**

The module was found as dynamically loadable shared library.

Examples

The following function emulates the default import statement:

```
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=None):
    # Fast path: see if the module has already been imported.
    if sys.modules.has_key(name):
        return sys.modules[name]

    # If any of the following calls raises an exception,
    # there's a problem we can't handle -- let the caller handle it.

    # See if it's a built-in module.
```


```
    m = imp.init_builtin(name)
    if m:
        return m

    # See if it's a frozen module.
    m = imp.init_frozen(name)
    if m:
        return m

    # Search the default path (i.e. sys.path).
    fp, pathname, (suffix, mode, type) = imp.find_module(name)

    # See what we got.
    # Note that fp will be closed automatically when we return.
    if type == imp.C_EXTENSION:
        return imp.load_dynamic(name, pathname)
    if type == imp.PY_SOURCE:
        return imp.load_source(name, pathname, fp)
    if type == imp.PY_COMPILED:
        return imp.load_compiled(name, pathname, fp)

    # Shouldn't get here at all.
    raise ImportError, '%s: unknown module type (%d)' % (name, type)
```

Built-in Module `__builtin__`

This module provides direct access to all `built-in' identifiers of Python; e.g. `__builtin__.open` is the full name for the built-in function `open`. See the section on Built-in Functions in the previous chapter.

Built-in Module `__main__`

This module represents the (otherwise anonymous) scope in which the interpreter's main program executes -- commands read either from standard input or from a script file.

Go to the previous, next section.

Go to the previous, next section.

String Services

The modules described in this chapter provide a wide range of string manipulation operations. Here's an overview:

string
--- Common string operations.

regex
--- Regular expression search and match operations.

regsub
--- Substitution and splitting operations that use regular expressions.

struct
--- Interpret strings as packed binary data.

Standard Module `string`

This module defines some constants useful for checking character classes and some useful string functions. See the modules `regex` and `regsub` for string functions based on regular expressions.

The constants defined in this module are are:

data: module string **digits**

The string `'0123456789'`.

data: module string **hexdigits**

The string `'0123456789abcdefABCDEF'`.

data: module string **letters**

The concatenation of the strings `lowercase` and `uppercase` described below.

data: module string **lowercase**

A string containing all the characters that are considered lowercase letters. On most systems this is the string `'abcdefghijklmnopqrstuvwxyz'`. Do not change its definition --- the effect on the routines `upper` and `swapcase` is undefined.

data: module string **octdigits**

The string `'01234567'`.

data: module string **uppercase**

A string containing all the characters that are considered uppercase letters. On most systems this is th string `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. Do not change its definition --- the effect on the routines `lower` and `swapcase` is undefined.

data: module string **whitespace**

A string containing all characters that are considered whitespace. On most systems this includes the characters space, tab, linefeed, return, formfeed, and vertical tab. Do not change its definition --- the effect on the routines `strip` and `split` is undefined.

The functions defined in this module are:

function of module string: **atof** *(s)*

Convert a string to a floating point number. The string must have the standard syntax for a floating point literal in Python, optionally preceded by a sign (`+' or `-').

function of module string: **atoi** *(s[, base])*

Convert string s to an integer in the given base. The string must consist of one or more digits, optionally preceded by a sign (`+' or `-'). The base defaults to 10. If it is 0, a default base is chosen depending on the leading characters of the string (after stripping the sign): `0x' or `0X' means 16, `0' means 8, anything else means 10. If base is 16, a leading `0x' or `0X' is always accepted. (Note: for a more flexible interpretation of numeric literals, use the built-in function `eval()`.)

function of module string: **atol** *(s[, base])*

Convert string s to a long integer in the given base. The string must consist of one or more digits, optionally preceded by a sign (`+' or `-'). The base argument has the same meaning as for `atoi()`. A trailing `l' or `L' is not allowed.

function of module string: **expandtabs** *(s, tabsize)*

Expand tabs in a string, i.e. replace them by one or more spaces, depending on the current column and the given tab size. The column number is reset to zero after each newline occurring in the string. This doesn't understand other non-printing characters or escape sequences.

function of module string: **find** *(s, sub[, start])*

Return the lowest index in s not smaller than start where the substring sub is found. Return `-1` when sub does not occur as a substring of s with index at least start. If start is omitted, it defaults to `0`. If start is negative, `len(s)` is added.

function of module string: **rfind** *(s, sub[, start])*

Like `find` but find the highest index.

function of module string: **index** *(s, sub[, start])*

Like `find` but raise `ValueError` when the substring is not found.

function of module string: **rindex** *(s, sub[, start])*

Like `rfind` but raise `ValueError` when the substring is not found.

function of module string: **count** *(s, sub[, start])*

Return the number of (non-overlapping) occurrences of substring sub in string s with index at least start. If start is omitted, it defaults to `0`. If start is negative, `len(s)` is added.

function of module string: **lower** *(s)*

Convert letters to lower case.

function of module string: **split** *(s)*

Return a list of the whitespace-delimited words of the string s.

function of module string: **splitfields** *(s, sep)*

Return a list containing the fields of the string s, using the string sep as a separator. The list will have one more items than the number of non-overlapping occurrences of the separator in the string. Thus, `string.splitfields(s, ' ')` is not the same as `string.split(s)`, as the latter only returns non-empty words. As a special case, `splitfields(s, '')` returns `[s]`, for any string s. (See also `regsub.split()`.)

function of module string: **join** *(words)*

Concatenate a list or tuple of words with intervening spaces.

function of module string: **joinfields** *(words, sep)*

Concatenate a list or tuple of words with intervening separators. It is always true that `string.joinfields(string.splitfields(t, sep), sep)` equals t.

function of module string: **strip** *(s)*

Remove leading and trailing whitespace from the string s.

function of module string: **swapcase** *(s)*

Convert lower case letters to upper case and vice versa.

function of module string: **upper** *(s)*

Convert letters to upper case.

function of module string: **ljust** *(s, width)*

function of module string: **rjust** *(s, width)*

function of module string: **center** *(s, width)*

These functions respectively left-justify, right-justify and center a string in a field of given width. They return a string that is at least width characters wide, created by padding the string s with spaces until the given width on the right, left or both sides. The string is never truncated.

function of module string: **zfill** *(s, width)*

Pad a numeric string on the left with zero digits until the given width is reached. Strings starting with a sign are handled correctly.

This module is implemented in Python. Much of its functionality has been reimplemented in the built-in module `strop`. However, you should *never* import the latter module directly. When `string` discovers that `strop` exists, it transparently replaces parts of itself with the implementation from `strop`. After initialization, there is *no* overhead in using `string` instead of `strop`.

Built-in Module `regex`

This module provides regular expression matching operations similar to those found in Emacs. It is always available.

By default the patterns are Emacs-style regular expressions; there is a way to change the syntax to match that of several well-known UNIX utilities.

This module is 8-bit clean: both patterns and strings may contain null bytes and characters whose high bit is set.

Please note: There is a little-known fact about Python string literals which means that you don't usually have to worry about doubling backslashes, even though they are used to escape special characters in string literals as well as in regular expressions. This is because Python doesn't remove backslashes from string literals if they are followed by an unrecognized escape character. *However*, if you want to include a literal backslash in a regular expression represented as a string literal, you have to *quadruple* it. E.g. to extract LaTeX `\section{...}' headers from a document, you can use this pattern: `'\\\\section{\(.*\)}'`.

The module defines these functions, and an exception:

function of module regex: **match** *(pattern, string)*

Return how many characters at the beginning of string match the regular expression pattern. Return `-1` if the string does not match the pattern (this is different from a zero-length match!).

function of module regex: **search** *(pattern, string)*

Return the first position in string that matches the regular expression pattern. Return -1 if no position in the string matches the pattern (this is different from a zero-length match anywhere!).

function of module regex: **compile** *(pattern[, translate])*

Compile a regular expression pattern into a regular expression object, which can be used for matching using its `match` and `search` methods, described below. The optional argument translate, if present, must be a 256-character string indicating how characters (both of the pattern and of the strings to be matched) are translated before comparing them; the `i`-th element of the string gives the translation for the character with ASCII code `i`. This can be used to implement case-insensitive matching; see the `casefold` data item below.

The sequence

```
prog = regex.compile(pat)
result = prog.match(str)
```

is equivalent to

```
result = regex.match(pat, str)
```

but the version using `compile()` is more efficient when multiple regular expressions are used concurrently in a single program. (The compiled version of the last pattern passed to `regex.match()` or `regex.search()` is cached, so programs that use only a single regular expression at a time needn't worry about compiling regular expressions.)

function of module regex: **set_syntax** *(flags)*

Set the syntax to be used by future calls to `compile`, `match` and `search`. (Already compiled expression objects are not affected.) The argument is an integer which is the OR of several flag bits. The return value is the previous value of the syntax flags. Names for the flags are defined in the standard module `regex_syntax`; read the file `` `regex_syntax.py' `` for more information.

function of module regex: **symcomp** *(pattern[, translate])*

This is like `compile`, but supports symbolic group names: if a parenthesis-enclosed group begins with a group name in angular brackets, e.g. `'\(<id>[a-z][a-z0-9]*\)'`, the group can be referenced by its name in arguments to the `group` method of the resulting compiled regular expression object, like this: `p.group('id')`. Group names may contain alphanumeric characters and `'_'` only.

exception: module regex **error**

Exception raised when a string passed to one of the functions here is not a valid regular expression (e.g., unmatched parentheses) or when some other error occurs during compilation or matching. (It is never an error if a string contains no match for a pattern.)

data: module regex **casefold**

A string suitable to pass as translate argument to `compile` to map all upper case characters to their lowercase equivalents.

Compiled regular expression objects support these methods:

Method: regex **match** *(string[, pos])*

Return how many characters at the beginning of string match the compiled regular expression. Return `-1` if the string does not match the pattern (this is different from a zero-length match!).

The optional second parameter pos gives an index in the string where the search is to start; it defaults to `0`. This is not completely equivalent to slicing the string; the `'^'` pattern character matches at the real begin of the string and at positions just after a newline, not necessarily at the index where the search is to start.

Method: regex **search** *(string[, pos])*

Return the first position in string that matches the regular expression `pattern`. Return `-1` if no position in the string matches the pattern (this is different from a zero-length match anywhere!).

The optional second parameter has the same meaning as for the `match` method.

Method: regex **group** *(index, index, ...)*

This method is only valid when the last call to the `match` or `search` method found a match. It returns one or more groups of the match. If there is a single index argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. If the index is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1..99], it is the string matching the the corresponding parenthesized group (using the default syntax, groups are parenthesized using
`(` and
`)`). If no such group exists, the corresponding result is `None`.

If the regular expression was compiled by `symcomp` instead of `compile`, the index arguments may also be strings identifying groups by their group name.

Compiled regular expressions support these data attributes:

attribute: regex **regs**

When the last call to the `match` or `search` method found a match, this is a tuple of pairs of indices corresponding to the beginning and end of all parenthesized groups in the pattern. Indices are relative to the string argument passed to `match` or `search`. The 0-th tuple gives the beginning and end or the whole pattern. When the last match or search failed, this is `None`.

attribute: regex **last**

When the last call to the `match` or `search` method found a match, this is the string argument passed to that method. When the last match or search failed, this is `None`.

attribute: regex **translate**

This is the value of the translate argument to `regex.compile` that created this regular expression object. If the translate argument was omitted in the `regex.compile` call, this is `None`.

attribute: regex **givenpat**

The regular expression pattern as passed to `compile` or `symcomp`.

attribute: regex **realpat**

The regular expression after stripping the group names for regular expressions compiled with `symcomp`. Same as `givenpat` otherwise.

attribute: regex **groupindex**

A dictionary giving the mapping from symbolic group names to numerical group indices for regular

expressions compiled with `symcomp`. `None` otherwise.

Standard Module `regsub`

This module defines a number of functions useful for working with regular expressions (see built-in module `regex`).

function of module regsub: **sub** *(pat, repl, str)*

Replace the first occurrence of pattern pat in string str by replacement repl. If the pattern isn't found, the string is returned unchanged. The pattern may be a string or an already compiled pattern. The replacement may contain references `\digit' to subpatterns and escaped backslashes.

function of module regsub: **gsub** *(pat, repl, str)*

Replace all (non-overlapping) occurrences of pattern pat in string str by replacement repl. The same rules as for `sub()` apply. Empty matches for the pattern are replaced only when not adjacent to a previous match, so e.g. `gsub('', '-', 'abc')` returns `'-a-b-c-'`.

function of module regsub: **split** *(str, pat)*

Split the string str in fields separated by delimiters matching the pattern pat, and return a list containing the fields. Only non-empty matches for the pattern are considered, so e.g. `split('a:b', ':*')` returns `['a', 'b']` and `split('abc', '')` returns `['abc']`.

Built-in Module `struct`

This module performs conversions between Python values and C structs represented as Python strings. It uses format strings (explained below) as compact descriptions of the lay-out of the C structs and the intended conversion to/from Python values.

See also built-in module `array`.

The module defines the following exception and functions:

exception: module struct **error**

Exception raised on various occasions; argument is a string describing what is wrong.

function of module struct: **pack** *(fmt, v1, v2, ...)*

Return a string containing the values `v1, v2, ...` packed according to the given format. The arguments must match the values required by the format exactly.

function of module struct: **unpack** *(fmt, string)*

Unpack the string (presumably packed by `pack(fmt, ...)`) according to the given format. The result is a tuple even if it contains exactly one item. The string must contain exactly the amount of data required by the format (i.e. `len(string)` must equal `calcsize(fmt)`).

function of module struct: **calcsize** *(fmt)*

Return the size of the struct (and hence of the string) corresponding to the given format.

Format characters have the following meaning; the conversion between C and Python values should be obvious given their types:

Format

C -- Python

`x'

pad byte -- no value

`c'

char -- string of length 1

`b'

signed char -- integer

`h'

short -- integer

`i'

int -- integer

`l'

long -- integer

`f'

float -- float

`d'

double -- float

A format character may be preceded by an integral repeat count; e.g. the format string `'4h'` means exactly the same as `'hhhh'`.

C numbers are represented in the machine's native format and byte order, and properly aligned by skipping pad bytes if necessary (according to the rules used by the C compiler).

Examples (all on a big-endian machine):

```
pack('hhl', 1, 2, 3) == '\000\001\000\002\000\000\000\003'
unpack('hhl', '\000\001\000\002\000\000\000\003') == (1, 2, 3)
calcsize('hhl') == 8
```

Hint: to align the end of a structure to the alignment requirement of a particular type, end the format with the code for that type with a repeat count of zero, e.g. the format `'llh0l'` specifies two pad bytes at the end, assuming longs are aligned on 4-byte boundaries.

(More format characters are planned, e.g. `'s'` for character arrays, upper case for unsigned variants, and a way to specify the byte order, which is useful for [de]constructing network packets and reading/writing portable binary file formats like TIFF and AIFF.)

Go to the previous, next section.

Go to the previous, next section.

Miscellaneous Services

The modules described in this chapter provide miscellaneous services that are available in all Python versions. Here's an overview:

math
--- Mathematical functions (`sin()` etc.).

rand
--- Integer random number generator.

whrandom
--- Floating point random number generator.

array
--- Efficient arrays of uniformly typed numeric values.

Built-in Module `math`

This module is always available. It provides access to the mathematical functions defined by the C standard. They are: function of module math: **acos** *(x)*

function of module math: **asin** *(x)*

function of module math: **atan** *(x)*

function of module math: **atan2** *(x, y)*

function of module math: **ceil** *(x)*

function of module math: **cos** *(x)*

function of module math: **cosh** *(x)*

function of module math: **exp** *(x)*

function of module math: **fabs** *(x)*

function of module math: **floor** *(x)*

function of module math: **fmod** *(x, y)*

function of module math: **frexp** *(x)*

function of module math: **hypot** *(x, y)*

function of module math: **ldexp** *(x, y)*

function of module math: **log** *(x)*

function of module math: **log10** *(x)*

function of module math: **modf** *(x)*

function of module math: **pow** *(x, y)*

function of module math: **sin** *(x)*

function of module math: **sinh** *(x)*

function of module math: **sqrt** *(x)*

function of module math: **tan** *(x)*

function of module math: **tanh** *(x)*

Note that `frexp` and `modf` have a different call/return pattern than their C equivalents: they take a single argument and return a pair of values, rather than returning their second return value through an `output parameter' (there is no such thing in Python).

The `hypot` function, which is not standard C, is not available on all platforms.

The module also defines two mathematical constants: data: module math **pi**

data: module math **e**

Standard Module `rand`

This module implements a pseudo-random number generator with an interface similar to `rand()` in C. the following functions:

function of module rand: **rand** *()*

Returns an integer random number in the range [0 ... 32768).

function of module rand: **choice** *(s)*

Returns a random element from the sequence (string, tuple or list) s.

function of module rand: **srand** *(seed)*

Initializes the random number generator with the given integral seed. When the module is first imported, the random number is initialized with the current time.

Standard Module `whrandom`

This module implements a Wichmann-Hill pseudo-random number generator. It defines the following functions:

function of module whrandom: **random** *()*

Returns the next random floating point number in the range [0.0 ... 1.0).

function of module whrandom: **seed** *(x, y, z)*

Initializes the random number generator from the integers x, y and z. When the module is first imported, the random number is initialized using values derived from the current time.

Built-in Module `array`

This module defines a new object type which can efficiently represent an array of basic values: characters, integers, floating point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a type code, which is a single character. The following type codes are defined:

Typecode

Type -- Minimal size in bytes

`'c'`

character -- 1

`'b'`

signed integer -- 1

`'h'`

signed integer -- 2

`'i'`

signed integer -- 2

`'l'`

signed integer -- 4

`'f'`

floating point -- 4

`'d'`

floating point -- 8

The actual representation of values is determined by the machine architecture (strictly speaking, by the C implementation). The actual size can be accessed through the itemsize attribute.

See also built-in module `struct`.

The module defines the following function:

function of module array: **array** *(typecode[, initializer])*

Return a new array whose items are restricted by typecode, and initialized from the optional initializer value, which must be a list or a string. The list or string is passed to the new array's `fromlist()` or `fromstring()` method (see below) to add initial items to the array.

Array objects support the following data items and methods:

data: module array **typecode**

The typecode character used to create the array.

data: module array **itemsize**

The length in bytes of one array item in the internal representation.

function of module array: **append** *(x)*

Append a new item with value x to the end of the array.

function of module array: **byteswap** *(x)*

"Byteswap" all items of the array. This is only supported for integer values. It is useful when reading data from a file written on a machine with a different byte order.

function of module array: **fromfile** *(f, n)*

Read n items (as machine values) from the file object f and append them to the end of the array. If less than n items are available, `EOFError` is raised, but the items that were available are still inserted into the array. f must be a real built-in file object; something else with a `read()` method won't do.

function of module array: **fromlist** *(list)*

Append items from the list. This is equivalent to `for x in list: a.append(x)` except that if there is a type error, the array is unchanged.

function of module array: **fromstring** *(s)*

Appends items from the string, interpreting the string as an array of machine values (i.e. as if it had been read from a file using the `fromfile()` method).

function of module array: **insert** *(i, x)*

Insert a new item with value x in the array before position i.

function of module array: **tofile** *(f)*

Write all items (as machine values) to the file object f.

function of module array: **tolist** *()*

Convert the array to an ordinary list with the same items.

function of module array: **tostring** *()*

Convert the array to an array of machine values and return the string representation (the same sequence of bytes that would be written to a file by the `tofile()` method.)

When an array object is printed or converted to a string, it is represented as `array(typecode, initializer)`. The initializer is omitted if the array is empty, otherwise it is a string if the typecode

is `'c'`, otherwise it is a list of numbers. The string is guaranteed to be able to be converted back to a array with the same type and value using reverse quotes (`). Examples:

```
array('l')
array('c', 'hello world')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

Go to the previous, next section.

Go to the previous, next section.

Generic Operating System Services

The modules described in this chapter provide interfaces to operating system features that are available on (almost) all operating systems, such as files and a clock. The interfaces are generally modelled after the UNIX or C interfaces but they are available on most other systems as well. Here's an overview:

os

--- Miscellaneous OS interfaces.

time

--- Time access and conversions.

getopt

--- Parser for command line options.

tempfile

--- Generate temporary file names.

Standard Module os

This module provides a more portable way of using operating system (OS) dependent functionality than importing an OS dependent built-in module like `posix`.

When the optional built-in module `posix` is available, this module exports the same functions and data as `posix`; otherwise, it searches for an OS dependent built-in module like `mac` and exports the same functions and data as found there. The design of all Python's built-in OS dependent modules is such that as long as the same functionality is available, it uses the same interface; e.g., the function `os.stat(file)` returns stat info about a file in a format compatible with the POSIX interface.

Extensions peculiar to a particular OS are also available through the `os` module, but using them is of course a threat to portability!

Note that after the first time `os` is imported, there is *no* performance penalty in using functions from `os` instead of directly from the OS dependent built-in module, so there should be *no* reason not to use `os`!

In addition to whatever the correct OS dependent module exports, the following variables and functions are always exported by `os`:

data: module os **name**

The name of the OS dependent module imported. The following names have currently been registered: `'posix'`, `'nt'`, `'dos'`, `'mac'`.

data: module os **path**

The corresponding OS dependent standard module for pathname operations, e.g., `posixpath` or

`macpath`. Thus, (given the proper imports), `os.path.split(file)` is equivalent to but more portable than `posixpath.split(file)`.

data: module os **curdir**

The constant string used by the OS to refer to the current directory, e.g. `'.'` for POSIX or `':'` for the Mac.

data: module os **pardir**

The constant string used by the OS to refer to the parent directory, e.g. `'..'` for POSIX or `'::'` for the Mac.

data: module os **sep**

The character used by the OS to separate pathname components, e.g. `'/'` for POSIX or `':'` for the Mac. Note that knowing this is not sufficient to be able to parse or concatenate pathnames--better use `os.path.split()` and `os.path.join()`---but it is occasionally useful.

data: module os **pathsep**

The character conventionally used by the OS to separate search patch components (as in `$PATH`), e.g. `':'` for POSIX or `';'` for MS-DOS.

data: module os **defpath**

The default search path used by `os.exec*p*()` if the environment doesn't have a `'PATH'` key.

function of module os: **execl** *(path, arg0, arg1, ...)*

This is equivalent to `os.execv(path, (arg0, arg1, ...))`.

function of module os: **execle** *(path, arg0, arg1, ..., env)*

This is equivalent to `os.execve(path, (arg0, arg1, ...), env)`.

function of module os: **execlp** *(path, arg0, arg1, ...)*

This is equivalent to `os.execvp(path, (arg0, arg1, ...))`.

function of module os: **execvp** *(path, args)*

This is like `os.execv(path, args)` but duplicates the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from `os.environ['PATH']`.

function of module os: **execvpe** *(path, args, env)*

This is a cross between `os.execve()` and `os.execvp()`. The directory list is obtained from `env['PATH']`.

(The functions `os.execv()` and `execve()` are not documented here, since they are implemented by the OS dependent module. If the OS dependent module doesn't define either of these, the functions that rely on it will raise an exception. They are documented in the section on module `posix`, together with

all other functions that `os` imports from the OS dependent module.)

Built-in Module `time`

This module provides various time-related functions. It is always available.

An explanation of some terminology and conventions is in order.

- The "epoch" is the point where the time starts. On January 1st of that year, at 0 hours, the "time since the epoch" is zero. For UNIX, the epoch is 1970. To find out what the epoch is, look at `gmtime(0)`.
- UTC is Coordinated Universal Time (formerly known as Greenwich Mean Time). The acronym UTC is not a mistake but a compromise between English and French.
- DST is Daylight Saving Time, an adjustment of the timezone by (usually) one hour during part of the year. DST rules are magic (determined by local law) and can change from year to year. The C library has a table containing the local rules (often it is read from a system file for flexibility) and is the only source of True Wisdom in this respect.
- The precision of the various real-time functions may be less than suggested by the units in which their value or argument is expressed. E.g. on most UNIX systems, the clock "ticks" only 50 or 100 times a second, and on the Mac, times are only accurate to whole seconds.

The module defines the following functions and data items:

data: module time **altzone**

The offset of the local DST timezone, in seconds west of the 0th meridian, if one is defined. Negative if the local DST timezone is east of the 0th meridian (as in Western Europe, including the UK). Only use this if `daylight` is nonzero.

function of module time: **asctime** *(tuple)*

Convert a tuple representing a time as returned by `gmtime()` or `localtime()` to a 24-character string of the following form: `'Sun Jun 20 23:21:05 1993'`. Note: unlike the C function of the same name, there is no trailing newline.

function of module time: **clock** *()*

Return the current CPU time as a floating point number expressed in seconds. The precision, and in fact the very definiton of the meaning of "CPU time", depends on that of the C function of the same name.

function of module time: **ctime** *(secs)*

Convert a time expressed in seconds since the epoch to a string representing local time. `ctime(t)` is equivalent to `asctime(localtime(t))`.

data: module time **daylight**

Nonzero if a DST timezone is defined.

function of module time: **gmtime** *(secs)*

Convert a time expressed in seconds since the epoch to a tuple of 9 integers, in UTC: year (e.g. 1993), month (1--12), day (1--31), hour (0--23), minute (0--59), second (0--59), weekday (0--6, monday is 0), Julian day (1--366), dst flag (always zero). Fractions of a second are ignored. Note subtle differences with the C function of this name.

function of module time: **localtime** *(secs)*

Like `gmtime` but converts to local time. The dst flag is set to 1 when DST applies to the given time.

function of module time: **mktime** *(tuple)*

This is the inverse function of `localtime`. Its argument is the full 9-tuple (since the dst flag is needed). It returns an integer.

function of module time: **sleep** *(secs)*

Suspend execution for the given number of seconds. The argument may be a floating point number to indicate a more precise sleep time.

function of module time: **time** *()*

Return the time as a floating point number expressed in seconds since the epoch, in UTC. Note that even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second.

data: module time **timezone**

The offset of the local (non-DST) timezone, in seconds west of the 0th meridian (i.e. negative in most of Western Europe, positive in the US, zero in the UK).

data: module time **tzname**

A tuple of two strings: the first is the name of the local non-DST timezone, the second is the name of the local DST timezone. If no DST timezone is defined, the second string should not be used.

Standard Module `getopt`

This module helps scripts to parse the command line arguments in `sys.argv`. It uses the same conventions as the UNIX `getopt()` function (including the special meanings of arguments of the form `-' and `--'). It defines the function `getopt.getopt(args, options)` and the exception `getopt.error`.

The first argument to `getopt()` is the argument list passed to the script with its first element chopped off (i.e., `sys.argv[1:]`). The second argument is the string of option letters that the script wants to recognize, with options that require an argument followed by a colon (i.e., the same format that UNIX `getopt()` uses). The return value consists of two elements: the first is a list of option-and-value pairs; the second is the list of program arguments left after the option list was stripped (this is a trailing slice of the first argument). Each option-and-value pair returned has the option as its first element, prefixed with a hyphen (e.g., `'-x'`), and the option argument as its second element, or an empty string if the option

has no argument. The options occur in the list in the same order in which they were found, thus allowing multiple occurrences. Example:

```
>>> import getopt, string
>>> args = string.split('-a -b -cfoo -d bar a1 a2')
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
>>>
```

The exception `getopt.error = 'getopt error'` is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none. The argument to the exception is a string indicating the cause of the error.

Standard Module `tempfile`

This module generates temporary file names. It is not UNIX specific, but it may require some help on non-UNIX systems.

Note: the modules does not create temporary files, nor does it automatically remove them when the current process exits or dies.

The module defines a single user-callable function:

function of module tempfile: **mktemp** *()*

Return a unique temporary filename. This is an absolute pathname of a file that does not exist at the time the call is made. No two calls will return the same filename.

The module uses two global variables that tell it how to construct a temporary name. The caller may assign values to them; by default they are initialized at the first call to `mktemp()`.

data: module tempfile **tempdir**

When set to a value other than `None`, this variable defines the directory in which filenames returned by `mktemp()` reside. The default is taken from the environment variable `TMPDIR`; if this is not set, either `/usr/tmp` is used (on UNIX), or the current working directory (all other systems). No check is made to see whether its value is valid.

data: module tempfile **template**

When set to a value other than `None`, this variable defines the prefix of the final component of the filenames returned by `mktemp()`. A string of decimal digits is added to generate unique filenames. The default is either "`@pid.`" where pid is the current process ID (on UNIX), or "`tmp`" (all other systems).

Warning: if a UNIX process uses `mktemp()`, then calls `fork()` and both parent and child continue to use `mktemp()`, the processes will generate conflicting temporary names. To resolve this, the child process should assign `None` to `template`, to force recomputing the default on the next call to `mktemp()`.

Go to the previous, next section.

Go to the previous, next section.

Optional Operating System Services

The modules described in this chapter provide interfaces to operating system features that are available on selected operating systems only. The interfaces are generally modelled after the UNIX or C interfaces but they are available on some other systems as well (e.g. Windows or NT). Here's an overview:

signal
: --- Set handlers for asynchronous events.

socket
: --- Low-level networking interface.

select
: --- Wait for I/O completion on multiple streams.

thread
: --- Create multiple threads of control within one namespace.

Built-in Module `signal`

This module provides mechanisms to use signal handlers in Python. Some general rules for working with signals handlers:

- A handler for a particular signal, once set, remains installed until it is explicitly reset (i.e. Python uses the BSD style interface).
- There is no way to "block" signals temporarily from critical sections (since this is not supported by all UNIX flavors).
- Although Python signal handlers are called asynchronously as far as the Python user is concerned, they can only occur between the "atomic" instructions of the Python interpreter. This means that signals arriving during long calculations implemented purely in C (e.g. regular expression matches on large bodies of text) may be delayed for an arbitrary amount of time.
- When a signal arrives during an I/O operation, it is possible that the I/O operation raises an exception after the signal handler returns. This is dependent on the underlying UNIX system's semantics regarding interrupted system calls.
- Because the C signal handler always returns, it makes little sense to catch synchronous errors like `SIGFPE` or `SIGSEGV`.
- Python installs a small number of signal handlers by default: `SIGPIPE` is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions), `SIGINT` is translated into a `KeyboardInterrupt` exception, and `SIGTERM` is caught so that necessary cleanup (especially `sys.exitfunc`) can be performed before actually terminating. All of these can be overridden.
- Some care must be taken if both signals and threads are used in the same program. The fundamental thing to remember in using signals and threads simultaneously is: always perform

`signal()` operations in the main thread of execution. Any thread can perform an `alarm()`, `getsignal()`, or `pause()`; only the main thread can set a new signal handler, and the main thread will be the only one to receive signals (this is enforced by the Python signal module, even if the underlying thread implementation supports sending signals to individual threads). This means that signals can't be used as a means of interthread communication. Use locks instead.

The variables defined in the signal module are:

data: module signal **SIG_DFL**

This is one of two standard signal handling options; it will simply perform the default function for the signal. For example, on most systems the default action for SIGQUIT is to dump core and exit, while the default action for SIGCLD is to simply ignore it.

data: module signal **SIG_IGN**

This is another standard signal handler, which will simply ignore the given signal.

data: module signal **SIG***

All the signal numbers are defined symbolically. For example, the hangup signal is defined as `signal.SIGHUP`; the variable names are identical to the names used in C programs, as found in `` `signal.h' ``. The UNIX man page for `` `signal' `` lists the existing signals (on some systems this is `` `signal(2)' ``, on others the list is in `` `signal(7)' ``). Note that not all systems define the same set of signal names; only those names defined by the system are defined by this module.

data: module signal **NSIG**

One more than the number of the highest signal number.

The signal module defines the following functions:

function of module signal: **alarm** *(time)*

If time is non-zero, this function requests that a `SIGALRM` signal be sent to the process in time seconds. Any previously scheduled alarm is canceled (i.e. only one alarm can be scheduled at any time). The returned value is then the number of seconds before any previously set alarm was to have been delivered. If time is zero, no alarm id scheduled, and any scheduled alarm is canceled. The return value is the number of seconds remaining before a previously scheduled alarm. If the return value is zero, no alarm is currently scheduled. (See the UNIX man page `alarm(2)`.)

function of module signal: **getsignal** *(signalnum)*

Return the current signal handler for the signal signalnum. The returned value may be a callable Python object, or one of the special values `signal.SIG_IGN`, `signal.SIG_DFL` or `None`. Here, `signal.SIG_IGN` means that the signal was previously ignored, `signal.SIG_DFL` means that the default way of handling the signal was previously in use, and `None` means that the previous signal handler was not installed from Python.

function of module signal: **pause** *()*

Cause the process to sleep until a signal is received; the appropriate handler will then be called. Retu nothing. (See the UNIX man page `signal(2)`.)

function of module signal: **signal** *(signalnum, handler)*

Set the handler for signal signalnum to the function handler. handler can be any callable Python object, or one of the special values `signal.SIG_IGN` or `signal.SIG_DFL`. The previous signal handler will be returned (see the description of `getsignal()` above). (See the UNIX man page `signal(2)`.)

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a `ValueError` exception to be raised.

The handler is called with two arguments: the signal number and the current stack frame (`None` or a frame object; see the reference manual for a description of frame objects).

Built-in Module socket

This module provides access to the BSD *socket* interface. It is available on UNIX systems that support this interface.

For an introduction to socket programming (in C), see the following papers: *An Introductory 4.3BSD Interprocess Communication Tutorial*, by Stuart Sechrest and *An Advanced 4.3BSD Interprocess Communication Tutorial*, by Samuel J. Leffler et al, both in the UNIX Programmer's Manual, Supplementary Documents 1 (sections PS1:7 and PS1:8). The UNIX manual pages for the various socket-related system calls are also a valuable source of information on the details of socket semantics.

The Python interface is a straightforward transliteration of the UNIX system call and library interface for sockets to Python's object-oriented style: the `socket()` function returns a socket object whose methods implement the various socket system calls. Parameter types are somewhat higer-level than in the C interface: as with `read()` and `write()` operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

Socket addresses are represented as a single string for the `AF_UNIX` address family and as a pair `(host, port)` for the `AF_INET` address family, where host is a string representing either a hostname in Internet domain notation like `'daring.cwi.nl'` or an IP address like `'100.50.200.5'`, and port is an integral port number. Other address families are currently not supported. The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised; errors related to socket or address semantics raise the error `socket.error`.

Non-blocking mode is supported through the `setblocking()` method.

The module `socket` exports the following constants and functions:

exception: module socket **error**

This exception is raised for socket- or address-related errors. The accompanying value is either a string

telling what went wrong or a pair `(errno, string)` representing an error returned by a system c similar to the value accompanying `posix.error`.

data: module socket **AF_UNIX**

data: module socket **AF_INET**

These constants represent the address (and protocol) families, used for the first argument to `socket()`. If the `AF_UNIX` constant is not defined then this protocol is unsupported.

data: module socket **SOCK_STREAM**

data: module socket **SOCK_DGRAM**

data: module socket **SOCK_RAW**

data: module socket **SOCK_RDM**

data: module socket **SOCK_SEQPACKET**

These constants represent the socket types, used for the second argument to `socket()`. (Only `SOCK_STREAM` and `SOCK_DGRAM` appear to be generally useful.)

data: module socket **SO_***

data: module socket **SOMAXCONN**

data: module socket **MSG_***

data: module socket **SOL_***

data: module socket **IPPROTO_***

data: module socket **IPPORT_***

data: module socket **INADDR_***

data: module socket **IP_***

Many constants of these forms, documented in the UNIX documentation on sockets and/or the IP protocol, are also defined in the socket module. They are generally used in arguments to the `setsockopt` and `getsockopt` methods of socket objects. In most cases, only those symbols that are defined in the UNIX header files are defined; for a few symbols, default values are provided.

function of module socket: **gethostbyname** *(hostname)*

Translate a host name to IP address format. The IP address is returned as a string, e.g., `'100.50.200.5'`. If the host name is an IP address itself it is returned unchanged.

function of module socket: **gethostname** *()*

Return a string containing the hostname of the machine where the Python interpreter is currently executing. If you want to know the current machine's IP address, use `socket.gethostbyname(socket.gethostname())`.

function of module socket: **gethostbyaddr** *(ip_address)*

Return a triple `(hostname, aliaslist, ipaddrlist)` where `hostname` is the primary host name responding to the given ip_address, `aliaslist` is a (possibly empty) list of alternative host names for the same address, and `ipaddrlist` is a list of IP addresses for the same interface on the same host (most likely containing only a single address).

function of module socket: **getservbyname** *(servicename, protocolname)*

Translate an Internet service name and protocol name to a port number for that service. The protocol name should be `'tcp'` or `'udp'`.

function of module socket: **socket** *(family, type[, proto])*

Create a new socket using the given address family, socket type and protocol number. The address family should be `AF_INET` or `AF_UNIX`. The socket type should be `SOCK_STREAM`, `SOCK_DGRAM` or perhaps one of the other `SOCK_' constants. The protocol number is usually zero and may be omitted in that case.

function of module socket: **fromfd** *(fd, family, type[, proto])*

Build a socket object from an existing file descriptor (an integer as returned by a file object's `fileno` method). Address family, socket type and protocol number are as for the `socket` function above. The file descriptor should refer to a socket, but this is not checked -- subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (e.g. a server started by the UNIX inet daemon).

Socket Objects

Socket objects have the following methods. Except for `makefile()` these correspond to UNIX system calls applicable to sockets.

Method: socket **accept** *()*

Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair `(conn, address)` where conn is a *new* socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection.

Method: socket **bind** *(address)*

Bind the socket to address. The socket must not already be bound. (The format of address depends on the address family -- see above.)

Method: socket **close** *()*

Close the socket. All future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

Method: socket **connect** *(address)*

Connect to a remote socket at address. (The format of address depends on the address family -- see above.)

Method: socket **fileno** *()*

Return the socket's file descriptor (a small integer). This is useful with `select`.

Method: socket **getpeername** *()*

Return the remote address to which the socket is connected. This is useful to find out the port number of a remote IP socket, for instance. (The format of the address returned depends on the address family --- see above.) On some systems this function is not supported.

Method: socket **getsockname** *()*

Return the socket's own address. This is useful to find out the port number of an IP socket, for instance. (The format of the address returned depends on the address family --- see above.)

Method: socket **getsockopt** *(level, optname[, buflen])*

Return the value of the given socket option (see the UNIX man page *getsockopt*(2)). The needed symbolic constants (`SO_*` etc.) are defined in this module. If buflen is absent, an integer option is assumed and its integer value is returned by the function. If buflen is present, it specifies the maximum length of the buffer used to receive the option in, and this buffer is returned as a string. It is up to the caller to decode the contents of the buffer (see the optional built-in module `struct` for a way to decode C structures encoded as strings).

Method: socket **listen** *(backlog)*

Listen for connections made to the socket. The backlog argument specifies the maximum number of queued connections and should be at least 1; the maximum value is system-dependent (usually 5).

Method: socket **makefile** *([mode[, bufsize]])*

Return a file object associated with the socket. (File objects were described earlier under Built-in Types.) The file object references a `dup()`ped version of the socket file descriptor, so the file object and socket object may be closed or garbage-collected independently. The optional mode and bufsize arguments are interpreted the same way as by the built-in `open()` function.

Method: socket **recv** *(bufsize[, flags])*

Receive data from the socket. The return value is a string representing the data received. The maximum amount of data to be received at once is specified by bufsize. See the UNIX manual page for the meaning of the optional argument flags; it defaults to zero.

Method: socket **recvfrom** *(bufsize[, flags])*

Receive data from the socket. The return value is a pair `(string, address)` where string is a string representing the data received and address is the address of the socket sending the data. The optional flags argument has the same meaning as for `recv()` above. (The format of address depends on the address family -- see above.)

Method: socket **send** *(string[, flags])*

Send data to the socket. The socket must be connected to a remote socket. The optional flags argument has the same meaning as for `recv()` above. Return the number of bytes sent.

Method: socket **sendto** *(string[, flags], address)*

Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by `address`. The optional flags argument has the same meaning as for `recv()` above. Return the number of bytes sent. (The format of address depends on the address family -- see above.)

Method: socket **setblocking** *(flag)*

Set blocking or non-blocking mode of the socket: if flag is 0, the socket is set to non-blocking, else to blocking mode. Initially all sockets are in blocking mode. In non-blocking mode, if a `recv` call doesn't find any data, or if a `send` call can't immediately dispose of the data, a `socket.error` exception is raised; in blocking mode, the calls block until they can proceed.

Method: socket **setsockopt** *(level, optname, value)*

Set the value of the given socket option (see the UNIX man page *setsockopt*(2)). The needed symbolic constants are defined in the `socket` module (`SO_*` etc.). The value can be an integer or a string representing a buffer. In the latter case it is up to the caller to ensure that the string contains the proper bits (see the optional built-in module `struct` for a way to encode C structures as strings).

Method: socket **shutdown** *(how)*

Shut down one or both halves of the connection. If how is `0`, further receives are disallowed. If how is `1`, further sends are disallowed. If how is `2`, further sends and receives are disallowed.

Note that there are no methods `read()` or `write()`; use `recv()` and `send()` without flags argument instead.

Example

Here are two minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence `socket`, `bind`, `listen`, `accept` (possibly repeating the `accept` to service more than one client), while a client only needs the sequence `socket`, `connect`. Also note that the server does not `send`/`receive` on the socket it is listening on but on the new socket returned by `accept`.

```
# Echo server program
from socket import *
HOST = ''                 # Symbolic name meaning the local host
PORT = 50007              # Arbitrary non-privileged server
s = socket(AF_INET, SOCK_STREAM)
s.bind(HOST, PORT)
s.listen(1)
```


```
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()

# Echo client program
from socket import *
HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = socket(AF_INET, SOCK_STREAM)
s.connect(HOST, PORT)
s.send('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', `data`
```

Built-in Module `select`

This module provides access to the function `select` available in most UNIX versions. It defines the following:

exception: module select **error**

The exception raised when an error occurs. The accompanying value is a pair containing the numeric error code from `errno` and the corresponding string, as would be printed by the C function `perror()`.

function of module select: **select** *(iwtd, owtd, ewtd[, timeout])*

This is a straightforward interface to the UNIX `select()` system call. The first three arguments are lists of `waitable objects': either integers representing UNIX file descriptors or objects with a parameterless method named `fileno()` returning such an integer. The three lists of waitable objects are for input, output and `exceptional conditions', respectively. Empty lists are allowed. The optional timeout argument specifies a time-out as a floating point number in seconds. When the timeout argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

The return value is a triple of lists of objects that are ready: subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.

Amongst the acceptable object types in the lists are Python file objects (e.g. `sys.stdin`, or objects returned by `open()` or `posix.popen()`), socket objects returned by `socket.socket()`, and the module `stdwin` which happens to define a function `fileno()` for just this purpose. You may also define a wrapper class yourself, as long as it has an appropriate `fileno()` method (that really returns a UNIX file descriptor, not just a random integer).

Built-in Module `thread`

This module provides low-level primitives for working with multiple threads (a.k.a. light-weight processes or tasks) -- multiple threads of control sharing their global data space. For synchronization, simple locks (a.k.a. mutexes or binary semaphores) are provided.

The module is optional and supported on SGI IRIX 4.x and 5.x and Sun Solaris 2.x systems, as well as on systems that have a PTHREAD implementation (e.g. KSR).

It defines the following constant and functions:

exception: module thread **error**

Raised on thread-specific errors.

function of module thread: **start_new_thread** *(func, arg)*

Start a new thread. The thread executes the function func with the argument list arg (which must be a tuple). When the function returns, the thread silently exits. When the function terminates with an unhandled exception, a stack trace is printed and then the thread exits (but other threads continue to run).

function of module thread: **exit** *()*

This is a shorthand for `thread.exit_thread()`.

function of module thread: **exit_thread** *()*

Raise the `SystemExit` exception. When not caught, this will cause the thread to exit silently.

function of module thread: **allocate_lock** *()*

Return a new lock object. Methods of locks are described below. The lock is initially unlocked.

function of module thread: **get_ident** *()*

Return the `thread identifier' of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

Lock objects have the following methods:

Method: lock **acquire** *([waitflag])*

Without the optional argument, this method acquires the lock unconditionally, if necessary waiting until it is released by another thread (only one thread at a time can acquire a lock -- that's their reason for existence), and returns `None`. If the integer waitflag argument is present, the action depends on its value: if it is zero, the lock is only acquired if it can be acquired immediately without waiting, while if it is nonzero, the lock is acquired unconditionally as before. If an argument is present, the return value is 1 if the lock is acquired successfully, 0 if not.

Method: lock **release** *()*

TOC

Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

Method: lock **locked** *()*

Return the status of the lock: 1 if it has been acquired by some thread, 0 if not.

Caveats:

- Threads interact strangely with interrupts: the `KeyboardInterrupt` exception will be received by an arbitrary thread. (When the `signal` module is available, interrupts always go to the main thread.)
- Calling `sys.exit()` or raising the `SystemExit` is equivalent to calling `thread.exit_thread()`.
- Not all built-in functions that may block waiting for I/O allow other threads to run. (The most popular ones (`sleep`, `read`, `select`) work as expected.)

Go to the previous, next section.

Go to the previous, next section.

UNIX Specific Services

The modules described in this chapter provide interfaces to features that are unique to the UNIX operating system, or in some cases to some or many variants of it. Here's an overview:

posix
--- The most common Posix system calls (normally used via module `os`).

posixpath
--- Common Posix pathname manipulations (normally used via `os.path`).

pwd
--- The password database (`getpwnam()` and friends).

grp
--- The group database (`getgrnam()` and friends).

dbm
--- The standard "database" interface, based on `ndbm`.

gdbm
--- GNU's reinterpretation of dbm.

termios
--- Posix style tty control.

fcntl
--- The `fcntl()` and `ioctl()` system calls.

posixfile
--- A file-like object with support for locking.

Built-in Module `posix`

This module provides access to operating system functionality that is standardized by the C Standard and the POSIX standard (a thinly disguised UNIX interface).

Do not import this module directly. Instead, import the module `os`, which provides a *portable* version of this interface. On UNIX, the `os` module provides a superset of the `posix` interface. On non-UNIX operating systems the `posix` module is not available, but a subset is always available through the `os` interface. Once `os` is imported, there is *no* performance penalty in using it instead of `posix`.

The descriptions below are very terse; refer to the corresponding UNIX manual entry for more information. Arguments called path refer to a pathname given as a string.

Errors are reported as exceptions; the usual exceptions are given for type errors, while errors reported by the system calls raise `posix.error`, described below.

Module `posix` defines the following data items:

data: module posix **environ**

A dictionary representing the string environment at the time the interpreter was started. For example, `posix.environ['HOME']` is the pathname of your home directory, equivalent to `getenv("HOME")` in C. Modifying this dictionary does not affect the string environment passed on by `execv()`, `popen()` or `system()`; if you need to change the environment, pass `environ` to `execve()` or add variable assignments and export statements to the command string for `system()` or `popen()`.(13)

exception: module posix **error**

This exception is raised when a POSIX function returns a POSIX-related error (e.g., not for illegal argument types). Its string value is `'posix.error'`. The accompanying value is a pair containing the numeric error code from `errno` and the corresponding string, as would be printed by the C function `perror()`.

It defines the following functions and constants:

function of module posix: **chdir** *(path)*

Change the current working directory to path.

function of module posix: **chmod** *(path, mode)*

Change the mode of path to the numeric mode.

function of module posix: **chown** *(path, uid, gid)*

Change the owner and group id of path to the numeric uid and gid. (Not on MS-DOS.)

function of module posix: **close** *(fd)*

Close file descriptor fd.

Note: this function is intended for low-level I/O and must be applied to a file descriptor as returned by `posix.open()` or `posix.pipe()`. To close a "file object" returned by the built-in function `open` or by `posix.popen` or `posix.fdopen`, use its `close()` method.

function of module posix: **dup** *(fd)*

Return a duplicate of file descriptor fd.

function of module posix: **dup2** *(fd, fd2)*

Duplicate file descriptor fd to fd2, closing the latter first if necessary. Return `None`.

function of module posix: **execv** *(path, args)*

Execute the executable path with argument list args, replacing the current process (i.e., the Python interpreter). The argument list may be a tuple or list of strings. (Not on MS-DOS.)

function of module posix: **execve** *(path, args, env)*

Execute the executable path with argument list args, and environment env, replacing the current process (i.e., the Python interpreter). The argument list may be a tuple or list of strings. The environment must be a dictionary mapping strings to strings. (Not on MS-DOS.)

function of module posix: **_exit** *(n)*

Exit to the system with status n, without calling cleanup handlers, flushing stdio buffers, etc. (Not on MS-DOS.)

Note: the standard way to exit is `sys.exit(n)`. `posix._exit()` should normally only be used in the child process after a `fork()`.

function of module posix: **fdopen** *(fd[, mode[, bufsize]])*

Return an open file object connected to the file descriptor fd. The mode and bufsize arguments have the same meaning as the corresponding arguments to the built-in `open()` function.

function of module posix: **fork** *()*

Fork a child process. Return 0 in the child, the child's process id in the parent. (Not on MS-DOS.)

function of module posix: **fstat** *(fd)*

Return status for file descriptor fd, like `stat()`.

function of module posix: **getcwd** *()*

Return a string representing the current working directory.

function of module posix: **getegid** *()*

Return the current process's effective group id. (Not on MS-DOS.)

function of module posix: **geteuid** *()*

Return the current process's effective user id. (Not on MS-DOS.)

function of module posix: **getgid** *()*

Return the current process's group id. (Not on MS-DOS.)

function of module posix: **getpid** *()*

Return the current process id. (Not on MS-DOS.)

function of module posix: **getppid** *()*

Return the parent's process id. (Not on MS-DOS.)

function of module posix: **getuid** *()*

Return the current process's user id. (Not on MS-DOS.)

function of module posix: **kill** *(pid, sig)*

Kill the process pid with signal sig. (Not on MS-DOS.)

function of module posix: **link** *(src, dst)*

Create a hard link pointing to src named dst. (Not on MS-DOS.)

function of module posix: **listdir** *(path)*

Return a list containing the names of the entries in the directory. The list is in arbitrary order. It includes the special entries `'.'` and `'..'` if they are present in the directory.

function of module posix: **lseek** *(fd, pos, how)*

Set the current position of file descriptor fd to position pos, modified by how: 0 to set the position relative to the beginning of the file; 1 to set it relative to the current position; 2 to set it relative to the end of the file.

function of module posix: **lstat** *(path)*

Like `stat()`, but do not follow symbolic links. (On systems without symbolic links, this is identical to `posix.stat`.)

function of module posix: **mkdir** *(path, mode)*

Create a directory named path with numeric mode mode.

function of module posix: **nice** *(increment)*

Add incr to the process' "niceness". Return the new niceness. (Not on MS-DOS.)

function of module posix: **open** *(file, flags, mode)*

Open the file file and set various flags according to flags and possibly its mode according to mode. Return the file descriptor for the newly opened file.

Note: this function is intended for low-level I/O. For normal usage, use the built-in function `open`, which returns a "file object" with `read()` and `write()` methods (and many more).

function of module posix: **pipe** *()*

Create a pipe. Return a pair of file descriptors `(r, w)` usable for reading and writing, respectively. (Not on MS-DOS.)

function of module posix: **popen** *(command[, mode[, bufsize]])*

Open a pipe to or from command. The return value is an open file object connected to the pipe, which can be read or written depending on whether mode is `'r'` (default) or `'w'`. The bufsize argument has the same meaning as the corresponding argument to the built-in `open()` function. (Not on MS-DOS.)

function of module posix: **read** *(fd, n)*

Read at most n bytes from file descriptor fd. Return a string containing the bytes read.

Note: this function is intended for low-level I/O and must be applied to a file descriptor as returned by `posix.open()` or `posix.pipe()`. To read a "file object" returned by the built-in function `open` or by `posix.popen` or `posix.fdopen`, or `sys.stdin`, use its `read()` or `readline()` methods.

function of module posix: **readlink** *(path)*

Return a string representing the path to which the symbolic link points. (On systems without symbolic links, this always raises `posix.error`.)

function of module posix: **rename** *(src, dst)*

Rename the file or directory src to dst.

function of module posix: **rmdir** *(path)*

Remove the directory path.

function of module posix: **setgid** *(gid)*

Set the current process's group id. (Not on MS-DOS.)

function of module posix: **setuid** *(uid)*

Set the current process's user id. (Not on MS-DOS.)

function of module posix: **stat** *(path)*

Perform a *stat* system call on the given path. The return value is a tuple of at least 10 integers giving the most important (and portable) members of the *stat* structure, in the order `st_mode`, `st_ino`, `st_dev`, `st_nlink`, `st_uid`, `st_gid`, `st_size`, `st_atime`, `st_mtime`, `st_ctime`. More items may be added at the end by some implementations. (On MS-DOS, some items are filled with dummy values.)

Note: The standard module `stat` defines functions and constants that are useful for extracting information from a stat structure.

function of module posix: **symlink** *(src, dst)*

Create a symbolic link pointing to src named dst. (On systems without symbolic links, this always raises `posix.error`.)

function of module posix: **system** *(command)*

Execute the command (a string) in a subshell. This is implemented by calling the Standard C function `system()`, and has the same limitations. Changes to `posix.environ`, `sys.stdin` etc. are not reflected in the environment of the executed command. The return value is the exit status of the process as returned by Standard C `system()`.

function of module posix: **times** *()*

Return a 4-tuple of floating point numbers indicating accumulated CPU times, in seconds. The items are: user time, system time, children's user time, and children's system time, in that order. See the UNIX manual page *times*(2). (Not on MS-DOS.)

TOC

function of module posix: **umask** *(mask)*

Set the current numeric umask and returns the previous umask. (Not on MS-DOS.)

function of module posix: **uname** *()*

Return a 5-tuple containing information identifying the current operating system. The tuple contains 5 strings: `(sysname, nodename, release, version, machine)`. Some systems truncate the nodename to 8 characters or to the leading component; a better way to get the hostname is `socket.gethostname()`. (Not on MS-DOS, nor on older UNIX systems.)

function of module posix: **unlink** *(path)*

Unlink path.

function of module posix: **utime** *(path, (atime, mtime))*

Set the access and modified time of the file to the given values. (The second argument is a tuple of two items.)

function of module posix: **wait** *()*

Wait for completion of a child process, and return a tuple containing its pid and exit status indication (encoded as by UNIX). (Not on MS-DOS.)

function of module posix: **waitpid** *(pid, options)*

Wait for completion of a child process given by proces id, and return a tuple containing its pid and exit status indication (encoded as by UNIX). The semantics of the call are affected by the value of the integer options, which should be 0 for normal operation. (If the system does not support `waitpid()`, this always raises `posix.error`. Not on MS-DOS.)

function of module posix: **write** *(fd, str)*

Write the string str to file descriptor fd. Return the number of bytes actually written.

Note: this function is intended for low-level I/O and must be applied to a file descriptor as returned by `posix.open()` or `posix.pipe()`. To write a "file object" returned by the built-in function `open` or by `posix.popen` or `posix.fdopen`, or `sys.stdout` or `sys.stderr`, use its `write()` method.

data: module posix **WNOHANG**

The option for `waitpid()` to avoid hanging if no child process status is available immediately.

Standard Module `posixpath`

This module implements some useful functions on POSIX pathnames.

Do not import this module directly. Instead, import the module `os` and use `os.path`.

function of module posixpath: **basename** *(p)*

Return the base name of pathname p. This is the second half of the pair returned by `posixpath.split(p)`.

function of module posixpath: **commonprefix** *(list)*

Return the longest string that is a prefix of all strings in list. If list is empty, return the empty string (").

function of module posixpath: **exists** *(p)*

Return true if p refers to an existing path.

function of module posixpath: **expanduser** *(p)*

Return the argument with an initial component of `~' or `~user' replaced by that user's home directory. An initial `~' is replaced by the environment variable `$HOME`; an initial `~user' is looked up in the password directory through the built-in module `pwd`. If the expansion fails, or if the path does not begin with a tilde, the path is returned unchanged.

function of module posixpath: **expandvars** *(p)*

Return the argument with environment variables expanded. Substrings of the form `$name' or `${name}' are replaced by the value of environment variable name. Malformed variable names and references to non-existing variables are left unchanged.

function of module posixpath: **isabs** *(p)*

Return true if p is an absolute pathname (begins with a slash).

function of module posixpath: **isfile** *(p)*

Return true if p is an existing regular file. This follows symbolic links, so both `islink()` and `isfile()` can be true for the same path.

function of module posixpath: **isdir** *(p)*

Return true if p is an existing directory. This follows symbolic links, so both `islink()` and `isdir()` can be true for the same path.

function of module posixpath: **islink** *(p)*

Return true if p refers to a directory entry that is a symbolic link. Always false if symbolic links are not supported.

function of module posixpath: **ismount** *(p)*

Return true if pathname p is a mount point: a point in a file system where a different file system has been mounted. The function checks whether p's parent, `` `p/..' ``, is on a different device than p, or whether `` `p/..' `` and p point to the same i-node on the same device -- this should detect mount points for all UNIX and POSIX variants.

function of module posixpath: **join** *(p, q)*

Join the paths p and q intelligently: If q is an absolute path, the return value is q. Otherwise, the

TOC

concatenation of p and q is returned, with a slash (`'/'`) inserted unless p is empty or ends in a slash.

function of module posixpath: **normcase** *(p)*

Normalize the case of a pathname. This returns the path unchanged; however, a similar function in `macpath` converts upper case to lower case.

function of module posixpath: **samefile** *(p, q)*

Return true if both pathname arguments refer to the same file or directory (as indicated by device number and i-node number). Raise an exception if a stat call on either pathname fails.

function of module posixpath: **split** *(p)*

Split the pathname p in a pair `(head, tail)`, where tail is the last pathname component and head is everything leading up to that. If p ends in a slash (except if it is the root), the trailing slash is removed and the operation applied to the result; otherwise, `join(head, tail)` equals p. The tail part never contains a slash. Some boundary cases: if p is the root, head equals p and tail is empty; if p is empty, both head and tail are empty; if p contains no slash, head is empty and tail equals p.

function of module posixpath: **splitext** *(p)*

Split the pathname p in a pair `(root, ext)` such that `root + ext == p`, the last component of root contains no periods, and ext is empty or begins with a period.

function of module posixpath: **walk** *(p, visit, arg)*

Calls the function visit with arguments `(arg, dirname, names)` for each directory in the directory tree rooted at p (including p itself, if it is a directory). The argument dirname specifies the visited directory, the argument names lists the files in the directory (gotten from `posix.listdir(dirname)`, so including `.' and `..'). The visit function may modify names to influence the set of directories visited below dirname, e.g., to avoid visiting certain parts of the tree. (The object referred to by names must be modified in place, using `del` or slice assignment.)

Built-in Module pwd

This module provides access to the UNIX password database. It is available on all UNIX versions.

Password database entries are reported as 7-tuples containing the following items from the password database (see `` `<pwd.h>' ``), in order: `pw_name`, `pw_passwd`, `pw_uid`, `pw_gid`, `pw_gecos`, `pw_dir`, `pw_shell`. The uid and gid items are integers, all others are strings. An exception is raised if the entry asked for cannot be found.

It defines the following items:

function of module pwd: **getpwuid** *(uid)*

Return the password database entry for the given numeric user ID.

function of module pwd: **getpwnam** *(name)*

Return the password database entry for the given user name.

function of module pwd: **getpwall** *()*

Return a list of all available password database entries, in arbitrary order.

Built-in Module grp

This module provides access to the UNIX group database. It is available on all UNIX versions.

Group database entries are reported as 4-tuples containing the following items from the group database (see `` `<grp.h>' ``), in order: `gr_name`, `gr_passwd`, `gr_gid`, `gr_mem`. The gid is an integer, name and password are strings, and the member list is a list of strings. (Note that most users are not explicitly listed as members of the group they are in according to the password database.) An exception is raised if the entry asked for cannot be found.

It defines the following items:

function of module grp: **getgrgid** *(gid)*

Return the group database entry for the given numeric group ID.

function of module grp: **getgrnam** *(name)*

Return the group database entry for the given group name.

function of module grp: **getgrall** *()*

Return a list of all available group entries, in arbitrary order.

Built-in Module dbm

Dbm provides python programs with an interface to the unix `ndbm` database library. Dbm objects are of the mapping type, so they can be handled just like objects of the built-in dictionary type, except that keys and values are always strings, and printing a dbm object doesn't print the keys and values.

The module defines the following constant and functions:

exception: module dbm **error**

Raised on dbm-specific errors, such as I/O errors. `KeyError` is raised for general mapping errors like specifying an incorrect key.

function of module dbm: **open** *(filename, rwmode, filemode)*

Open a dbm database and return a mapping object. filename is the name of the database file (without the `` `.dir' `` or `` `.pag' `` extensions), rwmode is `'r'`, `'w'` or `'rw'` to open the database fore reading, writing or both respectively, and filemode is the UNIX mode of the file, used only when the database has to be created (but to be supplied at all times).

Built-in Module gdbm

Gdbm provides python programs with an interface to the GNU `gdbm` database library. Gdbm objects are of the mapping type, so they can be handled just like objects of the built-in dictionary type, except that keys and values are always strings, and printing a gdbm object doesn't print the keys and values.

The module is based on the Dbm module, modified to use GDBM instead.

The module defines the following constant and functions:

exception: module gdbm **error**

Raised on gdbm-specific errors, such as I/O errors. `KeyError` is raised for general mapping errors like specifying an incorrect key.

function of module gdbm: **open** *(filename, rwmode, filemode)*

Open a gdbm database and return a mapping object. filename is the name of the database file, rwmode is `'r'`, `'w'`, `'c'`, or `'n'` for reader, writer (this also gives read access), create (writer, but create the database if it doesn't already exist) and newdb (which will always create a new database). Only one writer may open a gdbm file and many readers may open the file. Readers and writers cannot open the gdbm file at the same time. Note that the `GDBM_FAST` mode of opening the database is not supported. filemode is the UNIX mode of the file, used only when a database is created (but to be supplied at all times).

Built-in Module termios

This module provides an interface to the Posix calls for tty I/O control. For a complete description of these calls, see the Posix or UNIX manual pages. It is only available for those UNIX versions that support Posix `termios` style tty I/O control (and then only if configured at installation time).

All functions in this module take a file descriptor fd as their first argument. This must be an integer file descriptor, such as returned by `sys.stdin.fileno()`.

This module should be used in conjunction with the `TERMIOS` module, which defines the relevant symbolic constants (see the next section).

The module defines the following functions:

function of module termios: **tcgetattr** *(fd)*

Return a list containing the tty attributes for file descriptor fd, as follows: `[iflag, oflag, cflag, lflag, ispeed, ospeed, cc]` where cc is a list of the tty special characters (each a string of length 1, except the items with indices `VMIN` and `VTIME`, which are integers when these fields are defined). The interpretation of the flags and the speeds as well as the indexing in the cc array must be done using the symbolic constants defined in the `TERMIOS` module.

function of module termios: **tcsetattr** *(fd, when, attributes)*

Set the tty attributes for file descriptor fd from the attributes, which is a list like the one returned by `tcgetattr()`. The when argument determines when the attributes are changed: `TERMIOS.TCSA` to change immediately, `TERMIOS.TCSADRAIN` to change after transmitting all queued output, or `TERMIOS.TCSAFLUSH` to change after transmitting all queued output and discarding all queued input.

function of module termios: **tcsendbreak** *(fd, duration)*

Send a break on file descriptor fd. A zero duration sends a break for 0.25--0.5 seconds; a nonzero duration has a system dependent meaning.

function of module termios: **tcdrain** *(fd)*

Wait until all output written to file descriptor fd has been transmitted.

function of module termios: **tcflush** *(fd, queue)*

Discard queued data on file descriptor fd. The queue selector specifies which queue: `TERMIOS.TCIFLUSH` for the input queue, `TERMIOS.TCOFLUSH` for the output queue, or `TERMIOS.TCIOFLUSH` for both queues.

function of module termios: **tcflow** *(fd, action)*

Suspend or resume input or output on file descriptor fd. The action argument can be `TERMIOS.TCOOFF` to suspend output, `TERMIOS.TCOON` to restart output, `TERMIOS.TCIOFF` to suspend input, or `TERMIOS.TCION` to restart input.

Example

Here's a function that prompts for a password with echoing turned off. Note the technique using a separate `termios.tcgetattr()` call and a `try {...` finally} statement to ensure that the old tty attributes are restored exactly no matter what happens:

```
def getpass(prompt = "Password: "):
    import termios, TERMIOS, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~TERMIOS.ECHO          # lflags
    try:
        termios.tcsetattr(fd, TERMIOS.TCSADRAIN, new)
        passwd = raw_input(prompt)
    finally:
        termios.tcsetattr(fd, TERMIOS.TCSADRAIN, old)
    return passwd
```

Standard Module TERMIOS

This module defines the symbolic constants required to use the `termios` module (see the previous section). See the Posix or UNIX manual pages (or the source) for a list of those constants.

Note: this module resides in a system-dependent subdirectory of the Python library directory. You may have to generate it for your particular system using the script `` `Tools/scripts/h2py.py' ``.

Built-in Module fcntl

This module performs file control and I/O control on file descriptors. It is an interface to the fcntl() and ioctl() UNIX routines. File descriptors can be obtained with the fileno() method of a file or socket object.

The module defines the following functions:

function of module struct: **fcntl** *(fd, op[, arg])*

Perform the requested operation on file descriptor `fd`. The operation is defined by `op` and is operating system dependent. Typically these codes can be retrieved from the library module `FCNTL`. The argument `arg` is optional, and defaults to the integer value `0`. When it is present, it can either be an integer value, or a string. With the argument missing or an integer value, the return value of this function is the integer return value of the real `fcntl()` call. When the argument is a string it represents a binary structure, e.g. created by `struct.pack()`. The binary data is copied to a buffer whose address is passed to the real `fcntl()` call. The return value after a successful call is the contents of the buffer, converted to a string object. In case the `fcntl()` fails, an `IOError` will be raised.

function of module struct: **ioctl** *(fd, op, arg)*

This function is identical to the `fcntl()` function, except that the operations are typically defined in the library module `IOCTL`.

If the library modules `FCNTL` or `IOCTL` are missing, you can find the opcodes in the C include files `sys/fcntl` and `sys/ioctl`. You can create the modules yourself with the h2py script, found in the `Demo/scripts` directory.

Examples (all on a SVR4 compliant system):

```
import struct, FCNTL

file = open(...)
rv = fcntl(file.fileno(), FCNTL.O_NDELAY, 1)

lockdata = struct.pack('hhllhh', FCNTL.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl(file.fileno(), FCNTL.F_SETLKW, lockdata)
```

Note that in the first example the return value variable `rv` will hold an integer value; in the second example it will hold a string value.

Standard Module posixfile

This module implements some additional functionality over the built-in file objects. In particular, it implements file locking, control over the file flags, and an easy interface to duplicate the file object. The module defines a new file object, the posixfile object. It has all the standard file object methods and adds the methods described below. This module only works for certain flavors of UNIX, since it uses `fcntl()` for file locking.

To instantiate a posixfile object, use the `open()` function in the posixfile module. The resulting object looks and feels roughly the same as a standard file object.

The posixfile module defines the following constants:

data: module posixfile **SEEK_SET**

offset is calculated from the start of the file

data: module posixfile **SEEK_CUR**

offset is calculated from the current position in the file

data: module posixfile **SEEK_END**

offset is calculated from the end of the file

The posixfile module defines the following functions:

function of module posixfile: **open** *(filename[, mode[, bufsize]])*

Create a new posixfile object with the given filename and mode. The filename, mode and bufsize arguments are interpreted the same way as by the built-in `open()` function.

function of module posixfile: **fileopen** *(fileobject)*

Create a new posixfile object with the given standard file object. The resulting object has the same filename and mode as the original file object.

The posixfile object defines the following additional methods:

Method: posixfile **lock** *(fmt, [len[, start [, whence]]])*

Lock the specified section of the file that the file object is referring to. The format is explained below in a table. The len argument specifies the length of the section that should be locked. The default is `0`. start specifies the starting offset of the section, where the default is `0`. The whence argument specifies where the offset is relative to. It accepts one of the constants `SEEK_SET`, `SEEK_CUR` or `SEEK_END`. The default is `SEEK_SET`. For more information about the arguments refer to the fcntl manual page on your system.

Method: posixfile **flags** *([flags])*

Set the specified flags for the file that the file object is referring to. The new flags are ORed with the old

flags, unless specified otherwise. The format is explained below in a table. Without the flags argume string indicating the current flags is returned (this is the same as the '?' modifier). For more informati about the flags refer to the fcntl manual page on your system.

Method: posixfile **dup** *()*

Duplicate the file object and the underlying file pointer and file descriptor. The resulting object behaves as if it were newly opened.

Method: posixfile **dup2** *(fd)*

Duplicate the file object and the underlying file pointer and file descriptor. The new object will have the given file descriptor. Otherwise the resulting object behaves as if it were newly opened.

Method: posixfile **file** *()*

Return the standard file object that the posixfile object is based on. This is sometimes necessary for functions that insist on a standard file object.

All methods return `IOError` when the request fails.

Format characters for the `lock()` method have the following meaning:

Format

Meaning --

`u'

unlock the specified region

`r'

request a read lock for the specified section

`w'

request a write lock for the specified section

In addition the following modifiers can be added to the format:

Modifier

Meaning -- *Notes*

`|'

wait until the lock has been granted

`?'

return the first lock conflicting with the requested lock, or `None` if there is no conflict. -- (1)

Note:

(1) The lock returned is in the format `(mode, len, start, whence, pid)` where mode is a character representing the type of lock ('r' or 'w'). This modifier prevents a request from being granted; it is for query purposes only.

Format character for the `flags()` method have the following meaning:

Format

Meaning --

`a'

append only flag

`c'

close on exec flag

`n'

no delay flag (also called non-blocking flag)

`s'

synchronization flag

In addition the following modifiers can be added to the format:

Modifier

Meaning -- *Notes*

`!'

turn the specified flags 'off', instead of the default 'on' -- (1)

`='

replace the flags, instead of the default 'OR' operation -- (1)

`?'

return a string in which the characters represent the flags that are set. -- (2)

Note:

(1) The `!` and `=` modifiers are mutually exclusive.

(2) This string represents the flags after they may have been altered by the same call.

Examples:

```
from posixfile import *

file = open('/tmp/test', 'w')
file.lock('w|')
...
file.lock('u')
file.close()
```

Go to the previous, next section.

Go to the previous, next section.

The Python Debugger

The module `pdb` defines an interactive source code debugger for Python programs. It supports setting breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame. It also supports post-mortem debugging and can be called under program control.

The debugger is extensible -- it is actually defined as a class `Pdb`. This is currently undocumented but easily understood by reading the source. The extension interface uses the (also undocumented) modules `bdb` and `cmd`.

A primitive windowing version of the debugger also exists -- this is module `wdb`, which requires STDWIN (see the chapter on STDWIN specific modules).

The debugger's prompt is "`(Pdb) `". Typical usage to run a program under control of the debugger is:

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

Typical usage to inspect a crashed program is:

```
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "./mymodule.py", line 4, in test
    test2()
  File "./mymodule.py", line 3, in test2
    print spam
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print spam
(Pdb)
```

The module defines the following functions; each enters the debugger in a slightly different way:

function of module pdb: **run** *(statement[, globals[, locals]])*

Execute the statement (given as a string) under debugger control. The debugger prompt appears before any code is executed; you can set breakpoints and type `continue`, or you can step through the statement using `step` or `next` (all these commands are explained below). The optional globals and locals arguments specify the environment in which the code is executed; by default the dictionary of the module `__main__` is used. (See the explanation of the `exec` statement or the `eval()` built-in function.)

function of module pdb: **runeval** *(expression[, globals[, locals]])*

Evaluate the expression (given as a a string) under debugger control. When `runeval()` returns, it returns the value of the expression. Otherwise this function is similar to `run()`.

function of module pdb: **runcall** *(function[, argument, ...])*

Call the function (a function or method object, not a string) with the given arguments. When `runcall()` returns, it returns whatever the function call returned. The debugger prompt appears as soon as the function is entered.

function of module pdb: **set_trace** *()*

Enter the debugger at the calling stack frame. This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails).

function of module pdb: **post_mortem** *(traceback)*

Enter post-mortem debugging of the given traceback object.

function of module pdb: **pm** *()*

Enter post-mortem debugging of the traceback found in `sys.last_traceback`.

Debugger Commands

The debugger recognizes the following commands. Most commands can be abbreviated to one or two letters; e.g. "`h(elp)`" means that either "`h`" or "`help`" can be used to enter the help command (but not "`he`" or "`hel`", nor "`H`" or "`Help` or "`HELP`"). Arguments to commands must be separated by whitespace (spaces or tabs). Optional arguments are enclosed in square brackets ("`[]`") in the command syntax; the square brackets must not be typed. Alternatives in the command syntax are separated by a vertical bar ("`|`").

Entering a blank line repeats the last command entered. Exception: if the last command was a "`list`" command, the next 11 lines are listed.

Commands that the debugger doesn't recognize are assumed to be Python statements and are executed in the context of the program being debugged. Python statements can also be prefixed with an exclamation point ("`!`"). This is a powerful way to inspect the program being debugged; it is even possible to change

a variable or call a function. When an exception occurs in such a statement, the exception name is pr
but the debugger's state is not changed.

h(elp) [command

]

Without argument, print the list of available commands. With a command as argument, print help about that command. "`help pdb`" displays the full documentation file; if the environment variable `PAGER` is defined, the file is piped through that command instead. Since the command argument must be an identifier, "`help exec`" must be entered to get help on the "`!`" command.

w(here)

Print a stack trace, with the most recent frame at the bottom. An arrow indicates the current frame, which determines the context of most commands.

d(own)

Move the current frame one level down in the stack trace (to an older frame).

u(p)

Move the current frame one level up in the stack trace (to a newer frame).

b(reak) [lineno | function

]

With a lineno argument, set a break there in the current file. With a function argument, set a break at the entry of that function. Without argument, list all breaks.

cl(ear) [lineno

]

With a lineno argument, clear that break in the current file. Without argument, clear all breaks (but first ask confirmation).

s(tep)

Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).

n(ext)

Continue execution until the next line in the current function is reached or it returns. (The difference between `next` and `step` is that `step` stops inside a called function, while `next` executes called functions at (nearly) full speed, only stopping at the next line in the current function.)

r(eturn)

Continue execution until the current function returns.

c(ont(inue))

Continue execution, only stop when a breakpoint is encountered.

l(ist) [first [, last

]]

List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing. With one argument, list 11 lines around at that line. With two arguments, list the given range; if the second argument is less than the first, it is interpreted as a count.

a(rgs)

Print the argument list of the current function.

p expression

Evaluate the expression in the current context and print its value. (Note: `print` can also be used, but is not a debugger command -- this executes the Python `print` statement.)

[!

statement]

Execute the (one-line) statement in the context of the current stack frame. The exclamation point can be omitted unless the first word of the statement resembles a debugger command. To set a global variable, you can prefix the assignment command with a "`global`" command on the same line, e.g.:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

q(uit)

Quit from the debugger. The program being executed is aborted.

How It Works

Some changes were made to the interpreter:

- sys.settrace(func) sets the global trace function
- there can also a local trace function (see later)

Trace functions have three arguments: (frame, event, arg)

frame

is the current stack frame

event

is a string: `'call'`, `'line'`, `'return'` or `'exception'`

arg

is dependent on the event type

A trace function should return a new trace function or None. Class methods are accepted (and most useful!) as trace methods.

The events have the following meaning:

`'call'`

A function is called (or some other code block entered). The global trace function is called; arg is the argument list to the function; the return value specifies the local trace function.

`'line'`

The interpreter is about to execute a new line of code (sometimes multiple line events on one line exist). The local trace function is called; arg in None; the return value specifies the new local trace function.

`'return'`

A function (or other code block) is about to return. The local trace function is called; arg is the value that will be returned. The trace function's return value is ignored.

`'exception'`

An exception has occurred. The local trace function is called; arg is a triple (exception, value, traceback); the return value specifies the new local trace function

Note that as an exception is propagated down the chain of callers, an `'exception'` event is generated at each level.

Stack frame objects have the following read-only attributes:

f_code

the code object being executed

f_lineno

the current line number (`-1` for `'call'` events)

f_back

the stack frame of the caller, or None

f_locals

dictionary containing local name bindings

f_globals

dictionary containing global name bindings

Code objects have the following read-only attributes:

co_code

the code string

co_names

the list of names used by the code

co_consts

the list of (literal) constants used by the code

co_filename

the filename from which the code was compiled

TOC

Go to the previous, next section.

Go to the previous, next section.

The Python Profiler

Written by James Roskind(14)

The profiler was written after only programming in Python for 3 weeks. As a result, it is probably clumsy code, but I don't know for sure yet 'cause I'm a beginner :-). I did work hard to make the code run fast, so that profiling would be a reasonable thing to do. I tried not to repeat code fragments, but I'm sure I did some stuff in really awkward ways at times. Please send suggestions for improvements to: `jar@infoseek.com`. I won't promise *any* support. ...but I'd appreciate the feedback.

Introduction to the profiler

A profiler is a program that describes the run time performance of a program, providing a variety of statistics. This documentation describes the profiler functionality provided in the modules `profile` and `pstats`. This profiler provides deterministic profiling of any Python programs. It also provides a series of report generation tools to allow users to rapidly examine the results of a profile operation.

How Is This Profiler Different From The Old Profiler?

The big changes from old profiling module are that you get more information, and you pay less CPU time. It's not a trade-off, it's a trade-up.

To be specific:

Bugs removed:

Local stack frame is no longer molested, execution time is now charged to correct functions.

Accuracy increased:

Profiler execution time is no longer charged to user's code, calibration for platform is supported, file reads are not done *by* profiler *during* profiling (and charged to user's code!).

Speed increased:

Overhead CPU cost was reduced by more than a factor of two (perhaps a factor of five), lightweight profiler module is all that must be loaded, and the report generating module (`pstats`) is not needed during profiling.

Recursive functions support:

Cumulative times in recursive functions are correctly calculated; recursive entries are counted.

Large growth in report generating UI:

Distinct profiles runs can be added together forming a comprehensive report; functions that import statistics take arbitrary lists of files; sorting criteria is now based on keywords (instead of 4 integer options); reports shows what functions were profiled as well as what profile file was referenced; output format has been improved.

Instant Users Manual

This section is provided for users that "don't want to read the manual." It provides a very brief overview, and allows a user to rapidly perform profiling on an existing application.

To profile an application with a main entry point of `foo()', you would add the following to your module:

```
import profile
profile.run("foo()")
```

The above action would cause `foo()' to be run, and a series of informative lines (the profile) to be printed. The above approach is most useful when working with the interpreter. If you would like to save the results of a profile into a file for later examination, you can supply a file name as the second argument to the `run()` function:

```
import profile
profile.run("foo()", 'fooprof')
```

When you wish to review the profile, you should use the methods in the `pstats` module. Typically you would load the statistics data as follows:

```
import pstats
p = pstats.Stats('fooprof')
```

The class `Stats` (the above code just created an instance of this class) has a variety of methods for

manipulating and printing the data that was just read into `p'. When you ran `profile.run()` above what was printed was the result of three method calls:

```
p.strip_dirs().sort_stats(-1).print_stats()
```

The first method removed the extraneous path from all the module names. The second method sorted all the entries according to the standard module/line/name string that is printed (this is to comply with the semantics of the old profiler). The third method printed out all the statistics. You might try the following sort calls:

```
p.sort_stats('name')
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

```
p.sort_stats('cumulative').print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

```
p.sort_stats('time').print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

```
p.sort_stats('file').print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods ('cause they are spelled with `__init__` in them). As one final example, you could try:

```
p.sort_stats('time', 'cum').print_stats(.5, 'init')
```

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: `.5') of its original size, then only lines containing `init` are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (`p' is still sorted according to the last criteria) do:

```
p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If you want more functionality, you're going to have to read the manual, or guess what the following

functions do:

```
p.print_callees()
p.add('fooprof')
```

What Is Deterministic Profiling?

Deterministic profiling is meant to reflect the fact that all function call, function return, and exception events are monitored, and precise timings are made for the intervals between these events (during which time the user's code is executing). In contrast, statistical profiling (which is not done by this module) randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required to do deterministic profiling. Python automatically provides a hook (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run time statistics about the execution of a Python program.

Call count statistics can be used to identify bugs in code (surprising counts), and to identify possible inline-expansion points (high call counts). Internal time statistics can be used to identify "hot loops" that should be carefully optimized. Cumulative time statistics should be used to identify high level errors in the selection of algorithms. Note that the unusual handling of cumulative times in this profiler allows statistics for recursive implementations of algorithms to be directly compared to iterative implementations.

Reference Manual

The primary entry point for the profiler is the global function `profile.run()`. It is typically used to create any profile information. The reports are formatted and printed using methods of the class `pstats.Stats`. The following is a description of all of these standard entry points and functions. For a more in-depth view of some of the code, consider reading the later section on Profiler Extensions, which includes discussion of how to derive "better" profilers from the classes presented, or reading the source code for these modules.

profiler function: **profile.run** *(string[, filename[, ...]])*

This function takes a single argument that has can be passed to the `exec` statement, and an optional file name. In all cases this routine attempts to `exec` its first argument, and gather profiling statistics from the execution. If no file name is present, then this function automatically prints a simple profiling report, sorted by the standard name string (file/line/function-name) that is presented in each line. The following is a typical output from such a call:

@small{


```
      main()
      2706 function calls (2004 primitive calls) in 4.504 CPU seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     2    0.006    0.003    0.953    0.477 pobject.py:75(save_objects)
  43/3    0.533    0.012    0.749    0.250 pobject.py:99(evaluate)
  ...
}
```

The first line indicates that this profile was generated by the call: `profile.run('main()')`, and hence the exec'ed string is `'main()'`. The second line indicates that 2706 calls were monitored. Of those calls, 2004 were primitive. We define primitive to mean that the call was not induced via recursion. The next line: `Ordered by: standard name`, indicates that the text string in the far right column was used to sort the output. The column headings include:

ncalls

for the number of calls,

tottime

for the total time spent in the given function (and excluding time made in calls to sub-functions),

percall

is the quotient of `tottime` divided by `ncalls`

cumtime

is the total time spent in this and all subfunctions (i.e., from invocation till exit). This figure is accurate *even* for recursive functions.

percall

is the quotient of `cumtime` divided by primitive calls

filename:lineno(function)

provides the respective data of each function

When there are two numbers in the first column (e.g.: `43/3'), then the latter is the number of primitive calls, and the former is the actual number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

profiler function: **pstats.Stats** *(filename[, ...])*

This class constructor creates an instance of a "statistics object" from a filename (or set of filenames). `Stats` objects are manipulated by methods, in order to print useful reports.

The file selected by the above constructor must have been created by the corresponding version of `profile`. To be specific, there is *NO* file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers (e.g., the old system profiler).

If several files are provided, all the statistics for identical functions will be coalesced, so that an overview of several processes can be considered in a single report. If additional files need to be combined with data in an existing `Stats` object, the `add()` method can be used.

The `Stats` Class

Method: Stats **strip_dirs** *()*

This method for the `Stats` class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the stripped information is lost. After performing a strip operation, the object is considered to have its entries in a "random" order, as it was just after object initialization and loading. If `strip_dirs()` causes two function names to be indistinguishable (i.e., they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

Method: Stats **add** *(filename[, ...])*

This method of the `Stats` class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of `profile.run()`. Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

Method: Stats **sort_stats** *(key[, ...])*

This method modifies the `Stats` object by sorting it according to the supplied criteria. The argument is typically a string identifying the basis of a sort (example: `"time"` or `"name"`).

When more than one key is provided, then additional keys are used as secondary criteria when the there is equality in all keys selected before them. For example, sort_stats('name', 'file') will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

Abbreviations can be used for any key names, as long as the abbreviation is unambiguous. The following are the keys currently defined:

Valid Arg

-- *Meaning*

`"calls"`

call count

`"cumulative"`

cumulative time

`"file"`

file name

`"module"`

file name

`"pcalls"`

primitive call count

`"line"`
: line number

`"name"`
: function name

`"nfl"`
: name/file/line

`"stdname"`
: standard name

`"time"`
: internal time

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (i.e., alphabetical). The subtle distinction between `"nfl"` and `"stdname"` is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, `"nfl"` does a numeric compare of the line numbers. In fact, `sort_stats("nfl")` is the same as `sort_stats("name", "file", "line")`.

For compatibility with the old profiler, the numeric arguments `-1', `0', `1', and `2' are permitted. They are interpreted as `"stdname"`, `"calls"`, `"time"`, and `"cumulative"` respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

Method: Stats **reverse_order** *()*

This method for the `Stats` class reverses the ordering of the basic list within the object. This method is provided primarily for compatibility with the old profiler. Its utility is questionable now that ascending vs descending order is properly selected based on the sort key of choice.

Method: Stats **print_stats** *(restriction[, ...])*

This method for the `Stats` class prints out a report as described in the `profile.run()` definition.

The order of the printing is based on the last `sort_stats()` operation done on the object (subject to caveats in `add()` and `strip_dirs()`).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a regular expression (to pattern match the standard name that is printed). If several restrictions are provided, then they are applied sequentially. For example:

```
print_stats(.1, "foo:")
```

would first limit the printing to first 10% of list, and then only print functions that were part of filena `.*foo:'. In contrast, the command:

```
print_stats("foo:", .1)
```

would limit the list to all functions having file names `.*foo:', and then proceed to only print the first 10% of them.

Method: Stats **print_callers** *(restrictions[, ...])*

This method for the `Stats` class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by `print_stats()`, and the definition of the restricting argument is also identical. For convenience, a number is shown in parentheses after each caller to show how many times this specific call was made. A second non-parenthesized number is the cumulative time spent in the function at the right.

Method: Stats **print_callees** *(restrictions[, ...])*

This method for the `Stats` class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to the `print_callers()` method.

Method: Stats **ignore** *()*

This method of the `Stats` class is used to dispose of the value returned by earlier methods. All standard methods in this class return the instance that is being processed, so that the commands can be strung together. For example:

```
pstats.Stats('foofile').strip_dirs().sort_stats('cum') \
                       .print_stats().ignore()
```

would perform all the indicated functions, but it would not return the final reference to the `Stats` instance.(15)

Limitations

There are two fundamental limitations on this profiler. The first is that it relies on the Python interpreter to dispatch call, return, and exception events. Compiled C code does not get interpreted, and hence is "invisible" to the profiler. All time spent in C code (including builtin functions) will be charged to the Python function that invoked the C code. If the C code calls out to some native Python code, then those calls will be profiled properly.

The second limitation has to do with accuracy of timing information. There is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying "clock" is only ticking at a rate (typically) of about .001 seconds. Hence no measurements will be more accurate that that underlying clock. If enough measurements are taken, then the "error" will tend to average out. Unfortunately, removing this first error induces a second source of error...

The second problem is that it "takes a while" from when an event is dispatched until the profiler's call to get the time actually *gets* the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the time that the clock's value was obtained (and then squirreled away), until the user's code is once again executing. As a result, functions that are called many times, or call many functions, will typically accumulate this error. The error that accumulates in this fashion is typically less than the accuracy of the clock (i.e., less than one clock tick), but it *can* accumulate and become very significant. This profiler provides a means of calibrating itself for a given platform so that this error can be probabilistically (i.e., on the average) removed. After the profiler is calibrated, it will be more accurate (in a least square sense), but it will sometimes produce negative numbers (when call counts are exceptionally low, and the gods of probability work against you :-).) Do *NOT* be alarmed by negative numbers in the profile. They should *only* appear if you have calibrated your profiler, and the results are actually better than without calibration.

Calibration

The profiler class has a hard coded constant that is added to each event handling time to compensate for the overhead of calling the time function, and socking away the results. The following procedure can be used to obtain this constant for a given platform (see discussion in section Limitations above).

```
import profile
pr = profile.Profile()
pr.calibrate(100)
pr.calibrate(100)
pr.calibrate(100)
```

The argument to calibrate() is the number of times to try to do the sample calls to get the CPU times. If your computer is *very* fast, you might have to do:

```
pr.calibrate(1000)
```

or even:

```
pr.calibrate(10000)
```

The object of this exercise is to get a fairly consistent result. When you have a consistent answer, you are ready to use that number in the source code. For a Sun Sparcstation 1000 running Solaris 2.3, the magical number is about .00053. If you have a choice, you are better off with a smaller constant, and your results will "less often" show up as negative in profile statistics.

The following shows how the trace_dispatch() method in the Profile class should be modified to install the calibration constant on a Sun Sparcstation 1000:

```
def trace_dispatch(self, frame, event, arg):
    t = self.timer()
    t = t[0] + t[1] - self.t - .00053 # Calibration constant
```


```
        if self.dispatch[event](frame,t):
            t = self.timer()
            self.t = t[0] + t[1]
        else:
            r = self.timer()
            self.t = r[0] + r[1] - t # put back unrecorded delta
        return
```

Note that if there is no calibration constant, then the line containing the callibration constant should simply say:

```
        t = t[0] + t[1] - self.t  # no calibration constant
```

You can also achieve the same results using a derived class (and the profiler will actually run equally fast!!), but the above method is the simplest to use. I could have made the profiler "self calibrating", but it would have made the initialization of the profiler class slower, and would have required some *very* fancy coding, or else the use of a variable where the constant `.00053' was placed in the code shown. This is a **VERY** critical performance section, and there is no reason to use a variable lookup at this point, when a constant can be used.

Extensions -- Deriving Better Profilers

The `Profile` class of module `profile` was written so that derived classes could be developed to extend the profiler. Rather than describing all the details of such an effort, I'll just present the following two examples of derived classes that can be used to do profiling. If the reader is an avid Python programmer, then it should be possible to use these as a model and create similar (and perchance better) profile classes.

If all you want to do is change how the timer is called, or which timer function is used, then the basic class has an option for that in the constructor for the class. Consider passing the name of a function to call into the constructor:

```
    pr = profile.Profile(your_time_func)
```

The resulting profiler will call `your_time_func()` instead of `os.times()`. The function should return either a single number or a list of numbers (like what `os.times()` returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you *should* calibrate the profiler class for the timer function that you choose. For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (os.times is *pretty* bad, 'cause it returns a tuple of floating point values, so all arithmetic is floating point in the profiler!). If you want to substitute a better timer in the cleanest fashion, you should derive a class, and simply put in the replacement dispatch method that better handles your timer call, along with the appropriate calibration constant :-).

OldProfile Class

The following derived profiler simulates the old style profiler, providing errant results on recursive functions. The reason for the usefulness of this profiler is that it runs faster (i.e., less overhead) than the old profiler. It still creates all the caller stats, and is quite useful when there is *no* recursion in the user's code. It is also a lot more accurate than the old profiler, as it does not charge all its overhead time to the user's code.

```
class OldProfile(Profile):

    def trace_dispatch_exception(self, frame, t):
        rt, rtt, rct, rfn, rframe, rcur = self.cur
        if rcur and not rframe is frame:
            return self.trace_dispatch_return(rframe, t)
        return 0

    def trace_dispatch_call(self, frame, t):
        fn = `frame.f_code`

        self.cur = (t, 0, 0, fn, frame, self.cur)
        if self.timings.has_key(fn):
            tt, ct, callers = self.timings[fn]
            self.timings[fn] = tt, ct, callers
        else:
            self.timings[fn] = 0, 0, {}
        return 1

    def trace_dispatch_return(self, frame, t):
        rt, rtt, rct, rfn, frame, rcur = self.cur
        rtt = rtt + t
        sft = rtt + rct

        pt, ptt, pct, pfn, pframe, pcur = rcur
        self.cur = pt, ptt+rt, pct+sft, pfn, pframe, pcur

        tt, ct, callers = self.timings[rfn]
        if callers.has_key(pfn):
            callers[pfn] = callers[pfn] + 1
        else:
            callers[pfn] = 1
        self.timings[rfn] = tt+rtt, ct + sft, callers

        return 1
```


```
    def snapshot_stats(self):
        self.stats = {}
        for func in self.timings.keys():
            tt, ct, callers = self.timings[func]
            nor_func = self.func_normalize(func)
            nor_callers = {}
            nc = 0
            for func_caller in callers.keys():
                nor_callers[self.func_normalize(func_caller)]=\
                      callers[func_caller]
                nc = nc + callers[func_caller]
            self.stats[nor_func] = nc, nc, tt, ct, nor_callers
```

HotProfile Class

This profiler is the fastest derived profile example. It does not calculate caller-callee relationships, and does not calculate cumulative time under a function. It only calculates time spent in a function, so it runs very quickly (re: very low overhead). In truth, the basic profiler is so fast, that is probably not worth the savings to give up the data, but this class still provides a nice example.

```
class HotProfile(Profile):

    def trace_dispatch_exception(self, frame, t):
        rt, rtt, rfn, rframe, rcur = self.cur
        if rcur and not rframe is frame:
            return self.trace_dispatch_return(rframe, t)
        return 0

    def trace_dispatch_call(self, frame, t):
        self.cur = (t, 0, frame, self.cur)
        return 1

    def trace_dispatch_return(self, frame, t):
        rt, rtt, frame, rcur = self.cur

        rfn = `frame.f_code`

        pt, ptt, pframe, pcur = rcur
        self.cur = pt, ptt+rt, pframe, pcur

        if self.timings.has_key(rfn):
            nc, tt = self.timings[rfn]
            self.timings[rfn] = nc + 1, rt + rtt + tt
        else:
            self.timings[rfn] =      1, rt + rtt
```


```
        return 1


    def snapshot_stats(self):
        self.stats = {}
        for func in self.timings.keys():
            nc, tt = self.timings[func]
            nor_func = self.func_normalize(func)
            self.stats[nor_func] = nc, nc, tt, 0, {}
```

Go to the previous, next section.

Go to the previous, next section.

Internet and WWW Services

The modules described in this chapter provide various services to World-Wide Web (WWW) clients and/or services, and a few modules related to news and email. They are all implemented in Python. Some of these modules require the presence of the system-dependent module `sockets`, which is currently only fully supported on Unix and Windows NT. Here is an overview:

cgi
--- Common Gateway Interface, used to interpret forms in server-side scripts.

urllib
--- Open an arbitrary object given by URL (requires sockets).

httplib
--- HTTP protocol client (requires sockets).

ftplib
--- FTP protocol client (requires sockets).

gopherlib
--- Gopher protocol client (requires sockets).

nntplib
--- NNTP protocol client (requires sockets).

urlparse
--- Parse a URL string into a tuple (addressing scheme identifier, network location, path, parameters, query string, fragment identifier).

htmllib
--- A (slow) parser for HTML files.

sgmllib
--- Only as much of an SGML parser as needed to parse HTML.

rfc822
--- Parse RFC-822 style mail headers.

mimetools
--- Tools for parsing MIME style message bodies.

Standard Module `cgi`

This module makes it easy to write Python scripts that run in a WWW server using the Common Gateway Interface. It was written by Michael McLay and subsequently modified by Steve Majewski and Guido van Rossum.

When a WWW server finds that a URL contains a reference to a file in a particular subdirectory (usually `/cgibin`), it runs the file as a subprocess. Information about the request such as the full URL, the originating host etc., is passed to the subprocess in the shell environment; additional input from the client may be read from

standard input. Standard output from the subprocess is sent back across the network to the client as the resp from the request. The CGI protocol describes what the environment variables passed to the subprocess mea and how the output should be formatted. The official reference documentation for the CGI protocol can be found on the World-Wide Web at `<URL:http://hoohoo.ncsa.uiuc.edu/cgi/overview.html>`. The `cgi` module was based on version 1.1 of the protocol and should also work with version 1.0.

The `cgi` module defines several classes that make it easy to access the information passed to the subprocess from a Python script; in particular, it knows how to parse the input sent by an HTML "form" using either a POST or a GET request (these are alternatives for submitting forms in the HTTP protocol).

The formatting of the output is so trivial that no additional support is needed. All you need to do is print a minimal set of MIME headers describing the output format, followed by a blank line and your actual output. E.g. if you want to generate HTML, your script could start as follows:

```
# Header -- one or more lines:
print "Content-type: text/html"
# Blank line separating header from body:
print
# Body, in HTML format:
print "<TITLE>The Amazing SPAM Homepage!</TITLE>"
# etc...
```

The server will add some header lines of its own, but it won't touch the output following the header.

The `cgi` module defines the following functions:

function of module cgi: **parse** *()*

Read and parse the form submitted to the script and return a dictionary containing the form's fields. This should be called at most once per script invocation, as it may consume standard input (if the form was submitted through a POST request). The keys in the resulting dictionary are the field names used in the submission; the values are *lists* of the field values (since field name may be used multiple times in a single form). `%' escapes in the values are translated to their single-character equivalent using `urllib.unquote()`. As a side effect, this function sets `environ['QUERY_STRING']` to the raw query string, if it isn't already set.

function of module cgi: **print_environ_usage** *()*

Print a piece of HTML listing the environment variables that may be set by the CGI protocol. This is mainly useful when learning about writing CGI scripts.

function of module cgi: **print_environ** *()*

Print a piece of HTML text showing the entire contents of the shell environment. This is mainly useful when debugging a CGI script.

function of module cgi: **print_form** *(form)*

Print a piece of HTML text showing the contents of the form (a dictionary, an instance of the `FormContentDict` class defined below, or a subclass thereof). This is mainly useful when debugging a CGI script.

function of module cgi: **escape** *(string)*

Convert special characters in string to HTML escapes. In particular, "&" is replaced with "`&`", "<" is replaced with "`<`", and ">" is replaced with "`>`". This is useful when printing (almost) arbitrary text in an HTML context. Note that for inclusion in quoted tag attributes (e.g.), some additional characters would have to be converted -- in particular the string quote. There is currently no function that does this.

The module defines the following classes. Since the base class initializes itself by calling `parse()`, at most one instance of at most one of these classes should be created per script invocation:

function of module cgi: **FormContentDict** *()*

This class behaves like a (read-only) dictionary and has the same keys and values as the dictionary returned by `parse()` (i.e. each field name maps to a list of values). Additionally, it initializes its data member `query_string` to the raw query sent from the server.

function of module cgi: **SvFormContentDict** *()*

This class, derived from `FormContentDict`, is a little more user-friendly when you are expecting that each field name is only used once in the form. When you access for a particular field (using `form[fieldname]`), it will return the string value of that item if it is unique, or raise `IndexError` if the field was specified more than once in the form. (If the field wasn't specified at all, `KeyError` is raised.) To access fields that are specified multiple times, use `form.getlist(fieldname)`. The `values()` and `items()` methods return mixed lists --- containing strings for singly-defined fields, and lists of strings for multiply-defined fields.

(It currently defines some more classes, but these are experimental and/or obsolescent, and are thus not documented -- see the source for more informations.)

The module defines the following variable:

data: module cgi **environ**

The shell environment, exactly as received from the http server. See the CGI documentation for a description of the various fields.

Example

This example assumes that you have a WWW server up and running, e.g. NCSA's `httpd`.

Place the following file in a convenient spot in the WWW server's directory tree. E.g., if you place it in the subdirectory `` `test' `` of the root directory and call it `` `test.html' ``, its URL will be `` `http://yourservername/test/test.html' ``.

```
<TITLE>Test Form Input</TITLE>
<H1>Test Form Input</H1>
<FORM METHOD="POST" ACTION="/cgi-bin/test.py">
<INPUT NAME=Name> (Name)<br>
<INPUT NAME=Address> (Address)<br>
<INPUT TYPE=SUBMIT>
</FORM>
```

Selecting this file's URL from a forms-capable browser such as Mosaic or Netscape will bring up a simple form with two text input fields and a "submit" button.

But wait. Before pressing "submit", a script that responds to the form must also be installed. The test file as shown assumes that the script is called `test.py' and lives in the server's cgi-bin directory. Here's the test script:

```
#!/usr/local/bin/python

import cgi

print "Content-type: text/html"
print                                   # End of headers!
print "<TITLE>Test Form Output</TITLE>"
print "<H1>Test Form Output</H1>"

form = cgi.SvFormContentDict()          # Load the form

name = addr = None                      # Default: no name and address

# Extract name and address from the form, if given

if form.has_key('Name'):
        name = form['Name']
if form.has_key('Address'):
        addr = form['Address']

# Print an unnumbered list of the name and address, if present

print "<UL>"
if name is not None:
        print "<LI>Name:", cgi.escape(name)
if addr is not None:
        print "<LI>Address:", cgi.escape(addr)
print "</UL>"
```

The script should be made executable (`chmod +x script'). If the Python interpreter is not located at `/usr/local/bin/python' but somewhere else, the first line of the script should be modified accordingly.

Now that everything is installed correctly, we can try out the form. Bring up the test form in your WWW browser, fill in a name and address in the form, and press the "submit" button. The script should now run and its output is sent back to your browser. This should roughly look as follows:

Test Form Output

- Name: the name you entered
- Address: the address you entered

If you didn't enter a name or address, the corresponding line will be missing (since the browser doesn't send empty form fields to the server).

Standard Module `urllib`

This module provides a high-level interface for fetching data across the World-Wide Web. In particular, the `urlopen` function is similar to the built-in function `open`, but accepts URLs (Universal Resource Locators) instead of filenames. Some restrictions apply -- it can only open URLs for reading, and no seek operations are available.

it defines the following public functions:

function of module urllib: **urlopen** *(url)*

Open a network object denoted by a URL for reading. If the URL does not have a scheme identifier, or if it has `file:' as its scheme identifier, this opens a local file; otherwise it opens a socket to a server somewhere on the network. If the connection cannot be made, or if the server returns an error code, the `IOError` exception is raised. If all went well, a file-like object is returned. This supports the following methods: `read()`, `readline()`, `readlines()`, `fileno()`, `close()` and `info()`. Except for the last one, these methods have the same interface as for file objects -- see the section on File Objects earlier in this manual. (It's not a built-in file object, however, so it can't be used at those few places where a true built-in file object is required.)

The `info()` method returns an instance of the class `rfc822.Message` containing the headers received from the server, if the protocol uses such headers (currently the only supported protocol that uses this is HTTP). See the description of the `rfc822` module.

function of module urllib: **urlretrieve** *(url)*

Copy a network object denoted by a URL to a local file, if necessary. If the URL points to a local file, or a valid cached copy of the object exists, the object is not copied. Return a tuple (filename, headers) where filename is the local file name under which the object can be found, and headers is either `None` (for a local object) or whatever the `info()` method of the object returned by `urlopen()` returned (for a remote object, possibly cached). Exceptions are the same as for `urlopen()`.

function of module urllib: **urlcleanup** *()*

Clear the cache that may have been built up by previous calls to `urlretrieve()`.

function of module urllib: **quote** *(string[, addsafe])*

Replace special characters in string using the `%xx` escape. Letters, digits, and the characters "`_,.-`" are never quoted. The optional addsafe parameter specifies additional characters that should not be quoted -- its default value is `'/'`.

Example: `quote('/~conolly/')` yields `'/%7econnolly/'`.

function of module urllib: **unquote** *(string)*

Replace `%xx' escapes by their single-character equivalent.

Example: `unquote('/%7Econnolly/')` yields `'/~connolly/'`.

Restrictions:

- Currently, only the following protocols are supported: HTTP, (versions 0.9 and 1.0), Gopher (but not Gopher-+), FTP, and local files.
- The caching feature of `urlretrieve()` has been disabled until I find the time to hack proper

processing of Expiration time headers.

- There should be a function to query whether a particular URL is in the cache.
- For backward compatibility, if a URL appears to point to a local file but the file can't be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.
- The `urlopen()` and `urlretrieve()` functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive web client using these functions without using threads.
- The data returned by `urlopen()` or `urlretrieve()` is the raw data returned by the server. This may be binary data (e.g. an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the `Content-type` header. For the Gopher protocol, type information is encoded in the URL; there is currently no easy way to extract it. If the returned data is HTML, you can use the module `htmllib` to parse it.
- Although the `urllib` module contains (undocumented) routines to parse and unparse URL strings, the recommended interface for URL manipulation is in module `urlparse`.

Standard Module `httplib`

This module defines a class which implements the client side of the HTTP protocol. It is normally not used directly -- the module `urllib` uses it to handle URLs that use HTTP.

The module defines one class, `HTTP`. An `HTTP` instance represents one transaction with an HTTP server. It should be instantiated passing it a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form `host:port`, else the default HTTP port (80) is used. If no host is passed, no connection is made, and the `connect` method should be used to connect to a server. For example, the following calls all create instances that connect to the server at the same host and port:

```
>>> h1 = httplib.HTTP('www.cwi.nl')
>>> h2 = httplib.HTTP('www.cwi.nl:80')
>>> h3 = httplib.HTTP('www.cwi.nl', 80)
```

Once an `HTTP` instance has been connected to an HTTP server, it should be used as follows:

1. 1. Make exactly one call to the `putrequest()` method.
2. 2. Make zero or more calls to the `putheader()` method.
3. 3. Call the `endheaders()` method (this can be omitted if step 4 makes no calls).
4. 4. Optional calls to the `send()` method.
5. 5. Call the `getreply()` method.
6. 6. Call the `getfile()` method and read the data off the file object that it returns.

HTTP Objects

`HTTP` instances have the following methods:

Method: HTTP **set_debuglevel** *(level)*

Set the debugging level (the amount of debugging output printed). The default debug level is `0`, meaning no debugging output is printed.

Method: HTTP **connect** *(host[, port])*

Connect to the server given by host and port. See the intro for the default port. This should be called directly only if the instance was instantiated without passing a host.

Method: HTTP **send** *(data)*

Send data to the server. This should be used directly only after the `endheaders()` method has been called and before `getreply()` has been called.

Method: HTTP **putrequest** *(request, selector)*

This should be the first call after the connection to the server has been made. It sends a line to the server consisting of the request string, the selector string, and the HTTP version (`HTTP/1.0`).

Method: HTTP **putheader** *(header, argument[, ...])*

Send an RFC-822 style header to the server. It sends a line to the server consisting of the header, a colon and a space, and the first argument. If more arguments are given, continuation lines are sent, each consisting of a tab and an argument.

Method: HTTP **endheaders** *()*

Send a blank line to the server, signalling the end of the headers.

Method: HTTP **getreply** *()*

Complete the request by shutting down the sending end of the socket, read the reply from the server, and return a triple (replycode, message, headers). Here replycode is the integer reply code from the request (e.g. `200` if the request was handled properly); message is the message string corresponding to the reply code; and header is an instance of the class `rfc822.Message` containing the headers received from the server. See the description of the `rfc822` module.

Method: HTTP **getfile** *()*

Return a file object from which the data returned by the server can be read, using the `read()`, `readline()` or `readlines()` methods.

Example

Here is an example session:

```
>>> import httplib
>>> h = httplib.HTTP('www.cwi.nl')
>>> h.putrequest('GET', '/index.html')
>>> h.putheader('Accept', 'text/html')
>>> h.putheader('Accept', 'text/plain')
>>> h.endheaders()
>>> errcode, errmsg, headers = h.getreply()
>>> print errcode # Should be 200
>>> f = h.getfile()
>>> data f.read() # Get the raw HTML
>>> f.close()
```

>>>

Standard Module `ftplib`

This module defines the class `FTP` and a few related items. The `FTP` class implements the client side of the FTP protocol. You can use this to write Python programs that perform a variety of automated FTP jobs, such as mirroring other ftp servers. It is also used bu the module `urllib` to handle URLs that use FTP. For more information on FTP (File Transfer Protocol), see Internet RFC 959.

Here's a sample session using the `ftplib` module:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.cwi.nl')   # connect to host, default port
>>> ftp.login()               # user anonymous, passwd user@hostname
>>> ftp.retrlines('LIST')     # list directory contents
total 24418
drwxrwsr-x   5 ftp-usr  pdmaint     1536 Mar 20 09:48 .
dr-xr-srwt 105 ftp-usr  pdmaint     1536 Mar 21 14:32 ..
-rw-r--r--   1 ftp-usr  pdmaint     5305 Mar 20 09:48 INDEX
 .
 .
 .
>>> ftp.quit()
```

The module defines the following items:

function of module ftplib: **FTP** *([host[, user, passwd, acct]])*

Return a new instance of the `FTP` class. When host is given, the method call `connect(host)` is made. When user is given, additionally the method call `login(user, passwd, acct)` is made (where passwd and acct default to the empty string when not given).

data: module ftplib **all_errors**

The set of all exceptions (as a tuple) that methods of `FTP` instances may raise as a result of problems with the FTP connection (as opposed to programming errors made by the caller). This set includes the four exceptions listed below as well as `socket.error` and `IOError`.

exception: module ftplib **error_reply**

Exception raised when an unexpected reply is received from the server.

exception: module ftplib **error_temp**

Exception raised when an error code in the range 400--499 is received.

exception: module ftplib **error_perm**

Exception raised when an error code in the range 500--599 is received.

exception: module ftplib **error_proto**

Exception raised when a reply is received from the server that does not begin with a digit in the range 1--5.

FTP Objects

FTP instances have the following methods:

Method: FTP object **set_debuglevel** *(level)*

Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

Method: FTP object **connect** *(host[, port])*

Connect to the given host and port. The default port number is 21, as specified by the FTP protocol specification. It is rarely needed to specify a different port number. This function should be called only once for each instance; it should not be called at all if a host was given when the instance was created. All other methods can only be used after a connection has been made.

Method: FTP object **getwelcome** *()*

Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

Method: FTP object **login** *([user[, passwd[, acct]]])*

Log in as the given user. The passwd and acct parameters are optional and default to the empty string. If no user is specified, it defaults to `anonymous'. If user is `anonymous`, the default passwd is `realuser@host' where realuser is the real user name (glanced from the `LOGNAME' or `USER' environment variable) and host is the hostname as returned by `socket.gethostname()`. This function should be called only once for each instance, after a connection has been established; it should not be called at all if a host and user were given when the instance was created. Most FTP commands are only allowed after the client has logged in.

Method: FTP object **abort** *()*

Abort a file transfer that is in progress. Using this does not always work, but it's worth a try.

Method: FTP object **sendcmd** *(command)*

Send a simple command string to the server and return the response string.

Method: FTP object **voidcmd** *(command)*

Send a simple command string to the server and handle the response. Return nothing if a response code in the range 200--299 is received. Raise an exception otherwise.

Method: FTP object **retrbinary** *(command, callback, maxblocksize)*

Retrieve a file in binary transfer mode. command should be an appropriate `RETR' command, i.e. `"RETR filename"`. The callback function is called for each block of data received, with a single string argument giving the data block. The maxblocksize argument specifies the maximum block size (which may not be the actual size of the data blocks passed to callback).

Method: FTP object **retrlines** *(command[, callback])*

TOC

Retrieve a file or directory listing in ASCII transfer mode. var{command} should be an appropriate `RETR' command (see `retrbinary()` or a `LIST' command (usually just the string `"LIST"`). The callback function is called for each line, with the trailing CRLF stripped. The default callback prints the line to `sys.stdout`.

Method: FTP object **storbinary** *(command, file, blocksize)*

Store a file in binary transfer mode. command should be an appropriate `STOR' command, i.e. `"STOR filename"`. file is an open file object which is read until EOF using its `read()` method in blocks of size blocksize to provide the data to be stored.

Method: FTP object **storlines** *(command, file)*

Store a file in ASCII transfer mode. command should be an appropriate `STOR' command (see `storbinary()`). Lines are read until EOF from the open file object file using its `readline()` method to privide the data to be stored.

Method: FTP object **nlst** *(argument[, ...])*

Return a list of files as returned by the `NLST' command. The optional var{argument} is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `NLST' command.

Method: FTP object **dir** *(argument[, ...])*

Return a directory listing as returned by the `LIST' command, as a list of lines. The optional var{argument} is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `LIST' command. If the last argument is a function, it is used as a callback function as for `retrlines()`.

Method: FTP object **rename** *(fromname, toname)*

Rename file fromname on the server to toname.

Method: FTP object **cwd** *(pathname)*

Set the current directory on the server.

Method: FTP object **mkd** *(pathname)*

Create a new directory on the server.

Method: FTP object **pwd** *()*

Return the pathname of the current directory on the server.

Method: FTP object **quit** *()*

Send a `QUIT' command to the server and close the connection. This is the "polite" way to close a connection, but it may raise an exception of the server reponds with an error to the `QUIT` command.

Method: FTP object **close** *()*

Close the connection unilaterally. This should not be applied to an already closed connection (e.g. after a successful call to `quit()`.

Standard Module gopherlib

This module provides a minimal implementation of client side of the the Gopher protocol. It is used by the module `urllib` to handle URLs that use the Gopher protocol.

The module defines the following functions:

function of module gopherlib: **send_selector** *(selector, host[, port])*

Send a selector string to the gopher server at host and port (default 70). Return an open file object from which the returned document can be read.

function of module gopherlib: **send_query** *(selector, query, host[, port])*

Send a selector string and a query string to a gopher server at host and port (default 70). Return an open file object from which the returned document can be read.

Note that the data returned by the Gopher server can be of any type, depending on the first character of the selector string. If the data is text (first character of the selector is `0'), lines are terminated by CRLF, and the data is terminated by a line consisting of a single `.', and a leading `.' should be stripped from lines that begin with `..'. Directory listings (first charactger of the selector is `1') are transferred using the same protocol.

Standard Module nntplib

This module defines the class `NNTP` which implements the client side of the NNTP protocol. It can be used to implement a news reader or poster, or automated news processors. For more information on NNTP (Network News Transfer Protocol), see Internet RFC 977.

Here are two small examples of how it can be used. To list some statistics about a newsgroup and print the subjects of the last 10 articles:

@small{

```
>>> s = NNTP('news.cwi.nl')
>>> resp, count, first, last, name = s.group('comp.lang.python')
>>> print 'Group', name, 'has', count, 'articles, range', first, 'to', last
Group comp.lang.python has 59 articles, range 3742 to 3803
>>> resp, subs = s.xhdr('subject', first + '-' + last)
>>> for id, sub in subs[-10:]: print id, sub
...
3792 Re: Removing elements from a list while iterating...
3793 Re: Who likes Info files?
3794 Emacs and doc strings
3795 a few questions about the Mac implementation
3796 Re: executable python scripts
3797 Re: executable python scripts
3798 Re: a few questions about the Mac implementation
3799 Re: PROPOSAL: A Generic Python Object Interface for Python C Modules
3802 Re: executable python scripts
3803 Re: POSIX wait and SIGCHLD
```


```
>>> s.quit()
'205 news.cwi.nl closing connection.  Goodbye.'
>>>
```

}

To post an article from a file (this assumes that the article has valid headers):

```
>>> s = NNTP('news.cwi.nl')
>>> f = open('/tmp/article')
>>> s.post(f)
'240 Article posted successfully.'
>>> s.quit()
'205 news.cwi.nl closing connection.  Goodbye.'
>>>
```

The module itself defines the following items:

function of module nntplib: **NNTP** *(host[, port])*

Return a new instance of the `NNTP` class, representing a connection to the NNTP server running on host host, listening at port port. The default port is 119.

exception: module nntplib **error_reply**

Exception raised when an unexpected reply is received from the server.

exception: module nntplib **error_temp**

Exception raised when an error code in the range 400--499 is received.

exception: module nntplib **error_perm**

Exception raised when an error code in the range 500--599 is received.

exception: module nntplib **error_proto**

Exception raised when a reply is received from the server that does not begin with a digit in the range 1--5.

NNTP Objects

NNTP instances have the following methods. The response that is returned as the first item in the return tuple of almost all methods is the server's response: a string beginning with a three-digit code. If the server's response indicates an error, the method raises one of the above exceptions.

Method: NNTP object **getwelcome** *()*

Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

Method: NNTP object **set_debuglevel** *(level)*

Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request or response. A value of 2 or higher produces the maximum amount of debugging output,

logging each line sent and received on the connection (including message text).

Method: NNTP object **newgroups** *(date, time)*

Send a `NEWGROUPS' command. The date argument should be a string of the form `"yymmdd"` indicating the date, and time should be a string of the form `"hhmmss"` indicating the time. Return a pair `(response, groups)` where groups is a list of group names that are new since the given date and time.

Method: NNTP object **newnews** *(group, date, time)*

Send a `NEWNEWS' command. Here, group is a group name or `"*"`, and date and time have the same meaning as for `newgroups()`. Return a pair `(response, articles)` where articles is a list of article ids.

Method: NNTP object **list** *()*

Send a `LIST' command. Return a pair `(response, list)` where list is a list of tuples. Each tuple has the form `(group, last, first, flag)`, where group is a group name, last and first are the last and first article numbers (as strings), and flag is `'y'` if posting is allowed, `'n'` if not, and `'m'` if the newsgroup is moderated. (Note the ordering: last, first.)

Method: NNTP object **group** *(name)*

Send a `GROUP' command, where name is the group name. Return a tuple `(response, count, first, last, name)` where count is the (estimated) number of articles in the group, first is the first article number in the group, last is the last article number in the group, and name is the group name. The numbers are returned as strings.

Method: NNTP object **help** *()*

Send a `HELP' command. Return a pair `(response, list)` where list is a list of help strings.

Method: NNTP object **stat** *(id)*

Send a `STAT' command, where id is the message id (enclosed in `<' and `>') or an article number (as a string). Return a triple `(var{response`, number, id)} where number is the article number (as a string) and id is the article id (enclosed in `<' and `>').

Method: NNTP object **next** *()*

Send a `NEXT' command. Return as for `stat()`.

Method: NNTP object **last** *()*

Send a `LAST' command. Return as for `stat()`.

Method: NNTP object **head** *(id)*

Send a `HEAD' command, where id has the same meaning as for `stat()`. Return a pair `(response, list)` where list is a list of the article's headers (an uninterpreted list of lines, without trailing newlines).

Method: NNTP object **body** *(id)*

Send a `BODY' command, where id has the same meaning as for `stat()`. Return a pair `(response, list)` where list is a list of the article's body text (an uninterpreted list of lines, without trailing newlines).

Method: NNTP object **article** *(id)*

Send a `ARTICLE' command, where id has the same meaning as for `stat()`. Return a pair `(response, list)` where list is a list of the article's header and body text (an uninterpreted list of lines, without trailing newlines).

Method: NNTP object **slave** *()*

Send a `SLAVE' command. Return the server's response.

Method: NNTP object **xhdr** *(header, string)*

Send an `XHDR' command. This command is not defined in the RFC but is a common extension. The header argument is a header keyword, e.g. `"subject"`. The string argument should have the form `"first-last"` where first and last are the first and last article numbers to search. Return a pair `(response, list)`, where list is a list of pairs `(id, text)`, where id is an article id (as a string) and text is the text of the requested header for that article.

Method: NNTP object **post** *(file)*

Post an article using the `POST' command. The file argument is an open file object which is read until EOF using its `readline()` method. It should be a well-formed news article, including the required headers. The `post()` method automatically escapes lines beginning with `.'.

Method: NNTP object **ihave** *(id, file)*

Send an `IHAVE' command. If the response is not an error, treat file exactly as for the `post()` method.

Method: NNTP object **quit** *()*

Send a `QUIT' command and close the connection. Once this method has been called, no other methods of the NNTP object should be called.

Standard Module `urlparse`

This module defines a standard interface to break URL strings up in components (addessing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a "relative URL" to an absolute URL given a "base URL".

The module has been designed to match the current Internet draft on Relative Uniform Resource Locators (and discovered a bug in an earlier draft!).

It defines the following functions:

function of module urlparse: **urlparse** *(urlstring[, default_scheme[, allow_fragments]])*

Parse a URL into 6 components, returning a 6-tuple: (addressing scheme, network location, path, parameters, query, fragment identifier). This corresponds to the general structure of a URL: `scheme://netloc/path;parameters?query#fragment`. Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (e.g. the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the tuple items, except for a leading slash in the path component, which is retained if present.

Example:


```
urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
```

yields the tuple

```
('http', 'www.cwi.nl:80', '/%7Eguido/Python.html', '', '', '')
```

If the default_scheme argument is specified, it gives the default addressing scheme, to be used only if the URL string does not specify one. The default value for this argument is the empty string.

If the allow_fragments argument is zero, fragment identifiers are not allowed, even if the URL's addressing scheme normally does support them. The default value for this argument is `1`.

function of module urlparse: **urlunparse** *(tuple)*

Construct a URL string from a tuple as returned by `urlparse`. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had redundant delimiters, e.g. a ? with an empty query (the draft states that these are equivalent).

function of module urlparse: **urljoin** *(base, url[, allow_fragments])*

Construct a full ("absolute") URL by combining a "base URL" (base) with a "relative URL" (url). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL.

Example:

```
urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
```

yields the string

```
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

The allow_fragments argument has the same meaning as for `urlparse`.

Standard Module `htmllib`

This module defines a number of classes which can serve as a basis for parsing text files formatted in HTML (HyperText Mark-up Language). The classes are not directly concerned with I/O -- the have to be fed their input in string form, and will make calls to methods of a "formatter" object in order to produce output. The classes are designed to be used as base classes for other classes in order to add functionality, and allow most of their methods to be extended or overridden. In turn, the classes are derived from and extend the class `SGMLParser` defined in module `sgmllib`.

The following is a summary of the interface defined by `sgmllib.SGMLParser`:

- The interface to feed data to an instance is through the `feed()` method, which takes a string argument. This can be called with as little or as much text at a time as desired; `p.feed(a); p.feed(b)` has the same effect as `p.feed(a+b)`. When the data contains complete HTML elements, these are processed immediately; incomplete elements are saved in a buffer. To force processing of all unprocessed data, call the `close()` method.

  Example: to parse the entire contents of a file, do

`parser.feed(open(file).read()); parser.close()`.

- The interface to define semantics for HTML tags is very simple: derive a class and define methods called `start_tag()`, `end_tag()`, or `do_tag()`. The parser will call these at appropriate moments: `start_tag` or `do_tag` is called when an opening tag of the form `<tag ...>` is encountered; `end_tag` is called when a closing tag of the form `<tag>` is encountered. If an opening tag requires a corresponding closing tag, like `<H1>` ... `</H1>`, the class should define the `start_tag` method; if a tag requires no closing tag, like `<P>`, the class should define the `do_tag` method.

The module defines the following classes:

function of module htmllib: **HTMLParser** *()*

This is the most basic HTML parser class. It defines one additional entity name over the names defined by the `SGMLParser` base class, `&bullet;`. It also defines handlers for the following tags: `<LISTING>...</LISTING>`, `<XMP>...</XMP>`, and `<PLAINTEXT>` (the latter is terminated only by end of file).

function of module htmllib: **CollectingParser** *()*

This class, derived from `HTMLParser`, collects various useful bits of information from the HTML text. To this end it defines additional handlers for the following tags: `<A>...`, `<HEAD>...</HEAD>`, `<BODY>...</BODY>`, `<TITLE>...</TITLE>`, `<NEXTID>`, and `<ISINDEX>`.

function of module htmllib: **FormattingParser** *(formatter, stylesheet)*

This class, derived from `CollectingParser`, interprets a wide selection of HTML tags so it can produce formatted output from the parsed data. It is initialized with two objects, a formatter which should define a number of methods to format text into paragraphs, and a stylesheet which defines a number of static parameters for the formatting process. Formatters and style sheets are documented later in this section.

function of module htmllib: **AnchoringParser** *(formatter, stylesheet)*

This class, derived from `FormattingParser`, extends the handling of the `<A>...` tag pair to call the formatter's `bgn_anchor()` and `end_anchor()` methods. This allows the formatter to display the anchor in a different font or color, etc.

Instances of `CollectingParser` (and thus also instances of `FormattingParser` and `AnchoringParser`) have the following instance variables:

data: module htmllib **anchornames**

A list of the values of the `NAME` attributes of the `<A>` tags encountered.

data: module htmllib **anchors**

A list of the values of `HREF` attributes of the `<A>` tags encountered.

data: module htmllib **anchortypes**

A list of the values of the `TYPE` attributes of the `<A>` tags encountered.

data: module htmllib **inanchor**

Outside an `<A>...` tag pair, this is zero. Inside such a pair, it is a unique integer, which is positive if the anchor has a `HREF` attribute, negative if it hasn't. Its absolute value is one more than the index of the anchor in

the `anchors`, `anchornames` and `anchortypes` lists.

data: module htmllib **isindex**

True if the `<ISINDEX>` tag has been encountered.

data: module htmllib **nextid**

The attribute list of the last `<NEXTID>` tag encountered, or an empty list if none.

data: module htmllib **title**

The text inside the last `<TITLE>...</TITLE>` tag pair, or `''` if no title has been encountered yet.

The `anchors`, `anchornames` and `anchortypes` lists are "parallel arrays": items in these lists with the same index pertain to the same anchor. Missing attributes default to the empty string. Anchors with neither a `HREF` nor a `NAME` attribute are not entered in these lists at all.

The module also defines a number of style sheet classes. These should never be instantiated -- their class variables are the only behavior required. Note that style sheets are specifically designed for a particular formatter implementation. The currently defined style sheets are:

data: module htmllib **NullStylesheet**

A style sheet for use on a dumb output device such as an ASCII terminal.

data: module htmllib **X11Stylesheet**

A style sheet for use with an X11 server.

data: module htmllib **MacStylesheet**

A style sheet for use on Apple Macintosh computers.

data: module htmllib **StdwinStylesheet**

A style sheet for use with the `stdwin` module; it is an alias for either `X11Stylesheet` or `MacStylesheet`.

data: module htmllib **GLStylesheet**

A style sheet for use with the SGI Graphics Library and its font manager (the SGI-specific built-in modules `gl` and `fm`).

Style sheets have the following class variables:

data: module htmllib **stdfontset**

A list of up to four font definititions, respectively for the roman, italic, bold and constant-width variant of a font for normal text. If the list contains less than four font definitions, the last item is used as the default for missing items. The type of a font definition depends on the formatter in use; its only use is as a parameter to the formatter's `setfont()` method.

data: module htmllib **h1fontset**

data: module htmllib **h2fontset**

data: module htmllib **h3fontset**

The font set used for various headers (text inside `<H1>...</H1>` tag pairs etc.).

data: module htmllib **stdindent**

The indentation of normal text. This is measured in the "native" units of the formatter in use; for some formatters these are characters, for others (especially those that actually support variable-spacing fonts) in pixels or printer points.

data: module htmllib **ddindent**

The indentation used for the first level of `<DD>` tags.

data: module htmllib **ulindent**

The indentation used for the first level of `<UL>` tags.

data: module htmllib **h1indent**

The indentation used for level 1 headers.

data: module htmllib **h2indent**

The indentation used for level 2 headers.

data: module htmllib **literalindent**

The indentation used for literal text (text inside `<PRE>...</PRE>` and similar tag pairs).

Although no documented implementation of a formatter exists, the `FormattingParser` class assumes that formatters have a certain interface. This interface requires the following methods:

function of module htmllib: **setfont** *(fontspec)*

Set the font to be used subsequently. The fontspec argument is an item in a style sheet's font set.

function of module htmllib: **flush** *()*

Finish the current line, if not empty, and begin a new one.

function of module htmllib: **setleftindent** *(n)*

Set the left indentation of the following lines to n units.

function of module htmllib: **needvspace** *(n)*

Require at least n blank lines before the next line. Implies `flush()`.

function of module htmllib: **addword** *(word, space)*

Add a word to the current paragraph, followed by space spaces.

data: module htmllib **nospace**

If this instance variable is true, empty words should be ignored by `addword`. It should be set to false after a non-empty word has been added.

function of module htmllib: **setjust** *(justification)*

Set the justification of the current paragraph. The justification can be `'c'` (center), `'l'` (left justified), `'r'` (right justified) or `'lr'` (left and right justified).

function of module htmllib: **bgn_anchor** *(id)*

Begin an anchor. The id parameter is the value of the parser's `inanchor` attribute.

function of module htmllib: **end_anchor** *(id)*

End an anchor. The id parameter is the value of the parser's `inanchor` attribute.

A sample formatter implementation can be found in the module `fmt`, which in turn uses the module `Para`. These modules are not intended as standard library modules; they are available as an example of how to write a formatter.

Standard Module `sgmllib`

This module defines a class `SGMLParser` which serves as the basis for parsing text files formatted in SGML (Standard Generalized Mark-up Language). In fact, it does not provide a full SGML parser --- it only parses SGML insofar as it is used by HTML, and the module only exists as a basis for the `htmllib` module.

In particular, the parser is hardcoded to recognize the following elements:

- Opening and closing tags of the form "`<tag attr="value" ...>`" and "`</tag>`", respectively.
- Character references of the form "`&#name;`".
- Entity references of the form "`&name;`".
- SGML comments of the form "`<!--text>`".

The `SGMLParser` class must be instantiated without arguments. It has the following interface methods:

function of module sgmllib: **reset** *()*

Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.

function of module sgmllib: **setnomoretags** *()*

Stop processing tags. Treat all following input as literal input (CDATA). (This is only provided so the HTML tag `<PLAINTEXT>` can be implemented.)

function of module sgmllib: **setliteral** *()*

Enter literal mode (CDATA mode).

function of module sgmllib: **feed** *(data)*

Feed some text to the parser. It is processed insofar as it consists of complete elements; incomplete data is buffered until more data is fed or `close()` is called.

function of module sgmllib: **close** *()*

Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call `SGMLParser.close()`.

function of module sgmllib: **handle_charref** *(ref)*

This method is called to process a character reference of the form "`&#ref;`" where ref is a decimal number in the range 0-255. It translates the character to ASCII and calls the method `handle_data()` with the character as argument. If ref is invalid or out of range, the method `unknown_charref(ref)` is called instead.

function of module sgmllib: **handle_entityref** *(ref)*

This method is called to process an entity reference of the form "`&ref;`" where ref is an alphabetic entity reference. It looks for ref in the instance (or class) variable `entitydefs` which should give the entity's translation. If a translation is found, it calls the method `handle_data()` with the translation; otherwise, it calls the method `unknown_entityref(ref)`.

function of module sgmllib: **handle_data** *(data)*

This method is called to process arbitrary data. It is intended to be overridden by a derived class; the base class implementation does nothing.

function of module sgmllib: **unknown_starttag** *(tag, attributes)*

This method is called to process an unknown start tag. It is intended to be overridden by a derived class; the base class implementation does nothing. The attributes argument is a list of (name, value) pairs containing the attributes found inside the tag's <> brackets. The name has been translated to lower case and double quotes and backslashes in the value have been interpreted. For instance, for the tag , this method would be called as `unknown_starttag('a', [('href', 'http://www.cwi.nl/')])`.

function of module sgmllib: **unknown_endtag** *(tag)*

This method is called to process an unknown end tag. It is intended to be overridden by a derived class; the base class implementation does nothing.

function of module sgmllib: **unknown_charref** *(ref)*

This method is called to process an unknown character reference. It is intended to be overridden by a derived class; the base class implementation does nothing.

function of module sgmllib: **unknown_entityref** *(ref)*

This method is called to process an unknown entity reference. It is intended to be overridden by a derived class; the base class implementation does nothing.

Apart from overriding or extending the methods listed above, derived classes may also define methods of the following form to define processing of specific tags. Tag names in the input stream are case independent; the tag occurring in method names must be in lower case:

function of module sgmllib: **start_tag** *(attributes)*

This method is called to process an opening tag tag. It has preference over `do_tag()`. The attributes argument has the same meaning as described for `unknown_tag()` above.

function of module sgmllib: **do_tag** *(attributes)*

This method is called to process an opening tag tag that does not come with a matching closing tag. The attributes argument has the same meaning as described for `unknown_tag()` above.

function of module sgmllib: **end_tag** *()*

This method is called to process a closing tag tag.

Note that the parser maintains a stack of opening tags for which no matching closing tag has been found yet. Only tags processed by `start_tag()` are pushed on this stack. Definition of a `end_tag()` method is optional for these tags. For tags processed by `do_tag()` or by `unknown_tag()`, no `end_tag()` method must be defined.

Standard Module `rfc822`

This module defines a class, `Message`, which represents a collection of "email headers" as defined by the Internet standard RFC 822. It is used in various contexts, usually to read such headers from a file.

A `Message` instance is instantiated with an open file object as parameter. Instantiation reads headers from the file up to a blank line and stores them in the instance; after instantiation, the file is positioned directly after the blank line that terminates the headers.

Input lines as read from the file may either be terminated by CR-LF or by a single linefeed; a terminating CR-LF is replaced by a single linefeed before the line is stored.

All header matching is done independent of upper or lower case; e.g. `m['From']`, `m['from']` and `m['FROM']` all yield the same result.

Message Objects

A `Message` instance has the following methods:

function of module rfc822: **rewindbody** *()*

Seek to the start of the message body. This only works if the file object is seekable.

function of module rfc822: **getallmatchingheaders** *(name)*

Return a list of lines consisting of all headers matching name, if any. Each physical line, whether it is a continuation line or not, is a separate list item. Return the empty list if no header matches name.

function of module rfc822: **getfirstmatchingheader** *(name)*

Return a list of lines comprising the first header matching name, and its continuation line(s), if any. Return `None` if there is no header matching name.

function of module rfc822: **getrawheader** *(name)*

Return a single string consisting of the text after the colon in the first header matching name. This includes leading whitespace, the trailing linefeed, and internal linefeeds and whitespace if there any continuation line(s) were present. Return `None` if there is no header matching name.

function of module rfc822: **getheader** *(name)*

Like `getrawheader(name)`, but strip leading and trailing whitespace (but not internal whitespace).

function of module rfc822: **getaddr** *(name)*

Return a pair (full name, email address) parsed from the string returned by `getheader(name)`. If no he matching name exists, return `None, None`; otherwise both the full name and the address are (possibly em)strings.

Example: If `m`'s first `From` header contains the string `'jack@cwi.nl (Jack Jansen)'`, then `m.getaddr('From')` will yield the pair `('Jack Jansen', 'jack@cwi.nl')`. If the header contained `'Jack Jansen <jack@cwi.nl>'` instead, it would yield the exact same result.

function of module rfc822: **getaddrlist** *(name)*

This is similar to `getaddr(list)`, but parses a header containing a list of email addresses (e.g. a `To` header) and returns a list of (full name, email address) pairs (even if there was only one address in the header). If there is no header matching name, return an empty list.

XXX The current version of this function is not really correct. It yields bogus results if a full name contains a comma.

function of module rfc822: **getdate** *(name)*

Retrieve a header using `getheader` and parse it into a 9-tuple compatible with `time.mktime()`. If there is no header matching name, or it is unparsable, return `None`.

Date parsing appears to be a black art, and not all mailers adhere to the standard. While it has been tested and found correct on a large collection of email from many sources, it is still possible that this function may occasionally yield an incorrect result.

`Message` instances also support a read-only mapping interface. In particular: `m[name]` is the same as `m.getheader(name)`; and `len(m)`, `m.has_key(name)`, `m.keys()`, `m.values()` and `m.items()` act as expected (and consistently).

Finally, `Message` instances have two public instance variables:

data: module rfc822 **headers**

A list containing the entire set of header lines, in the order in which they were read. Each line contains a trailing newline. The blank line terminating the headers is not contained in the list.

data: module rfc822 **fp**

The file object passed at instantiation time.

Standard Module `mimetools`

This module defines a subclass of the class `rfc822.Message` and a number of utility functions that are useful for the manipulation for MIME style multipart or encoded message.

It defines the following items:

function of module mimetools: **Message** *(fp)*

Return a new instance of the `mimetools.Message` class. This is a subclass of the `rfc822.Message` class, with some additional methods (see below).

function of module mimetools: **choose_boundary** *()*

Return a unique string that has a high likelihood of being usable as a part boundary. The string has the form `"hostipaddr.uid.pid.timestamp.random"`.

function of module mimetools: **decode** *(input, output, encoding)*

Read data encoded using the allowed MIME encoding from open file object input and write the decoded data to open file object output. Valid values for encoding include `"base64"`, `"quoted-printable"` and `"uuencode"`.

function of module mimetools: **encode** *(input, output, encoding)*

Read data from open file object input and write it encoded using the allowed MIME encoding to open file object output. Valid values for encoding are the same as for `decode()`.

function of module mimetools: **copyliteral** *(input, output)*

Read lines until EOF from open file input and write them to open file output.

function of module mimetools: **copybinary** *(input, output)*

Read blocks until EOF from open file input and write them to open file output. The block size is currently fixed at 8192.

Additional Methods of Message objects

The `mimetools.Message` class defines the following methods in addition to the `rfc822.Message` class:

Method: mimetool.Message **getplist** *()*

Return the parameter list of the `Content-type` header. This is a list if strings. For parameters of the form `key=value', key is converted to lower case but value is not. For example, if the message contains the header `Content-type: text/html; spam=1; Spam=2; Spam' then `getplist()` will return the Python list `['spam=1', 'spam=2', 'Spam']`.

Method: mimetool.Message **getparam** *(name)*

Return the value of the first parameter (as returned by `getplist()` of the form `name=value' for the given name. If value is surrounded by quotes of the form <...> or "...", these are removed.

Method: mimetool.Message **getencoding** *()*

Return the encoding specified in the `Content-transfer-encoding' message header. If no such header exists, return `"7bit"`. The encoding is converted to lower case.

Method: mimetool.Message **gettype** *()*

Return the message type (of the form `type/var{subtype'}) as specified in the `Content-type' header. If no such header exists, return `"text/plain"`. The type is converted to lower case.

Method: mimetool.Message **getmaintype** *()*

Return the main type as specified in the `Content-type' header. If no such header exists, return `"text"`. The main type is converted to lower case.

Method: mimetool.Message **getsubtype** *()*

Return the subtype as specified in the `Content-type' header. If no such header exists, return `"plain"`. The subtype is converted to lower case.

Go to the previous, next section.

Go to the previous, next section.

Multimedia Services

The modules described in this chapter implement various algorithms or interfaces that are mainly useful for multimedia applications. They are available at the discretion of the installation. Here's an overview:

audioop
--- Manipulate raw audio data.

imageop
--- Manipulate raw image data.

aifc
--- Read and write audio files in AIFF or AIFC format.

jpeg
--- Read and write image files in compressed JPEG format.

rgbimg
--- Read and write image files in "SGI RGB" format (the module is *not* SGI specific though)!

Built-in Module `audioop`

The `audioop` module contains some useful operations on sound fragments. It operates on sound fragments consisting of signed integer samples 8, 16 or 32 bits wide, stored in Python strings. This is the same format as used by the `al` and `sunaudiodev` modules. All scalar items are integers, unless specified otherwise.

A few of the more complicated operations only take 16-bit samples, otherwise the sample size (in bytes) is always a parameter of the operation.

The module defines the following variables and functions:

exception: module audioop **error**

This exception is raised on all errors, such as unknown number of bytes per sample, etc.

function of module audioop: **add** *(fragment1, fragment2, width)*

Return a fragment which is the addition of the two samples passed as parameters. width is the sample width in bytes, either `1`, `2` or `4`. Both fragments should have the same length.

function of module audioop: **adpcm2lin** *(adpcmfragment, width, state)*

Decode an Intel/DVI ADPCM coded fragment to a linear fragment. See the description of `lin2adpcm` for details on ADPCM coding. Return a tuple `(sample, newstate)` where the sample has the width specified in width.

function of module audioop: **adpcm32lin** *(adpcmfragment, width, state)*

Decode an alternative 3-bit ADPCM code. See `lin2adpcm3` for details.

function of module audioop: **avg** *(fragment, width)*

Return the average over all samples in the fragment.

function of module audioop: **avgpp** *(fragment, width)*

Return the average peak-peak value over all samples in the fragment. No filtering is done, so the usefulness of this routine is questionable.

function of module audioop: **bias** *(fragment, width, bias)*

Return a fragment that is the original fragment with a bias added to each sample.

function of module audioop: **cross** *(fragment, width)*

Return the number of zero crossings in the fragment passed as an argument.

function of module audioop: **findfactor** *(fragment, reference)*

Return a factor F such that `rms(add(fragment, mul(reference, -F)))` is minimal, i.e., return the factor with which you should multiply reference to make it match as well as possible to fragment. The fragments should both contain 2-byte samples.

The time taken by this routine is proportional to `len(fragment)`.

function of module audioop: **findfit** *(fragment, reference)*

This routine (which only accepts 2-byte sample fragments)

Try to match reference as well as possible to a portion of fragment (which should be the longer fragment). This is (conceptually) done by taking slices out of fragment, using `findfactor` to compute the best match, and minimizing the result. The fragments should both contain 2-byte samples. Return a tuple `(offset, factor)` where offset is the (integer) offset into fragment where the optimal match started and factor is the (floating-point) factor as per `findfactor`.

function of module audioop: **findmax** *(fragment, length)*

Search fragment for a slice of length length samples (not bytes!) with maximum energy, i.e., return i for which `rms(fragment[i*2:(i+length)*2])` is maximal. The fragments should both contain 2-byte samples.

The routine takes time proportional to `len(fragment)`.

function of module audioop: **getsample** *(fragment, width, index)*

Return the value of sample index from the fragment.

function of module audioop: **lin2lin** *(fragment, width, newwidth)*

Convert samples between 1-, 2- and 4-byte formats.

function of module audioop: **lin2adpcm** *(fragment, width, state)*

Convert samples to 4 bit Intel/DVI ADPCM encoding. ADPCM coding is an adaptive coding scheme, whereby each 4 bit number is the difference between one sample and the next, divided by a (varying) s The Intel/DVI ADPCM algorithm has been selected for use by the IMA, so it may well become a standard.

`State` is a tuple containing the state of the coder. The coder returns a tuple `(adpcmfrag, newstate)`, and the newstate should be passed to the next call of lin2adpcm. In the initial call `None` can be passed as the state. adpcmfrag is the ADPCM coded fragment packed 2 4-bit values per byte.

function of module audioop: **lin2adpcm3** *(fragment, width, state)*

This is an alternative ADPCM coder that uses only 3 bits per sample. It is not compatible with the Intel/DVI ADPCM coder and its output is not packed (due to laziness on the side of the author). Its use is discouraged.

function of module audioop: **lin2ulaw** *(fragment, width)*

Convert samples in the audio fragment to U-LAW encoding and return this as a Python string. U-LAW is an audio encoding format whereby you get a dynamic range of about 14 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

function of module audioop: **minmax** *(fragment, width)*

Return a tuple consisting of the minimum and maximum values of all samples in the sound fragment.

function of module audioop: **max** *(fragment, width)*

Return the maximum of the *absolute value* of all samples in a fragment.

function of module audioop: **maxpp** *(fragment, width)*

Return the maximum peak-peak value in the sound fragment.

function of module audioop: **mul** *(fragment, width, factor)*

Return a fragment that has all samples in the original framgent multiplied by the floating-point value factor. Overflow is silently ignored.

function of module audioop: **reverse** *(fragment, width)*

Reverse the samples in a fragment and returns the modified fragment.

function of module audioop: **rms** *(fragment, width)*

Return the root-mean-square of the fragment, i.e. the square root of the quotient of the sum of all squared sample value, divided by the sumber of samples. This is a measure of the power in an audio signal.

function of module audioop: **tomono** *(fragment, width, lfactor, rfactor)*

Convert a stereo fragment to a mono fragment. The left channel is multiplied by lfactor and the right channel by rfactor before adding the two channels to give a mono signal.

function of module audioop: **tostereo** *(fragment, width, lfactor, rfactor)*

Generate a stereo fragment from a mono fragment. Each pair of samples in the stereo fragment are

computed from the mono sample, whereby left channel samples are multiplied by lfactor and right cha samples by rfactor.

function of module audioop: **ulaw2lin** *(fragment, width)*

Convert sound fragments in ULAW encoding to linearly encoded sound fragments. ULAW encoding always uses 8 bits samples, so width refers only to the sample width of the output fragment here.

Note that operations such as `mul` or `max` make no distinction between mono and stereo fragments, i.e. all samples are treated equal. If this is a problem the stereo fragment should be split into two mono fragments first and recombined later. Here is an example of how to do that:

```
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
    lsample = audioop.mul(sample, width, lfactor)
    rsample = audioop.mul(sample, width, rfactor)
    lsample = audioop.tostereo(lsample, width, 1, 0)
    rsample = audioop.tostereo(rsample, width, 0, 1)
    return audioop.add(lsample, rsample, width)
```

If you use the ADPCM coder to build network packets and you want your protocol to be stateless (i.e. to be able to tolerate packet loss) you should not only transmit the data but also the state. Note that you should send the initial state (the one you passed to `lin2adpcm`) along to the decoder, not the final state (as returned by the coder). If you want to use `struct` to store the state in binary you can code the first element (the predicted value) in 16 bits and the second (the delta index) in 8.

The ADPCM coders have never been tried against other ADPCM coders, only against themselves. It could well be that I misinterpreted the standards in which case they will not be interoperable with the respective standards.

The `find...` routines might look a bit funny at first sight. They are primarily meant to do echo cancellation. A reasonably fast way to do this is to pick the most energetic piece of the output sample, locate that in the input sample and subtract the whole output sample from the input sample:

```
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)    # one tenth second
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    # Optional (for better cancellation):
    # factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)],
    #             out_test)
    prefill = '\0'*(pos+ipos)*2
    postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
    outputdata = prefill + audioop.mul(outputdata,2,-factor) + postfill
    return audioop.add(inputdata, outputdata, 2)
```

Built-in Module `imageop`

The `imageop` module contains some useful operations on images. It operates on images consisting of 8 or 32 bit pixels stored in Python strings. This is the same format as used by `gl.lrectwrite` and the `imgfile` module.

The module defines the following variables and functions:

exception: module imageop **error**

This exception is raised on all errors, such as unknown number of bits per pixel, etc.

function of module imageop: **crop** *(image, psize, width, height, x0, y0, x1, y1)*

Return the selected part of image, which should by width by height in size and consist of pixels of psize bytes. x0, y0, x1 and y1 are like the `lrectread` parameters, i.e. the boundary is included in the new image. The new boundaries need not be inside the picture. Pixels that fall outside the old image will have their value set to zero. If x0 is bigger than x1 the new image is mirrored. The same holds for the y coordinates.

function of module imageop: **scale** *(image, psize, width, height, newwidth, newheight)*

Return image scaled to size newwidth by newheight. No interpolation is done, scaling is done by simple-minded pixel duplication or removal. Therefore, computer-generated images or dithered images will not look nice after scaling.

function of module imageop: **tovideo** *(image, psize, width, height)*

Run a vertical low-pass filter over an image. It does so by computing each destination pixel as the average of two vertically-aligned source pixels. The main use of this routine is to forestall excessive flicker if the image is displayed on a video device that uses interlacing, hence the name.

function of module imageop: **grey2mono** *(image, width, height, threshold)*

Convert a 8-bit deep greyscale image to a 1-bit deep image by tresholding all the pixels. The resulting image is tightly packed and is probably only useful as an argument to `mono2grey`.

function of module imageop: **dither2mono** *(image, width, height)*

Convert an 8-bit greyscale image to a 1-bit monochrome image using a (simple-minded) dithering algorithm.

function of module imageop: **mono2grey** *(image, width, height, p0, p1)*

Convert a 1-bit monochrome image to an 8 bit greyscale or color image. All pixels that are zero-valued on input get value p0 on output and all one-value input pixels get value p1 on output. To convert a monochrome black-and-white image to greyscale pass the values `0` and `255` respectively.

function of module imageop: **grey2grey4** *(image, width, height)*

Convert an 8-bit greyscale image to a 4-bit greyscale image without dithering.

function of module imageop: **grey2grey2** *(image, width, height)*

Convert an 8-bit greyscale image to a 2-bit greyscale image without dithering.

function of module imageop: **dither2grey2** *(image, width, height)*

Convert an 8-bit greyscale image to a 2-bit greyscale image with dithering. As for `dither2mono`, the dithering algorithm is currently very simple.

function of module imageop: **grey42grey** *(image, width, height)*

Convert a 4-bit greyscale image to an 8-bit greyscale image.

function of module imageop: **grey22grey** *(image, width, height)*

Convert a 2-bit greyscale image to an 8-bit greyscale image.

Standard Module `aifc`

This module provides support for reading and writing AIFF and AIFF-C files. AIFF is Audio Interchange File Format, a format for storing digital audio samples in a file. AIFF-C is a newer version of the format that includes the ability to compress the audio data.

Audio files have a number of parameters that describe the audio data. The sampling rate or frame rate is the number of times per second the sound is sampled. The number of channels indicate if the audio is mono, stereo, or quadro. Each frame consists of one sample per channel. The sample size is the size in bytes of each sample. Thus a frame consists of nchannels*samplesize bytes, and a second's worth of audio consists of nchannels*samplesize*framerate bytes.

For example, CD quality audio has a sample size of two bytes (16 bits), uses two channels (stereo) and has a frame rate of 44,100 frames/second. This gives a frame size of 4 bytes (2*2), and a second's worth occupies 2*2*44100 bytes, i.e. 176,400 bytes.

Module `aifc` defines the following function:

function of module aifc: **open** *(file, mode)*

Open an AIFF or AIFF-C file and return an object instance with methods that are described below. The argument file is either a string naming a file or a file object. The mode is either the string `'r'` when the file must be opened for reading, or `'w'` when the file must be opened for writing. When used for writing, the file object should be seekable, unless you know ahead of time how many samples you are going to write in total and use `writeframesraw()` and `setnframes()`.

Objects returned by `aifc.open()` when a file is opened for reading have the following methods:

Method: aifc object **getnchannels** *()*

Return the number of audio channels (1 for mono, 2 for stereo).

Method: aifc object **getsampwidth** *()*

Return the size in bytes of individual samples.

Method: aifc object **getframerate** *()*

Return the sampling rate (number of audio frames per second).

Method: aifc object **getnframes** *()*

Return the number of audio frames in the file.

Method: aifc object **getcomptype** *()*

Return a four-character string describing the type of compression used in the audio file. For AIFF files, the returned value is `'NONE'`.

Method: aifc object **getcompname** *()*

Return a human-readable description of the type of compression used in the audio file. For AIFF files, the returned value is `'not compressed'`.

Method: aifc object **getparams** *()*

Return a tuple consisting of all of the above values in the above order.

Method: aifc object **getmarkers** *()*

Return a list of markers in the audio file. A marker consists of a tuple of three elements. The first is the mark ID (an integer), the second is the mark position in frames from the beginning of the data (an integer), the third is the name of the mark (a string).

Method: aifc object **getmark** *(id)*

Return the tuple as described in `getmarkers` for the mark with the given id.

Method: aifc object **readframes** *(nframes)*

Read and return the next nframes frames from the audio file. The returned data is a string containing for each frame the uncompressed samples of all channels.

Method: aifc object **rewind** *()*

Rewind the read pointer. The next `readframes` will start from the beginning.

Method: aifc object **setpos** *(pos)*

Seek to the specified frame number.

Method: aifc object **tell** *()*

Return the current frame number.

Method: aifc object **close** *()*

Close the AIFF file. After calling this method, the object can no longer be used.

Objects returned by `aifc.open()` when a file is opened for writing have all the above methods, except for `readframes` and `setpos`. In addition the following methods exist. The `get` methods can only be

called after the corresponding `set` methods have been called. Before the first `writeframes` or `writeframesraw`, all parameters except for the number of frames must be filled in.

Method: aifc object **aiff** *()*

Create an AIFF file. The default is that an AIFF-C file is created, unless the name of the file ends in '.aiff' in which case the default is an AIFF file.

Method: aifc object **aifc** *()*

Create an AIFF-C file. The default is that an AIFF-C file is created, unless the name of the file ends in '.aiff' in which case the default is an AIFF file.

Method: aifc object **setnchannels** *(nchannels)*

Specify the number of channels in the audio file.

Method: aifc object **setsampwidth** *(width)*

Specify the size in bytes of audio samples.

Method: aifc object **setframerate** *(rate)*

Specify the sampling frequency in frames per second.

Method: aifc object **setnframes** *(nframes)*

Specify the number of frames that are to be written to the audio file. If this parameter is not set, or not set correctly, the file needs to support seeking.

Method: aifc object **setcomptype** *(type, name)*

Specify the compression type. If not specified, the audio data will not be compressed. In AIFF files, compression is not possible. The name parameter should be a human-readable description of the compression type, the type parameter should be a four-character string. Currently the following compression types are supported: NONE, ULAW, ALAW, G722.

Method: aifc object **setparams** *(nchannels, sampwidth, framerate, comptype, compname)*

Set all the above parameters at once. The argument is a tuple consisting of the various parameters. This means that it is possible to use the result of a `getparams` call as argument to `setparams`.

Method: aifc object **setmark** *(id, pos, name)*

Add a mark with the given id (larger than 0), and the given name at the given position. This method can be called at any time before `close`.

Method: aifc object **tell** *()*

Return the current write position in the output file. Useful in combination with `setmark`.

Method: aifc object **writeframes** *(data)*

Write data to the output file. This method can only be called after the audio file parameters have been set.

Method: aifc object **writeframesraw** *(data)*

Like `writeframes`, except that the header of the audio file is not updated.

Method: aifc object **close** *()*

Close the AIFF file. The header of the file is updated to reflect the actual size of the audio data. After calling this method, the object can no longer be used.

Built-in Module `jpeg`

The module `jpeg` provides access to the jpeg compressor and decompressor written by the Independent JPEG Group. JPEG is a (draft?) standard for compressing pictures. For details on jpeg or the Independent JPEG Group software refer to the JPEG standard or the documentation provided with the software.

The `jpeg` module defines these functions:

function of module jpeg: **compress** *(data, w, h, b)*

Treat data as a pixmap of width w and height h, with b bytes per pixel. The data is in SGI GL order, so the first pixel is in the lower-left corner. This means that `lrectread` return data can immediately be passed to compress. Currently only 1 byte and 4 byte pixels are allowed, the former being treated as greyscale and the latter as RGB color. Compress returns a string that contains the compressed picture, in JFIF format.

function of module jpeg: **decompress** *(data)*

Data is a string containing a picture in JFIF format. It returns a tuple `(data, width, height, bytesperpixel)`. Again, the data is suitable to pass to `lrectwrite`.

function of module jpeg: **setoption** *(name, value)*

Set various options. Subsequent compress and decompress calls will use these options. The following options are available:

`'forcegray'`

Force output to be grayscale, even if input is RGB.

`'quality'`

Set the quality of the compressed image to a value between `0` and `100` (default is `75`). Compress only.

`'optimize'`

Perform Huffman table optimization. Takes longer, but results in smaller compressed image. Compress only.

`'smooth'`

Perform inter-block smoothing on uncompressed image. Only useful for low-quality images. Decompress only.

Compress and uncompress raise the error `jpeg.error` in case of errors.

Built-in Module `rgbimg`

The rgbimg module allows python programs to access SGI imglib image files (also known as `.rgb' files). The module is far from complete, but is provided anyway since the functionality that there is is enough in some cases. Currently, colormap files are not supported.

The module defines the following variables and functions:

exception: module rgbimg **error**

This exception is raised on all errors, such as unsupported file type, etc.

function of module rgbimg: **sizeofimage** *(file)*

This function returns a tuple `(x, y)` where x and y are the size of the image in pixels. Only 4 byte RGBA pixels, 3 byte RGB pixels, and 1 byte greyscale pixels are currently supported.

function of module rgbimg: **longimagedata** *(file)*

This function reads and decodes the image on the specified file, and returns it as a Python string. The string has 4 byte RGBA pixels. The bottom left pixel is the first in the string. This format is suitable to pass to `gl.lrectwrite`, for instance.

function of module rgbimg: **longstoimage** *(data, x, y, z, file)*

This function writes the RGBA data in data to image file file. x and y give the size of the image. z is 1 if the saved image should be 1 byte greyscale, 3 if the saved image should be 3 byte RGB data, or 4 if the saved images should be 4 byte RGBA data. The input data always contains 4 bytes per pixel. These are the formats returned by `gl.lrectread`.

function of module rgbimg: **ttob** *(flag)*

This function sets a global flag which defines whether the scan lines of the image are read or written from bottom to top (flag is zero, compatible with SGI GL) or from top to bottom(flag is one, compatible with X).

Go to the previous, next section.

Go to the previous, next section.

Cryptographic Services

The modules described in this chapter implement various algorithms of a cryptographic nature. They are available at the discretion of the installation. Here's an overview:

md5

--- RSA's MD5 message digest algorithm.

mpz

--- Interface to the GNU MP library for arbitrary precision arithmetic.

rotor

--- Enigma-like encryption and decryption.

Hardcore cypherpunks will probably find the Python Cryptography Kit of further interest; the package adds built-in modules for DES and IDEA encryption, and provides a Python module for reading and decrypting PGP files. The Python Cryptography Kit is not distributed with Python but available separately. See the URL `http://www.cs.mcgill.ca/%7Efnord/crypt.html' for more information.

Built-in Module md5

This module implements the interface to RSA's MD5 message digest algorithm (see also Internet RFC 1321). Its use is quite straightforward: use the `md5.new()` to create an md5 object. You can now feed this object with arbitrary strings using the `update()` method, and at any point you can ask it for the digest (a strong kind of 128-bit checksum, a.k.a. "fingerprint") of the contatenation of the strings fed to it so far using the `digest()` method.

For example, to obtain the digest of the string `"Nobody inspects the spammish repetition"`:

```
>>> import md5
>>> m = md5.new()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\273d\234\203\335\036\245\311\331\336\311\241\215\360\377\351'
```

More condensed:

```
>>> md5.new("Nobody inspects the spammish repetition").digest()
'\273d\234\203\335\036\245\311\331\336\311\241\215\360\377\351'
```

function of module md5: **new** *([arg])*

Return a new md5 object. If arg is present, the method call `update(arg)` is made.

function of module md5: **md5** *([arg])*

For backward compatibility reasons, this is an alternative name for the `new()` function.

An md5 object has the following methods:

Method: md5 **update** *(arg)*

Update the md5 object with the string arg. Repeated calls are equivalent to a single call with the concatenation of all the arguments, i.e. `m.update(a); m.update(b)` is equivalent to `m.update(a+b)`.

Method: md5 **digest** *()*

Return the digest of the strings passed to the `update()` method so far. This is an 8-byte string which may contain non-ASCII characters, including null bytes.

Method: md5 **copy** *()*

Return a copy ("clone") of the md5 object. This can be used to efficiently compute the digests of strings that share a common initial substring.

Built-in Module `mpz`

This module implements the interface to part of the GNU MP library. This library contains arbitrary precision integer and rational number arithmetic routines. Only the interfaces to the *integer* (`mpz_...') routines are provided. If not stated otherwise, the description in the GNU MP documentation can be applied.

In general, mpz-numbers can be used just like other standard Python numbers, e.g. you can use the built-in operators like `+`, `*`, etc., as well as the standard built-in functions like `abs`, `int`, ..., `divmod`, `pow`. **Please note:** the *bitwise-xor* operation has been implemented as a bunch of *and*s, *invert*s and *or*s, because the library lacks an `mpz_xor` function, and I didn't need one.

You create an mpz-number by calling the function called `mpz` (see below for an exact description). An mpz-number is printed like this: `mpz(value)`.

function of module mpz: **mpz** *(value)*

Create a new mpz-number. value can be an integer, a long, another mpz-number, or even a string. If it is a string, it is interpreted as an array of radix-256 digits, least significant digit first, resulting in a positive number. See also the `binary` method, described below.

A number of *extra* functions are defined in this module. Non mpz-arguments are converted to mpz-values first, and the functions return mpz-numbers.

function of module mpz: **powm** *(base, exponent, modulus)*

Return `pow(base, exponent) % modulus`. If `exponent == 0`, return `mpz(1)`. In contrast to

the C-library function, this version can handle negative exponents.

function of module mpz: **gcd** *(op1, op2)*

Return the greatest common divisor of op1 and op2.

function of module mpz: **gcdext** *(a, b)*

Return a tuple `(g, s, t)`, such that `a*s + b*t == g == gcd(a, b)`.

function of module mpz: **sqrt** *(op)*

Return the square root of op. The result is rounded towards zero.

function of module mpz: **sqrtrem** *(op)*

Return a tuple `(root, remainder)`, such that `root*root + remainder == op`.

function of module mpz: **divm** *(numerator, denominator, modulus)*

Returns a number q. such that `q * denominator % modulus == numerator`. One could also implement this function in Python, using `gcdext`.

An mpz-number has one method:

Method: mpz **binary** *()*

Convert this mpz-number to a binary string, where the number has been stored as an array of radix-256 digits, least significant digit first.

The mpz-number must have a value greater than or equal to zero, otherwise a `ValueError`-exception will be raised.

Built-in Module `rotor`

This module implements a rotor-based encryption algorithm, contributed by Lance Ellinghouse. The design is derived from the Enigma device, a machine used during World War II to encipher messages. A rotor is simply a permutation. For example, if the character `A' is the origin of the rotor, then a given rotor might map `A' to `L', `B' to `Z', `C' to `G', and so on. To encrypt, we choose several different rotors, and set the origins of the rotors to known positions; their initial position is the ciphering key. To encipher a character, we permute the original character by the first rotor, and then apply the second rotor's permutation to the result. We continue until we've applied all the rotors; the resulting character is our ciphertext. We then change the origin of the final rotor by one position, from `A' to `B'; if the final rotor has made a complete revolution, then we rotate the next-to-last rotor by one position, and apply the same procedure recursively. In other words, after enciphering one character, we advance the rotors in the same fashion as a car's odometer. Decoding works in the same way, except we reverse the permutations and apply them in the opposite order.

The available functions in this module are:

function of module rotor: **newrotor** *(key[, numrotors])*

Return a rotor object. key is a string containing the encryption key for the object; it can contain arbitrary binary data. The key will be used to randomly generate the rotor permutations and their initial positions. numrotors is the number of rotor permutations in the returned object; if it is omitted, a default value of 6 will be used.

Rotor objects have the following methods:

Method: rotor **setkey** *()*

Reset the rotor to its initial state.

Method: rotor **encrypt** *(plaintext)*

Reset the rotor object to its initial state and encrypt plaintext, returning a string containing the ciphertext. The ciphertext is always the same length as the original plaintext.

Method: rotor **encryptmore** *(plaintext)*

Encrypt plaintext without resetting the rotor object, and return a string containing the ciphertext.

Method: rotor **decrypt** *(ciphertext)*

Reset the rotor object to its initial state and decrypt ciphertext, returning a string containing the ciphertext. The plaintext string will always be the same length as the ciphertext.

Method: rotor **decryptmore** *(ciphertext)*

Decrypt ciphertext without resetting the rotor object, and return a string containing the ciphertext.

An example usage:

```
>>> import rotor
>>> rt = rotor.newrotor('key', 12)
>>> rt.encrypt('bar')
'\2534\363'
>>> rt.encryptmore('bar')
'\357\375$'
>>> rt.encrypt('bar')
'\2534\363'
>>> rt.decrypt('\2534\363')
'bar'
>>> rt.decryptmore('\357\375$')
'bar'
>>> rt.decrypt('\357\375$')
'l(\315'
>>> del rt
```

The module's code is not an exact simulation of the original Enigma device; it implements the rotor encryption scheme differently from the original. The most important difference is that in the original Enigma, there were only 5 or 6 different rotors in existence, and they were applied twice to each

character; the cipher key was the order in which they were placed in the machine. The Python rotor module uses the supplied key to initialize a random number generator; the rotor permutations and the initial positions are then randomly generated. The original device only enciphered the letters of the alphabet, while this module can handle any 8-bit binary data; it also produces binary output. This module can also operate with an arbitrary number of rotors.

The original Enigma cipher was broken in 1944. The version implemented here is probably a good deal more difficult to crack (especially if you use many rotors), but it won't be impossible for a truly skilful and determined attacker to break the cipher. So if you want to keep the NSA out of your files, this rotor cipher may well be unsafe, but for discouraging casual snooping through your files, it will probably be just fine, and may be somewhat safer than using the Unix `crypt' command.

Go to the previous, next section.

Go to the previous, next section.

Macintosh Specific Services

The modules in this chapter are available on the Apple Macintosh only.

Built-in Module `mac`

This module provides a subset of the operating system dependent functionality provided by the optional built-in module `posix`. It is best accessed through the more portable standard module `os`.

The following functions are available in this module: `chdir`, `getcwd`, `listdir`, `mkdir`, `rename`, `rmdir`, `stat`, `sync`, `unlink`, as well as the exception `error`.

Standard Module `macpath`

This module provides a subset of the pathname manipulation functions available from the optional standard module `posixpath`. It is best accessed through the more portable standard module `os`, as `os.path`.

The following functions are available in this module: `normcase`, `isabs`, `join`, `split`, `isdir`, `isfile`, `exists`.

Built-in Module `ctb`

This module provides a partial interface to the Macintosh Communications Toolbox. Currently, only Connection Manager tools are supported. It may not be available in all Mac Python versions.

data: module ctb **error**

The exception raised on errors.

data: module ctb **cmData**

data: module ctb **cmCntl**

data: module ctb **cmAttn**

Flags for the channel argument of the Read and Write methods.

data: module ctb **cmFlagsEOM**

End-of-message flag for Read and Write.

data: module ctb **choose***

Values returned by Choose.

data: module ctb **cmStatus***

Bits in the status as returned by Status.

function of module ctb: **available** *()*

Return 1 if the communication toolbox is available, zero otherwise.

function of module ctb: **CMNew** *(name, sizes)*

Create a connection object using the connection tool named name. sizes is a 6-tuple given buffer sizes for data in, data out, control in, control out, attention in and attention out. Alternatively, passing `None` will result in default buffer sizes.

connection object

For all connection methods that take a timeout argument, a value of `-1` is indefinite, meaning that the command runs to completion.

attribute: connection object **callback**

If this member is set to a value other than `None` it should point to a function accepting a single argument (the connection object). This will make all connection object methods work asynchronously, with the callback routine being called upon completion.

Note: for reasons beyond my understanding the callback routine is currently never called. You are advised against using asynchronous calls for the time being.

Method: connection object **Open** *(timeout)*

Open an outgoing connection, waiting at most timeout seconds for the connection to be established.

Method: connection object **Listen** *(timeout)*

Wait for an incoming connection. Stop waiting after timeout seconds. This call is only meaningful to some tools.

Method: connection object **accept** *(yesno)*

Accept (when yesno is non-zero) or reject an incoming call after Listen returned.

Method: connection object **Close** *(timeout, now)*

Close a connection. When now is zero, the close is orderly (i.e. outstanding output is flushed, etc.) with a timeout of timeout seconds. When now is non-zero the close is immediate, discarding output.

Method: connection object **Read** *(len, chan, timeout)*

Read len bytes, or until timeout seconds have passed, from the channel chan (which is one of cmData, cmCntl or cmAttn). Return a 2-tuple: the data read and the end-of-message flag.

Method: connection object **Write** *(buf, chan, timeout, eom)*

Write buf to channel chan, aborting after timeout seconds. When eom has the value cmFlagsEOM an end-of-message indicator will be written after the data (if this concept has a meaning for this communication tool). The method returns the number of bytes written.

Method: connection object **Status** *()*

Return connection status as the 2-tuple `(sizes, flags)`. sizes is a 6-tuple giving the actual buffer sizes used (see CMNew), flags is a set of bits describing the state of the connection.

Method: connection object **GetConfig** *()*

Return the configuration string of the communication tool. These configuration strings are tool-dependent, but usually easily parsed and modified.

Method: connection object **SetConfig** *(str)*

Set the configuration string for the tool. The strings are parsed left-to-right, with later values taking precedence. This means individual configuration parameters can be modified by simply appending something like `'baud 4800'` to the end of the string returned by GetConfig and passing that to this method. The method returns the number of characters actually parsed by the tool before it encountered an error (or completed successfully).

Method: connection object **Choose** *()*

Present the user with a dialog to choose a communication tool and configure it. If there is an outstanding connection some choices (like selecting a different tool) may cause the connection to be aborted. The return value (one of the choose* constants) will indicate this.

Method: connection object **Idle** *()*

Give the tool a chance to use the processor. You should call this method regularly.

Method: connection object **Abort** *()*

Abort an outstanding asynchronous Open or Listen.

Method: connection object **Reset** *()*

Reset a connection. Exact meaning depends on the tool.

Method: connection object **Break** *(length)*

Send a break. Whether this means anything, what it means and interpretation of the length parameter depend on the tool in use.

TOC

Built-in Module macconsole

This module is available on the Macintosh, provided Python has been built using the Think C compiler. It provides an interface to the Think console package, with which basic text windows can be created.

data: module macconsole **options**

An object allowing you to set various options when creating windows, see below.

data: module macconsole **C_ECHO**

data: module macconsole **C_NOECHO**

data: module macconsole **C_CBREAK**

data: module macconsole **C_RAW**

Options for the `setmode` method. C_ECHO and C_CBREAK enable character echo, the other two disable it, C_ECHO and C_NOECHO enable line-oriented input (erase/kill processing, etc).

function of module macconsole: **copen** *()*

Open a new console window. Return a console window object.

function of module macconsole: **fopen** *(fp)*

Return the console window object corresponding with the given file object. fp should be one of `sys.stdin`, `sys.stdout` or `sys.stderr`.

macconsole options object

These options are examined when a window is created:

option: macconsole **top**

option: macconsole **left**

The origin of the window.

option: macconsole **nrows**

option: macconsole **ncols**

The size of the window.

option: macconsole **txFont**

option: macconsole **txSize**

option: macconsole **txStyle**

The font, fontsize and fontstyle to be used in the window.

option: macconsole **title**

The title of the window.

option: macconsole **pause_atexit**

If set non-zero, the window will wait for user action before closing.

console window object

attribute: console window **file**

The file object corresponding to this console window. If the file is buffered, you should call `file.flush()` between `write()` and `read()` calls.

Method: console window **setmode** *(mode)*

Set the input mode of the console to C_ECHO, etc.

Method: console window **settabs** *(n)*

Set the tabsize to n spaces.

Method: console window **cleos** *()*

Clear to end-of-screen.

Method: console window **cleol** *()*

Clear to end-of-line.

Method: console window **inverse** *(onoff)*

Enable inverse-video mode: characters with the high bit set are displayed in inverse video (this disables the upper half of a non-ASCII character set).

Method: console window **gotoxy** *(x, y)*

Set the cursor to position `(x, y)`.

Method: console window **hide** *()*

Hide the window, remembering the contents.

Method: console window **show** *()*

Show the window again.

Method: console window **echo2printer** *()*

Copy everything written to the window to the printer as well.

Built-in Module macdnr

This module provides an interface to the Macintosh Domain Name Resolver. It is usually used in conjunction with the mactcp module, to map hostnames to IP-addresses. It may not be available in all Mac Python versions.

The `macdnr` module defines the following functions:

function of module macdnr: **Open** *([filename])*

Open the domain name resolver extension. If filename is given it should be the pathname of the extension, otherwise a default is used. Normally, this call is not needed since the other calls will open the extension automatically.

function of module macdnr: **Close** *()*

Close the resolver extension. Again, not needed for normal use.

function of module macdnr: **StrToAddr** *(hostname)*

Look up the IP address for hostname. This call returns a dnr result object of the "address" variation.

function of module macdnr: **AddrToName** *(addr)*

Do a reverse lookup on the 32-bit integer IP-address addr. Returns a dnr result object of the "address" variation.

function of module macdnr: **AddrToStr** *(addr)*

Convert the 32-bit integer IP-address addr to a dotted-decimal string. Returns the string.

function of module macdnr: **HInfo** *(hostname)*

Query the nameservers for a `HInfo` record for host hostname. These records contain hardware and software information about the machine in question (if they are available in the first place). Returns a dnr result object of the "hinfo" variety.

function of module macdnr: **MXInfo** *(domain)*

Query the nameservers for a mail exchanger for domain. This is the hostname of a host willing to accept SMTP mail for the given domain. Returns a dnr result object of the "mx" variety.

dnr result object

Since the DNR calls all execute asynchronously you do not get the results back immediately. Instead, you get a dnr result object. You can check this object to see whether the query is complete, and access its attributes to obtain the information when it is.

Alternatively, you can also reference the result attributes directly, this will result in an implicit wait for the query to complete.

The rtnCode and cname attributes are always available, the others depend on the type of query (addr, hinfo or mx).

Method: dnr result object **wait** *()*

Wait for the query to complete.

Method: dnr result object **isdone** *()*

Return 1 if the query is complete.

attribute: dnr result object **rtnCode**

The error code returned by the query.

attribute: dnr result object **cname**

The canonical name of the host that was queried.

attribute: dnr result object **ip0**

attribute: dnr result object **ip1**

attribute: dnr result object **ip2**

attribute: dnr result object **ip3**

At most four integer IP addresses for this host. Unused entries are zero. Valid only for address queries.

attribute: dnr result object **cpuType**

attribute: dnr result object **osType**

Textual strings giving the machine type an OS name. Valid for hinfo queries.

attribute: dnr result object **exchange**

The name of a mail-exchanger host. Valid for mx queries.

attribute: dnr result object **preference**

The preference of this mx record. Not too useful, since the Macintosh will only return a single mx record. Mx queries only.

The simplest way to use the module to convert names to dotted-decimal strings, without worrying about idle time, etc:

```
>>> def gethostname(name):
...     import macdnr
...     dnrr = macdnr.StrToAddr(name)
...     return macdnr.AddrToStr(dnrr.ip0)
```

Built-in Module `macfs`

This module provides access to macintosh FSSpec handling, the Alias Manager, finder aliases and the Standard File package.

Whenever a function or method expects a file argument, this argument can be one of three things: (1) a full or partial Macintosh pathname, (2) an FSSpec object or (3) a 3-tuple `(wdRefNum, parID, name)` as described in Inside Mac VI. and the standard file package can also be found there.

function of module macfs: **FSSpec** *(file)*

Create an FSSpec object for the specified file.

function of module macfs: **RawFSSpec** *(data)*

Create an FSSpec object given the raw data for the C structure for the FSSpec as a string. This is mainly useful if you have obtained an FSSpec structure over a network.

function of module macfs: **RawAlias** *(data)*

Create an Alias object given the raw data for the C structure for the alias as a string. This is mainly useful if you have obtained an FSSpec structure over a network.

function of module macfs: **ResolveAliasFile** *(file)*

Resolve an alias file. Returns a 3-tuple `(fsspec, isfolder, aliased)` where fsspec is the resulting FSSpec object, isfolder is true if fsspec points to a folder and aliased is true if the file was an alias in the first place (otherwise the FSSpec object for the file itself is returned).

function of module macfs: **StandardGetFile** *([type, ...])*

Present the user with a standard "open input file" dialog. Optionally, you can pass up to four 4-char file types to limit the files the user can choose from. The function returns an FSSpec object and a flag indicating that the user completed the dialog without cancelling.

function of module macfs: **StandardPutFile** *(prompt, [default])*

Present the user with a standard "open output file" dialog. prompt is the prompt string, and the optional default argument initializes the output file name. The function returns an FSSpec object and a flag indicating that the user completed the dialog without cancelling.

function of module macfs: **GetDirectory** *()*

Present the user with a non-standard "select a directory" dialog. Return an FSSpec object and a success-indicator.

FSSpec objects

attribute: FSSpec object **data**

The raw data from the FSSpec object, suitable for passing to other applications, for instance.

Method: FSSpec object **as_pathname** *()*

Return the full pathname of the file described by the FSSpec object.

Method: FSSpec object **as_tuple** *()*

Return the `(wdRefNum, parID, name)` tuple of the file described by the FSSpec object.

Method: FSSpec object **NewAlias** *([file])*

Create an Alias object pointing to the file described by this FSSpec. If the optional file parameter is present the alias will be relative to that file, otherwise it will be absolute.

Method: FSSpec object **NewAliasMinimal** *()*

Create a minimal alias pointing to this file.

Method: FSSpec object **GetCreatorType** *()*

Return the 4-char creator and type of the file.

Method: FSSpec object **SetCreatorType** *(creator, type)*

Set the 4-char creator and type of the file.

alias objects

attribute: alias object **data**

The raw data for the Alias record, suitable for storing in a resource or transmitting to other programs.

Method: alias object **Resolve** *([file])*

Resolve the alias. If the alias was created as a relative alias you should pass the file relative to which it is. Return the FSSpec for the file pointed to and a flag indicating whether the alias object itself was modified during the search process.

Method: alias object **GetInfo** *(num)*

An interface to the C routine `GetAliasInfo()`.

Method: alias object **Update** *(file, [file2])*

Update the alias to point to the file given. If file2 is present a relative alias will be created.

Note that it is currently not possible to directly manipulate a resource as an alias object. Hence, after calling Update or after Resolve indicates that the alias has changed the Python program is responsible for

getting the data from the alias object and modifying the resource.

Built-in Module `mactcp`

This module provides an interface to the Macintosh TCP/IP driver MacTCP.`macdnr` which provides an interface to the name-server (allowing you to translate hostnames to ip-addresses), a module `MACTCP` which has symbolic names for constants constants used by MacTCP and a wrapper module `socket` which mimics the UNIX socket interface (as far as possible). It may not be available in all Mac Python versions.

A complete description of the MacTCP interface can be found in the Apple MacTCP API documentation.

function of module mactcp: **MTU** *()*

Return the Maximum Transmit Unit (the packet size) of the network interface.

function of module mactcp: **IPAddr** *()*

Return the 32-bit integer IP address of the network interface.

function of module mactcp: **NetMask** *()*

Return the 32-bit integer network mask of the interface.

function of module mactcp: **TCPCreate** *(size)*

Create a TCP Stream object. size is the size of the receive buffer, `4096` is suggested by various sources.

function of module mactcp: **UDPCreate** *(size, port)*

Create a UDP stream object. size is the size of the receive buffer (and, hence, the size of the biggest datagram you can receive on this port). port is the UDP port number you want to receive datagrams on, a value of zero will make MacTCP select a free port.

TCP Stream Objects

attribute: TCP stream **asr**

When set to a value different than `None` this should point to a function with two integer parameters: an event code and a detail. This function will be called upon network-generated events such as urgent data arrival. In addition, it is called with eventcode `MACTCP.PassiveOpenDone` when a `PassiveOpen` completes. This is a Python addition to the MacTCP semantics. It is safe to do further calls from the `asr`.

Method: TCP stream **PassiveOpen** *(port)*

Wait for an incoming connection on TCP port port (zero makes the system pick a free port). The call returns immediately, and you should use wait to wait for completion. You should not issue any method calls other than `wait`, `isdone` or `GetSockName` before the call completes.

Method: TCP stream **wait** *()*

Wait for `PassiveOpen` to complete.

Method: TCP stream **isdone** *()*

Return 1 if a `PassiveOpen` has completed.

Method: TCP stream **GetSockName** *()*

Return the TCP address of this side of a connection as a 2-tuple `(host, port)`, both integers.

Method: TCP stream **ActiveOpen** *(lport, host, rport)*

Open an outgoing connection to TCP address `(host, rport)`. Use local port lport (zero makes the system pick a free port). This call blocks until the connection has been established.

Method: TCP stream **Send** *(buf, push, urgent)*

Send data buf over the connection. Push and urgent are flags as specified by the TCP standard.

Method: TCP stream **Rcv** *(timeout)*

Receive data. The call returns when timeout seconds have passed or when (according to the MacTCP documentation) "a reasonable amount of data has been received". The return value is a 3-tuple `(data, urgent, mark)`. If urgent data is outstanding `Rcv` will always return that before looking at any normal data. The first call returning urgent data will have the urgent flag set, the last will have the mark flag set.

Method: TCP stream **Close** *()*

Tell MacTCP that no more data will be transmitted on this connection. The call returns when all data has been acknowledged by the receiving side.

Method: TCP stream **Abort** *()*

Forcibly close both sides of a connection, ignoring outstanding data.

Method: TCP stream **Status** *()*

Return a TCP status object for this stream giving the current status (see below).

TCP Status Objects

This object has no methods, only some members holding information on the connection. A complete description of all fields in this objects can be found in the Apple documentation. The most interesting ones are:

attribute: TCP status **localHost**

attribute: TCP status **localPort**

attribute: TCP status **remoteHost**

attribute: TCP status **remotePort**

The integer IP-addresses and port numbers of both endpoints of the connection.

attribute: TCP status **sendWindow**

The current window size.

attribute: TCP status **amtUnackedData**

The number of bytes sent but not yet acknowledged. `sendWindow - amtUnackedData` is what you can pass to `Send` without blocking.

attribute: TCP status **amtUnreadData**

The number of bytes received but not yet read (what you can `Recv` without blocking).

UDP Stream Objects

Note that, unlike the name suggests, there is nothing stream-like about UDP.

attribute: UDP stream **asr**

The asynchronous service routine to be called on events such as datagram arrival without outstanding `Read` call. The `asr` has a single argument, the event code.

attribute: UDP stream **port**

A read-only member giving the port number of this UDP stream.

Method: UDP stream **Read** *(timeout)*

Read a datagram, waiting at most timeout seconds (is infinite). Return the data.

Method: UDP stream **Write** *(host, port, buf)*

Send buf as a datagram to IP-address host, port port.

Built-in Module `macspeech`

This module provides an interface to the Macintosh Speech Manager, allowing you to let the Macintosh utter phrases. You need a version of the speech manager extension (version 1 and 2 have been tested) in your `Extensions` folder for this to work. The module does not provide full access to all features of the Speech Manager yet. It may not be available in all Mac Python versions.

function of module macspeech: **Available** *()*

Test availability of the Speech Manager extension (and, on the PowerPC, the Speech Manager shared library). Return 0 or 1.

function of module macspeech: **Version** *()*

Return the (integer) version number of the Speech Manager.

function of module macspeech: **SpeakString** *(str)*

Utter the string str using the default voice, asynchronously. This aborts any speech that may still be active from prior `SpeakString` invocations.

function of module macspeech: **Busy** *()*

Return the number of speech channels busy, system-wide.

function of module macspeech: **CountVoices** *()*

Return the number of different voices available.

function of module macspeech: **GetIndVoice** *(num)*

Return a voice object for voice number num.

voice objects

Voice objects contain the description of a voice. It is currently not yet possible to access the parameters of a voice.

Method: voice object **GetGender** *()*

Return the gender of the voice: 0 for male, 1 for female and for neuter.

Method: voice object **NewChannel** *()*

Return a new speech channel object using this voice.

speech channel objects

A speech channel object allows you to speak strings with slightly more control than `SpeakString()`, and allows you to use multiple speakers at the same time. Please note that channel pitch and rate are interrelated in some way, so that to make your Macintosh sing you will have to adjust both.

Method: speech channel object **SpeakText** *(str)*

Start uttering the given string.

Method: speech channel object **Stop** *()*

Stop babbling.

Method: speech channel object **GetPitch** *()*

Return the current pitch of the channel, as a floating-point number.

Method: speech channel object **SetPitch** *(pitch)*

TOC

Set the pitch of the channel.

Method: speech channel object **GetRate** *()*

Get the speech rate (utterances per minute) of the channel as a floating point number.

Method: speech channel object **SetRate** *(rate)*

Set the speech rate of the channel.

Go to the previous, next section.

Go to the previous, next section.

Standard Windowing Interface

The modules in this chapter are available only on those systems where the STDWIN library is available. STDWIN runs on UNIX under X11 and on the Macintosh. See CWI report CS-R8817.

Warning: Using STDWIN is not recommended for new applications. It has never been ported to Microsoft Windows or Windows NT, and for X11 or the Macintosh it lacks important functionality -- in particular, it has no tools for the construction of dialogs. For most platforms, alternative, native solutions exist (though none are currently documented in this manual): Tkinter for UNIX under X11, native Xt with Motif or Athena widgets for UNIX under X11, Win32 for Windows and Windows NT, and a collection of native toolkit interfaces for the Macintosh.

Built-in Module `stdwin`

This module defines several new object types and functions that provide access to the functionality of STDWIN.

On Unix running X11, it can only be used if the `DISPLAY` environment variable is set or an explicit `-display displayname' argument is passed to the Python interpreter.

Functions have names that usually resemble their C STDWIN counterparts with the initial `w' dropped. Points are represented by pairs of integers; rectangles by pairs of points. For a complete description of STDWIN please refer to the documentation of STDWIN for C programmers (aforementioned CWI report).

Functions Defined in Module `stdwin`

The following functions are defined in the `stdwin` module:

function of module stdwin: **open** *(title)*

Open a new window whose initial title is given by the string argument. Return a window object; window object methods are described below.(16)

function of module stdwin: **getevent** *()*

Wait for and return the next event. An event is returned as a triple: the first element is the event type, a small integer; the second element is the window object to which the event applies, or `None` if it applies to no window in particular; the third element is type-dependent. Names for event types and command codes are defined in the standard module `stdwinevent`.

function of module stdwin: **pollevent** *()*

Return the next event, if one is immediately available. If no event is available, return `()`.

function of module stdwin: **getactive** *()*

Return the window that is currently active, or `None` if no window is currently active. (This can be emulated by monitoring WE_ACTIVATE and WE_DEACTIVATE events.)

function of module stdwin: **listfontnames** *(pattern)*

Return the list of font names in the system that match the pattern (a string). The pattern should normally be `'*'`; returns all available fonts. If the underlying window system is X11, other patterns follow the standard X11 font selection syntax (as used e.g. in resource definitions), i.e. the wildcard character `'*'` matches any sequence of characters (including none) and `'?'` matches any single character. On the Macintosh this function currently returns an empty list.

function of module stdwin: **setdefscrollbars** *(hflag, vflag)*

Set the flags controlling whether subsequently opened windows will have horizontal and/or vertical scroll bars.

function of module stdwin: **setdefwinpos** *(h, v)*

Set the default window position for windows opened subsequently.

function of module stdwin: **setdefwinsize** *(width, height)*

Set the default window size for windows opened subsequently.

function of module stdwin: **getdefscrollbars** *()*

Return the flags controlling whether subsequently opened windows will have horizontal and/or vertical scroll bars.

function of module stdwin: **getdefwinpos** *()*

Return the default window position for windows opened subsequently.

function of module stdwin: **getdefwinsize** *()*

Return the default window size for windows opened subsequently.

function of module stdwin: **getscrsize** *()*

Return the screen size in pixels.

function of module stdwin: **getscrmm** *()*

Return the screen size in millimeters.

function of module stdwin: **fetchcolor** *(colorname)*

Return the pixel value corresponding to the given color name. Return the default foreground color for unknown color names. Hint: the following code tests whether you are on a machine that supports more than two colors:


```
if stdwin.fetchcolor('black') <> \
          stdwin.fetchcolor('red') <> \
          stdwin.fetchcolor('white'):
    print 'color machine'
else:
    print 'monochrome machine'
```

function of module stdwin: **setfgcolor** *(pixel)*

Set the default foreground color. This will become the default foreground color of windows opened subsequently, including dialogs.

function of module stdwin: **setbgcolor** *(pixel)*

Set the default background color. This will become the default background color of windows opened subsequently, including dialogs.

function of module stdwin: **getfgcolor** *()*

Return the pixel value of the current default foreground color.

function of module stdwin: **getbgcolor** *()*

Return the pixel value of the current default background color.

function of module stdwin: **setfont** *(fontname)*

Set the current default font. This will become the default font for windows opened subsequently, and is also used by the text measuring functions `textwidth`, `textbreak`, `lineheight` and `baseline` below. This accepts two more optional parameters, size and style: Size is the font size (in `points'). Style is a single character specifying the style, as follows: `'b'` = bold, `'i'` = italic, `'o'` = bold + italic, `'u'` = underline; default style is roman. Size and style are ignored under X11 but used on the Macintosh. (Sorry for all this complexity -- a more uniform interface is being designed.)

function of module stdwin: **menucreate** *(title)*

Create a menu object referring to a global menu (a menu that appears in all windows). Methods of menu objects are described below. Note: normally, menus are created locally; see the window method `menucreate` below. **Warning:** the menu only appears in a window as long as the object returned by this call exists.

function of module stdwin: **newbitmap** *(width, height)*

Create a new bitmap object of the given dimensions. Methods of bitmap objects are described below. Not available on the Macintosh.

function of module stdwin: **fleep** *()*

Cause a beep or bell (or perhaps a `visual bell' or flash, hence the name).

function of module stdwin: **message** *(string)*

Display a dialog box containing the string. The user must click OK before the function returns.

function of module stdwin: **askync** *(prompt, default)*

Display a dialog that prompts the user to answer a question with yes or no. Return 0 for no, 1 for yes. If the user hits the Return key, the default (which must be 0 or 1) is returned. If the user cancels the dialog, the `KeyboardInterrupt` exception is raised.

function of module stdwin: **askstr** *(prompt, default)*

Display a dialog that prompts the user for a string. If the user hits the Return key, the default string is returned. If the user cancels the dialog, the `KeyboardInterrupt` exception is raised.

function of module stdwin: **askfile** *(prompt, default, new)*

Ask the user to specify a filename. If new is zero it must be an existing file; otherwise, it must be a new file. If the user cancels the dialog, the `KeyboardInterrupt` exception is raised.

function of module stdwin: **setcutbuffer** *(i, string)*

Store the string in the system's cut buffer number i, where it can be found (for pasting) by other applications. On X11, there are 8 cut buffers (numbered 0..7). Cut buffer number 0 is the `clipboard' on the Macintosh.

function of module stdwin: **getcutbuffer** *(i)*

Return the contents of the system's cut buffer number i.

function of module stdwin: **rotatecutbuffers** *(n)*

On X11, rotate the 8 cut buffers by n. Ignored on the Macintosh.

function of module stdwin: **getselection** *(i)*

Return X11 selection number i. Selections are not cut buffers. Selection numbers are defined in module `stdwinevents`. Selection `WS_PRIMARY` is the primary selection (used by xterm, for instance); selection `WS_SECONDARY` is the secondary selection; selection `WS_CLIPBOARD` is the clipboard selection (used by xclipboard). On the Macintosh, this always returns an empty string.

function of module stdwin: **resetselection** *(i)*

Reset selection number i, if this process owns it. (See window method `setselection()`).

function of module stdwin: **baseline** *()*

Return the baseline of the current font (defined by STDWIN as the vertical distance between the baseline and the top of the characters).

function of module stdwin: **lineheight** *()*

Return the total line height of the current font.

function of module stdwin: **textbreak** *(str, width)*

Return the number of characters of the string that fit into a space of width bits wide when drawn in th curent font.

function of module stdwin: **textwidth** *(str)*

Return the width in bits of the string when drawn in the current font.

function of module stdwin: **connectionnumber** *()*

function of module stdwin: **fileno** *()*

(X11 under UNIX only) Return the "connection number" used by the underlying X11 implementation. (This is normally the file number of the socket.) Both functions return the same value; `connectionnumber()` is named after the corresponding function in X11 and STDWIN, while `fileno()` makes it possible to use the `stdwin` module as a "file" object parameter to `select.select()`. Note that if `select()` implies that input is possible on `stdwin`, this does not guarantee that an event is ready -- it may be some internal communication going on between the X server and the client library. Thus, you should call `stdwin.pollevent()` until it returns `None` to check for events if you don't want your program to block. Because of internal buffering in X11, it is also possible that `stdwin.pollevent()` returns an event while `select()` does not find `stdwin` to be ready, so you should read any pending events with `stdwin.pollevent()` until it returns `None` before entering a blocking `select()` call.

Window Objects

Window objects are created by `stdwin.open()`. They are closed by their `close()` method or when they are garbage-collected. Window objects have the following methods:

Method: window **begindrawing** *()*

Return a drawing object, whose methods (described below) allow drawing in the window.

Method: window **change** *(rect)*

Invalidate the given rectangle; this may cause a draw event.

Method: window **gettitle** *()*

Returns the window's title string.

Method: window **getdocsize** *()*

Return a pair of integers giving the size of the document as set by `setdocsize()`.

Method: window **getorigin** *()*

Return a pair of integers giving the origin of the window with respect to the document.

Method: window **gettitle** *()*

Return the window's title string.

Method: window **getwinsize** *()*

Return a pair of integers giving the size of the window.

Method: window **getwinpos** *()*

Return a pair of integers giving the position of the window's upper left corner (relative to the upper left corner of the screen).

Method: window **menucreate** *(title)*

Create a menu object referring to a local menu (a menu that appears only in this window). Methods of menu objects are described below. **Warning:** the menu only appears as long as the object returned by this call exists.

Method: window **scroll** *(rect, point)*

Scroll the given rectangle by the vector given by the point.

Method: window **setdocsize** *(point)*

Set the size of the drawing document.

Method: window **setorigin** *(point)*

Move the origin of the window (its upper left corner) to the given point in the document.

Method: window **setselection** *(i, str)*

Attempt to set X11 selection number i to the string str. (See stdwin method `getselection()` for the meaning of i.) Return true if it succeeds. If succeeds, the window "owns" the selection until (a) another application takes ownership of the selection; or (b) the window is deleted; or (c) the application clears ownership by calling `stdwin.resetselection(i)`. When another application takes ownership of the selection, a `WE_LOST_SEL` event is received for no particular window and with the selection number as detail. Ignored on the Macintosh.

Method: window **settimer** *(dsecs)*

Schedule a timer event for the window in `dsecs/10` seconds.

Method: window **settitle** *(title)*

Set the window's title string.

Method: window **setwincursor** *(name)*

Set the window cursor to a cursor of the given name. It raises the `RuntimeError` exception if no cursor of the given name exists. Suitable names include `'ibeam'`, `'arrow'`, `'cross'`, `'watch'` and `'plus'`. On X11, there are many more (see `` `<X11/cursorfont.h>' ``).

Method: window **setwinpos** *(h, v)*

Set the the position of the window's upper left corner (relative to the upper left corner of the screen).

Method: window **setwinsize** *(width, height)*

Set the window's size.

Method: window **show** *(rect)*

Try to ensure that the given rectangle of the document is visible in the window.

Method: window **textcreate** *(rect)*

Create a text-edit object in the document at the given rectangle. Methods of text-edit objects are described below.

Method: window **setactive** *()*

Attempt to make this window the active window. If successful, this will generate a WE_ACTIVATE event (and a WE_DEACTIVATE event in case another window in this application became inactive).

Method: window **close** *()*

Discard the window object. It should not be used again.

Drawing Objects

Drawing objects are created exclusively by the window method `begindrawing()`. Only one drawing object can exist at any given time; the drawing object must be deleted to finish drawing. No drawing object may exist when `stdwin.getevent()` is called. Drawing objects have the following methods:

Method: drawing **box** *(rect)*

Draw a box just inside a rectangle.

Method: drawing **circle** *(center, radius)*

Draw a circle with given center point and radius.

Method: drawing **elarc** *(center, (rh, rv), (a1, a2))*

Draw an elliptical arc with given center point. `(rh, rv)` gives the half sizes of the horizontal and vertical radii. `(a1, a2)` gives the angles (in degrees) of the begin and end points. 0 degrees is at 3 o'clock, 90 degrees is at 12 o'clock.

Method: drawing **erase** *(rect)*

Erase a rectangle.

Method: drawing **fillcircle** *(center, radius)*

Draw a filled circle with given center point and radius.

Method: drawing **fillelarc** *(center, (rh, rv), (a1, a2))*

Draw a filled elliptical arc; arguments as for `elarc`.

Method: drawing **fillpoly** *(points)*

Draw a filled polygon given by a list (or tuple) of points.

Method: drawing **invert** *(rect)*

Invert a rectangle.

Method: drawing **line** *(p1, p2)*

Draw a line from point p1 to p2.

Method: drawing **paint** *(rect)*

Fill a rectangle.

Method: drawing **poly** *(points)*

Draw the lines connecting the given list (or tuple) of points.

Method: drawing **shade** *(rect, percent)*

Fill a rectangle with a shading pattern that is about percent percent filled.

Method: drawing **text** *(p, str)*

Draw a string starting at point p (the point specifies the top left coordinate of the string).

Method: drawing **xorcircle** *(center, radius)*

Method: drawing **xorelarc** *(center, (rh, rv), (a1, a2))*

Method: drawing **xorline** *(p1, p2)*

Method: drawing **xorpoly** *(points)*

Draw a circle, an elliptical arc, a line or a polygon, respectively, in XOR mode.

Method: drawing **setfgcolor** *()*

Method: drawing **setbgcolor** *()*

Method: drawing **getfgcolor** *()*

Method: drawing **getbgcolor** *()*

These functions are similar to the corresponding functions described above for the `stdwin` module, but affect or return the colors currently used for drawing instead of the global default colors. When a drawing object is created, its colors are set to the window's default colors, which are in turn initialized from the global default colors when the window is created.

Method: drawing **setfont** *()*

Method: drawing **baseline** *()*

Method: drawing **lineheight** *()*

Method: drawing **textbreak** *()*

Method: drawing **textwidth** *()*

These functions are similar to the corresponding functions described above for the `stdwin` module, but affect or use the current drawing font instead of the global default font. When a drawing object is created, its font is set to the window's default font, which is in turn initialized from the global default font when the window is created.

Method: drawing **bitmap** *(point, bitmap, mask)*

Draw the bitmap with its top left corner at point. If the optional mask argument is present, it should be either the same object as bitmap, to draw only those bits that are set in the bitmap, in the foreground color, or `None`, to draw all bits (ones are drawn in the foreground color, zeros in the background color). Not available on the Macintosh.

Method: drawing **cliprect** *(rect)*

Set the "clipping region" to a rectangle. The clipping region limits the effect of all drawing operations, until it is changed again or until the drawing object is closed. When a drawing object is created the clipping region is set to the entire window. When an object to be drawn falls partly outside the clipping region, the set of pixels drawn is the intersection of the clipping region and the set of pixels that would be drawn by the same operation in the absence of a clipping region.

Method: drawing **noclip** *()*

Reset the clipping region to the entire window.

Method: drawing **close** *()*

Method: drawing **enddrawing** *()*

Discard the drawing object. It should not be used again.

Menu Objects

A menu object represents a menu. The menu is destroyed when the menu object is deleted. The following methods are defined:

Method: menu **additem** *(text, shortcut)*

Add a menu item with given text. The shortcut must be a string of length 1, or omitted (to specify no shortcut).

Method: menu **setitem** *(i, text)*

Set the text of item number i.

Method: menu **enable** *(i, flag)*

Enable or disables item i.

Method: menu **check** *(i, flag)*

Set or clear the check mark for item i.

Method: menu **close** *()*

Discard the menu object. It should not be used again.

Bitmap Objects

A bitmap represents a rectangular array of bits. The top left bit has coordinate (0, 0). A bitmap can be drawn with the `bitmap` method of a drawing object. Bitmaps are currently not available on the Macintosh.

The following methods are defined:

Method: bitmap **getsize** *()*

Return a tuple representing the width and height of the bitmap. (This returns the values that have been passed to the `newbitmap` function.)

Method: bitmap **setbit** *(point, bit)*

Set the value of the bit indicated by point to bit.

Method: bitmap **getbit** *(point)*

Return the value of the bit indicated by point.

Method: bitmap **close** *()*

Discard the bitmap object. It should not be used again.

Text-edit Objects

A text-edit object represents a text-edit block. For semantics, see the STDWIN documentation for C programmers. The following methods exist:

Method: text-edit **arrow** *(code)*

Pass an arrow event to the text-edit block. The code must be one of `WC_LEFT`, `WC_RIGHT`, `WC_UP` or `WC_DOWN` (see module `stdwinevents`).

Method: text-edit **draw** *(rect)*

Pass a draw event to the text-edit block. The rectangle specifies the redraw area.

Method: text-edit **event** *(type, window, detail)*

Pass an event gotten from `stdwin.getevent()` to the text-edit block. Return true if the event was

handled.

Method: text-edit **getfocus** *()*

Return 2 integers representing the start and end positions of the focus, usable as slice indices on the string returned by `gettext()`.

Method: text-edit **getfocustext** *()*

Return the text in the focus.

Method: text-edit **getrect** *()*

Return a rectangle giving the actual position of the text-edit block. (The bottom coordinate may differ from the initial position because the block automatically shrinks or grows to fit.)

Method: text-edit **gettext** *()*

Return the entire text buffer.

Method: text-edit **move** *(rect)*

Specify a new position for the text-edit block in the document.

Method: text-edit **replace** *(str)*

Replace the text in the focus by the given string. The new focus is an insert point at the end of the string.

Method: text-edit **setfocus** *(i, j)*

Specify the new focus. Out-of-bounds values are silently clipped.

Method: text-edit **settext** *(str)*

Replace the entire text buffer by the given string and set the focus to `(0, 0)`.

Method: text-edit **setview** *(rect)*

Set the view rectangle to rect. If rect is `None`, viewing mode is reset. In viewing mode, all output from the text-edit object is clipped to the viewing rectangle. This may be useful to implement your own scrolling text subwindow.

Method: text-edit **close** *()*

Discard the text-edit object. It should not be used again.

Example

Here is a minimal example of using STDWIN in Python. It creates a window and draws the string "Hello world" in the top left corner of the window. The window will be correctly redrawn when covered and re-exposed. The program quits when the close icon or menu item is requested.

```
import stdwin
```


```
from stdwinevents import *

def main():
    mywin = stdwin.open('Hello')
    #
    while 1:
        (type, win, detail) = stdwin.getevent()
        if type == WE_DRAW:
            draw = win.begindrawing()
            draw.text((0, 0), 'Hello, world')
            del draw
        elif type == WE_CLOSE:
            break

main()
```

Standard Module `stdwinevents`

This module defines constants used by STDWIN for event types (`WE_ACTIVATE` etc.), command codes (`WC_LEFT` etc.) and selection types (`WS_PRIMARY` etc.). Read the file for details. Suggested usage is

```
>>> from stdwinevents import *
>>>
```

Standard Module `rect`

This module contains useful operations on rectangles. A rectangle is defined as in module `stdwin`: a pair of points, where a point is a pair of integers. For example, the rectangle

```
(10, 20), (90, 80)
```

is a rectangle whose left, top, right and bottom edges are 10, 20, 90 and 80, respectively. Note that the positive vertical axis points down (as in `stdwin`).

The module defines the following objects:

exception: module rect **error**

The exception raised by functions in this module when they detect an error. The exception argument is a string describing the problem in more detail.

data: module rect **empty**

The rectangle returned when some operations return an empty result. This makes it possible to quickly check whether a result is empty:


```
>>> import rect
>>> r1 = (10, 20), (90, 80)
>>> r2 = (0, 0), (10, 20)
>>> r3 = rect.intersect([r1, r2])
>>> if r3 is rect.empty: print 'Empty intersection'
Empty intersection
>>>
```

function of module rect: **is_empty** *(r)*

Returns true if the given rectangle is empty. A rectangle `(left, top), (right, bottom)` is empty if `left >= right` or `top => bottom`.

function of module rect: **intersect** *(list)*

Returns the intersection of all rectangles in the list argument. It may also be called with a tuple argument. Raises `rect.error` if the list is empty. Returns `rect.empty` if the intersection of the rectangles is empty.

function of module rect: **union** *(list)*

Returns the smallest rectangle that contains all non-empty rectangles in the list argument. It may also be called with a tuple argument or with two or more rectangles as arguments. Returns `rect.empty` if the list is empty or all its rectangles are empty.

function of module rect: **pointinrect** *(point, rect)*

Returns true if the point is inside the rectangle. By definition, a point `(h, v)` is inside a rectangle `(left, top), (right, bottom)` if `left <= h < right` and `top <= v < bottom`.

function of module rect: **inset** *(rect, (dh, dv))*

Returns a rectangle that lies inside the `rect` argument by dh pixels horizontally and dv pixels vertically. If dh or dv is negative, the result lies outside rect.

function of module rect: **rect2geom** *(rect)*

Converts a rectangle to geometry representation: `(left, top), (width, height)`.

function of module rect: **geom2rect** *(geom)*

Converts a rectangle given in geometry representation back to the standard rectangle representation `(left, top), (right, bottom)`.

Go to the previous, next section.

Go to the previous, next section.

SGI IRIX Specific Services

The modules described in this chapter provide interfaces to features that are unique to SGI's IRIX operating system (versions 4 and 5).

Built-in Module `al`

This module provides access to the audio facilities of the SGI Indy and Indigo workstations. See section 3A of the IRIX man pages for details. You'll need to read those man pages to understand what these functions do! Some of the functions are not available in IRIX releases before 4.0.5. Again, see the manual to check whether a specific function is available on your platform.

All functions and methods defined in this module are equivalent to the C functions with `AL' prefixed to their name.

Symbolic constants from the C header file `` `<audio.h>' `` are defined in the standard module `AL`, see below.

Warning: the current version of the audio library may dump core when bad argument values are passed rather than returning an error status. Unfortunately, since the precise circumstances under which this may happen are undocumented and hard to check, the Python interface can provide no protection against this kind of problems. (One example is specifying an excessive queue size -- there is no documented upper limit.)

The module defines the following functions:

function of module al: **openport** *(name, direction[, config])*

The name and direction arguments are strings. The optional config argument is a configuration object as returned by `al.newconfig()`. The return value is an port object; methods of port objects are described below.

function of module al: **newconfig** *()*

The return value is a new configuration object; methods of configuration objects are described below.

function of module al: **queryparams** *(device)*

The device argument is an integer. The return value is a list of integers containing the data returned by ALqueryparams().

function of module al: **getparams** *(device, list)*

The device argument is an integer. The list argument is a list such as returned by `queryparams`; it is modified in place (!).

function of module al: **setparams** *(device, list)*

The device argument is an integer. The list argument is a list such as returned by `al.queryparams`.

Configuration Objects

Configuration objects (returned by `al.newconfig()` have the following methods:

Method: audio configuration object **getqueuesize** *()*

Return the queue size.

Method: audio configuration object **setqueuesize** *(size)*

Set the queue size.

Method: audio configuration object **getwidth** *()*

Get the sample width.

Method: audio configuration object **setwidth** *(width)*

Set the sample width.

Method: audio configuration object **getchannels** *()*

Get the channel count.

Method: audio configuration object **setchannels** *(nchannels)*

Set the channel count.

Method: audio configuration object **getsampfmt** *()*

Get the sample format.

Method: audio configuration object **setsampfmt** *(sampfmt)*

Set the sample format.

Method: audio configuration object **getfloatmax** *()*

Get the maximum value for floating sample formats.

Method: audio configuration object **setfloatmax** *(floatmax)*

Set the maximum value for floating sample formats.

Port Objects

Port objects (returned by `al.openport()` have the following methods:

Method: audio port object **closeport** *()*

Close the port.

Method: audio port object **getfd** *()*

Return the file descriptor as an int.

Method: audio port object **getfilled** *()*

Return the number of filled samples.

Method: audio port object **getfillable** *()*

Return the number of fillable samples.

Method: audio port object **readsamps** *(nsamples)*

Read a number of samples from the queue, blocking if necessary. Return the data as a string containing the raw data, (e.g., 2 bytes per sample in big-endian byte order (high byte, low byte) if you have set the sample width to 2 bytes).

Method: audio port object **writesamps** *(samples)*

Write samples into the queue, blocking if necessary. The samples are encoded as described for the `readsamps` return value.

Method: audio port object **getfillpoint** *()*

Return the `fill point'.

Method: audio port object **setfillpoint** *(fillpoint)*

Set the `fill point'.

Method: audio port object **getconfig** *()*

Return a configuration object containing the current configuration of the port.

Method: audio port object **setconfig** *(config)*

Set the configuration from the argument, a configuration object.

Method: audio port object **getstatus** *(list)*

Get status information on last error.

Standard Module AL

This module defines symbolic constants needed to use the built-in module `al` (see above); they are equivalent to those defined in the C header file `` `<audio.h>' `` except that the name prefix `AL_' is omitted. Read the module source for a complete list of the defined names. Suggested use:

```
import al
```

```
from AL import *
```

Built-in Module cd

This module provides an interface to the Silicon Graphics CD library. It is available only on Silicon Graphics systems.

The way the library works is as follows. A program opens the CD-ROM device with `cd.open()` and creates a parser to parse the data from the CD with `cd.createparser()`. The object returned by `cd.open()` can be used to read data from the CD, but also to get status information for the CD-ROM device, and to get information about the CD, such as the table of contents. Data from the CD is passed to the parser, which parses the frames, and calls any callback functions that have previously been added.

An audio CD is divided into tracks or programs (the terms are used interchangeably). Tracks can be subdivided into indices. An audio CD contains a table of contents which gives the starts of the tracks on the CD. Index 0 is usually the pause before the start of a track. The start of the track as given by the table of contents is normally the start of index 1.

Positions on a CD can be represented in two ways. Either a frame number or a tuple of three values, minutes, seconds and frames. Most functions use the latter representation. Positions can be both relative to the beginning of the CD, and to the beginning of the track.

Module `cd` defines the following functions and constants:

function of module cd: **createparser** *()*

Create and return an opaque parser object. The methods of the parser object are described below.

function of module cd: **msftoframe** *(min, sec, frame)*

Converts a `(minutes, seconds, frames)` triple representing time in absolute time code into the corresponding CD frame number.

function of module cd: **open** *([device[, mode]])*

Open the CD-ROM device. The return value is an opaque player object; methods of the player object are described below. The device is the name of the SCSI device file, e.g. /dev/scsi/sc0d4l0, or `None`. If omited or `None`, the hardware inventory is consulted to locate a CD-ROM drive. The `mode`, if not omited, should be the string 'r'.

The module defines the following variables:

data: module cd **error**

Exception raised on various errors.

data: module cd **DATASIZE**

The size of one frame's worth of audio data. This is the size of the audio data as passed to the callback of type `audio`.

data: module cd **BLOCKSIZE**

The size of one uninterpreted frame of audio data.

The following variables are states as returned by `getstatus`:

data: module cd **READY**

The drive is ready for operation loaded with an audio CD.

data: module cd **NODISC**

The drive does not have a CD loaded.

data: module cd **CDROM**

The drive is loaded with a CD-ROM. Subsequent play or read operations will return I/O errors.

data: module cd **ERROR**

An error aoocurred while trying to read the disc or its table of contents.

data: module cd **PLAYING**

The drive is in CD player mode playing an audio CD through its audio jacks.

data: module cd **PAUSED**

The drive is in CD layer mode with play paused.

data: module cd **STILL**

The equivalent of `PAUSED` on older (non 3301) model Toshiba CD-ROM drives. Such drives have never been shipped by SGI.

data: module cd **audio**

data: module cd **pnum**

data: module cd **index**

data: module cd **ptime**

data: module cd **atime**

data: module cd **catalog**

data: module cd **ident**

data: module cd **control**

Integer constants describing the various types of parser callbacks that can be set by the `addcallback()` method of CD parser objects (see below).

Player objects (returned by `cd.open()`) have the following methods:

Method: CD player object **allowremoval** *()*

Unlocks the eject button on the CD-ROM drive permitting the user to eject the caddy if desired.

Method: CD player object **bestreadsize** *()*

Returns the best value to use for the `num_frames` parameter of the `readda` method. Best is defined as the value that permits a continuous flow of data from the CD-ROM drive.

Method: CD player object **close** *()*

Frees the resources associated with the player object. After calling `close`, the methods of the object should no longer be used.

Method: CD player object **eject** *()*

Ejects the caddy from the CD-ROM drive.

Method: CD player object **getstatus** *()*

Returns information pertaining to the current state of the CD-ROM drive. The returned information is a tuple with the following values: `state`, `track`, `rtime`, `atime`, `ttime`, `first`, `last`, `scsi_audio`, `cur_block`. `rtime` is the time relative to the start of the current track; `atime` is the time relative to the beginning of the disc; `ttime` is the total time on the disc. For more information on the meaning of the values, see the manual for CDgetstatus. The value of `state` is one of the following: `cd.ERROR`, `cd.NODISC`, `cd.READY`, `cd.PLAYING`, `cd.PAUSED`, `cd.STILL`, or `cd.CDROM`.

Method: CD player object **gettrackinfo** *(track)*

Returns information about the specified track. The returned information is a tuple consisting of two elements, the start time of the track and the duration of the track.

Method: CD player object **msftoblock** *(min, sec, frame)*

Converts a minutes, seconds, frames triple representing a time in absolute time code into the corresponding logical block number for the given CD-ROM drive. You should use `cd.msftoframe()` rather than `msftoblock()` for comparing times. The logical block number differs from the frame number by an offset required by certain CD-ROM drives.

Method: CD player object **play** *(start, play)*

Starts playback of an audio CD in the CD-ROM drive at the specified track. The audio output appears on the CD-ROM drive's headphone and audio jacks (if fitted). Play stops at the end of the disc. `start` is the number of the track at which to start playing the CD; if `play` is 0, the CD will be set to an initial paused state. The method `togglepause()` can then be used to commence play.

Method: CD player object **playabs** *(min, sec, frame, play)*

Like `play()`, except that the start is given in minutes, seconds, frames instead of a track number.

Method: CD player object **playtrack** *(start, play)*

Like `play()`, except that playing stops at the end of the track.

Method: CD player object **playtrackabs** *(track, min, sec, frame, play)*

Like `play()`, except that playing begins at the spcified absolute time and ends at the end of the specified track.

Method: CD player object **preventremoval** *()*

Locks the eject button on the CD-ROM drive thus preventing the user from arbitrarily ejecting the caddy.

Method: CD player object **readda** *(num_frames)*

Reads the specified number of frames from an audio CD mounted in the CD-ROM drive. The return value is a string representing the audio frames. This string can be passed unaltered to the `parseframe` method of the parser object.

Method: CD player object **seek** *(min, sec, frame)*

Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to an absolute time code location specified in minutes, seconds, and frames. The return value is the logical block number to which the pointer has been set.

Method: CD player object **seekblock** *(block)*

Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to the specified logical block number. The return value is the logical block number to which the pointer has been set.

Method: CD player object **seektrack** *(track)*

Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to the specified track. The return value is the logical block number to which the pointer has been set.

Method: CD player object **stop** *()*

Stops the current playing operation.

Method: CD player object **togglepause** *()*

Pauses the CD if it is playing, and makes it play if it is paused.

Parser objects (returned by `cd.createparser()`) have the following methods:

Method: CD parser object **addcallback** *(type, func, arg)*

Adds a callback for the parser. The parser has callbacks for eight different types of data in the digital audio data stream. Constants for these types are defined at the `cd` module level (see above). The callback is called as follows: `func(arg, type, data)`, where `arg` is the user supplied argument, `type` is the particular type of callback, and `data` is the data returned for this `type` of callback. The type of the data depends on the `type` of callback as follows:

`cd.audio`:

The argument is a string which can be passed unmodified to `al.writesamps()`.

`cd.pnum`:

The argument is an integer giving the program (track) number.

`cd.index`:

The argument is an integer giving the index number.

`cd.ptime`:

The argument is a tuple consisting of the program time in minutes, seconds, and frames.

`cd.atime`:

The argument is a tuple consisting of the absolute time in minutes, seconds, and frames.

`cd.catalog`:

The argument is a string of 13 characters, giving the catalog number of the CD.

`cd.ident`:

The argument is a string of 12 characters, giving the ISRC identification number of the recording. The string consists of two characters country code, three characters owner code, two characters giving the year, and five characters giving a serial number.

`cd.control`:

The argument is an integer giving the control bits from the CD subcode data.

Method: CD parser object **deleteparser** *()*

Deletes the parser and frees the memory it was using. The object should not be used after this call. This call is done automatically when the last reference to the object is removed.

Method: CD parser object **parseframe** *(frame)*

Parses one or more frames of digital audio data from a CD such as returned by `readda`. It determines which subcodes are present in the data. If these subcodes have changed since the last frame, then `parseframe` executes a callback of the appropriate type passing to it the subcode data found in the frame. Unlike the C function, more than one frame of digital audio data can be passed to this method.

Method: CD parser object **removecallback** *(type)*

Removes the callback for the given `type`.

Method: CD parser object **resetparser** *()*

Resets the fields of the parser used for tracking subcodes to an initial state. `resetparser` should be called after the disc has been changed.

Built-in Module `fl`

This module provides an interface to the FORMS Library by Mark Overmars. The source for the library can be retrieved by anonymous ftp from host `ftp.cs.ruu.nl', directory `SGI/FORMS'. It was last tested with version 2.0b.

Most functions are literal translations of their C equivalents, dropping the initial `fl_' from their name. Constants used by the library are defined in module `FL` described below.

The creation of objects is a little different in Python than in C: instead of the `current form' maintained by the library to which new FORMS objects are added, all functions that add a FORMS object to a form are methods of the Python object representing the form. Consequently, there are no Python equivalents for the C functions `fl_addto_form` and `fl_end_form`, and the equivalent of `fl_bgn_form` is called `fl.make_form`.

Watch out for the somewhat confusing terminology: FORMS uses the word object for the buttons, sliders etc. that you can place in a form. In Python, `object' means any value. The Python interface to FORMS introduces two new Python object types: form objects (representing an entire form) and FORMS objects (representing one button, slider etc.). Hopefully this isn't too confusing...

There are no `free objects' in the Python interface to FORMS, nor is there an easy way to add object classes written in Python. The FORMS interface to GL event handling is available, though, so you can mix FORMS with pure GL windows.

Please note: importing `fl` implies a call to the GL function `foreground()` and to the FORMS routine `fl_init()`.

Functions Defined in Module `fl`

Module `fl` defines the following functions. For more information about what they do, see the description of the equivalent C function in the FORMS documentation:

function of module fl: **make_form** *(type, width, height)*

Create a form with given type, width and height. This returns a form object, whose methods are described below.

function of module fl: **do_forms** *()*

The standard FORMS main loop. Returns a Python object representing the FORMS object needing interaction, or the special value `FL.EVENT`.

function of module fl: **check_forms** *()*

Check for FORMS events. Returns what `do_forms` above returns, or `None` if there is no event that immediately needs interaction.

function of module fl: **set_event_call_back** *(function)*

Set the event callback function.

function of module fl: **set_graphics_mode** *(rgbmode, doublebuffering)*

Set the graphics modes.

function of module fl: **get_rgbmode** *()*

Return the current rgb mode. This is the value of the C global variable `fl_rgbmode`.

function of module fl: **show_message** *(str1, str2, str3)*

Show a dialog box with a three-line message and an OK button.

function of module fl: **show_question** *(str1, str2, str3)*

Show a dialog box with a three-line message and YES and NO buttons. It returns `1` if the user pressed YES, `0` if NO.

function of module fl: **show_choice** *(str1, str2, str3, but1[, but2, but3])*

Show a dialog box with a three-line message and up to three buttons. It returns the number of the button clicked by the user (`1`, `2` or `3`).

function of module fl: **show_input** *(prompt, default)*

Show a dialog box with a one-line prompt message and text field in which the user can enter a string. The second argument is the default input string. It returns the string value as edited by the user.

function of module fl: **show_file_selector** *(message, directory, pattern, default)*

Show a dialog box in which the user can select a file. It returns the absolute filename selected by the user, or `None` if the user presses Cancel.

function of module fl: **get_directory** *()*

function of module fl: **get_pattern** *()*

function of module fl: **get_filename** *()*

These functions return the directory, pattern and filename (the tail part only) selected by the user in the last `show_file_selector` call.

function of module fl: **qdevice** *(dev)*

function of module fl: **unqdevice** *(dev)*

function of module fl: **isqueued** *(dev)*

function of module fl: **qtest** *()*

function of module fl: **qread** *()*

function of module fl: **qreset** *()*

function of module fl: **qenter** *(dev, val)*

function of module fl: **get_mouse** *()*

function of module fl: **tie** *(button, valuator1, valuator2)*

These functions are the FORMS interfaces to the corresponding GL functions. Use these if you want to

handle some GL events yourself when using `fl.do_events`. When a GL event is detected that FORMS cannot handle, `fl.do_forms()` returns the special value `FL.EVENT` and you should cal `fl.qread()` to read the event from the queue. Don't use the equivalent GL functions!

function of module fl: **color** *()*

function of module fl: **mapcolor** *()*

function of module fl: **getmcolor** *()*

See the description in the FORMS documentation of `fl_color`, `fl_mapcolor` and `fl_getmcolor`.

Form Objects

Form objects (returned by `fl.make_form()` above) have the following methods. Each method corresponds to a C function whose name is prefixed with `fl_'; and whose first argument is a form pointer; please refer to the official FORMS documentation for descriptions.

All the `add_...' functions return a Python object representing the FORMS object. Methods of FORMS objects are described below. Most kinds of FORMS object also have some methods specific to that kind; these methods are listed here.

Method: form object **show_form** *(placement, bordertype, name)*

Show the form.

Method: form object **hide_form** *()*

Hide the form.

Method: form object **redraw_form** *()*

Redraw the form.

Method: form object **set_form_position** *(x, y)*

Set the form's position.

Method: form object **freeze_form** *()*

Freeze the form.

Method: form object **unfreeze_form** *()*

Unfreeze the form.

Method: form object **activate_form** *()*

Activate the form.

Method: form object **deactivate_form** *()*

Deactivate the form.

Method: form object **bgn_group** *()*

Begin a new group of objects; return a group object.

Method: form object **end_group** *()*

End the current group of objects.

Method: form object **find_first** *()*

Find the first object in the form.

Method: form object **find_last** *()*

Find the last object in the form.

Method: form object **add_box** *(type, x, y, w, h, name)*

Add a box object to the form. No extra methods.

Method: form object **add_text** *(type, x, y, w, h, name)*

Add a text object to the form. No extra methods.

Method: form object **add_clock** *(type, x, y, w, h, name)*

Add a clock object to the form.
Method: `get_clock`.

Method: form object **add_button** *(type, x, y, w, h, name)*

Add a button object to the form.
Methods: `get_button`, `set_button`.

Method: form object **add_lightbutton** *(type, x, y, w, h, name)*

Add a lightbutton object to the form.
Methods: `get_button`, `set_button`.

Method: form object **add_roundbutton** *(type, x, y, w, h, name)*

Add a roundbutton object to the form.
Methods: `get_button`, `set_button`.

Method: form object **add_slider** *(type, x, y, w, h, name)*

Add a slider object to the form.
Methods: `set_slider_value`, `get_slider_value`, `set_slider_bounds`, `get_slider_bounds`, `set_slider_return`, `set_slider_size`, `set_slider_precision`, `set_slider_step`.

Method: form object **add_valslider** *(type, x, y, w, h, name)*

Add a valslider object to the form.
Methods: `set_slider_value`, `get_slider_value`, `set_slider_bounds`, `get_slider_bounds`, `set_slider_return`, `set_slider_size`, `set_slider_precision`, `set_slider_step`.

Method: form object **add_dial** *(type, x, y, w, h, name)*

Add a dial object to the form.
Methods: `set_dial_value`, `get_dial_value`, `set_dial_bounds`, `get_dial_bounds`.

Method: form object **add_positioner** *(type, x, y, w, h, name)*

Add a positioner object to the form.
Methods: `set_positioner_xvalue`, `set_positioner_yvalue`, `set_positioner_xbounds`, `set_positioner_ybounds`, `get_positioner_xvalue`, `get_positioner_yvalue`, `get_positioner_xbounds`, `get_positioner_ybounds`.

Method: form object **add_counter** *(type, x, y, w, h, name)*

Add a counter object to the form.
Methods: `set_counter_value`, `get_counter_value`, `set_counter_bounds`, `set_counter_step`, `set_counter_precision`, `set_counter_return`.

Method: form object **add_input** *(type, x, y, w, h, name)*

Add a input object to the form.
Methods: `set_input`, `get_input`, `set_input_color`, `set_input_return`.

Method: form object **add_menu** *(type, x, y, w, h, name)*

Add a menu object to the form.
Methods: `set_menu`, `get_menu`, `addto_menu`.

Method: form object **add_choice** *(type, x, y, w, h, name)*

Add a choice object to the form.
Methods: `set_choice`, `get_choice`, `clear_choice`, `addto_choice`, `replace_choice`, `delete_choice`, `get_choice_text`, `set_choice_fontsize`, `set_choice_fontstyle`.

Method: form object **add_browser** *(type, x, y, w, h, name)*

Add a browser object to the form.
Methods: `set_browser_topline`, `clear_browser`, `add_browser_line`, `addto_browser`, `insert_browser_line`, `delete_browser_line`, `replace_browser_line`, `get_browser_line`, `load_browser`, `get_browser_maxline`, `select_browser_line`, `deselect_browser_line`, `deselect_browser`, `isselected_browser_line`, `get_browser`, `set_browser_fontsize`, `set_browser_fontstyle`, `set_browser_specialkey`.

Method: form object **add_timer** *(type, x, y, w, h, name)*

Add a timer object to the form.
Methods: `set_timer`, `get_timer`.

Form objects have the following data attributes; see the FORMS documentation:

Name

Type -- *Meaning*

`window`

int (read-only) -- GL window id

`w`

float -- form width

`h`

float -- form height

`x`

float -- form x origin

`y`

float -- form y origin

`deactivated`

int -- nonzero if form is deactivated

`visible`

int -- nonzero if form is visible

`frozen`

int -- nonzero if form is frozen

`doublebuf`

int -- nonzero if double buffering on

FORMS Objects

Besides methods specific to particular kinds of FORMS objects, all FORMS objects also have the following methods:

Method: FORMS object **set_call_back** *(function, argument)*

Set the object's callback function and argument. When the object needs interaction, the callback function will be called with two arguments: the object, and the callback argument. (FORMS objects without a callback function are returned by `fl.do_forms()` or `fl.check_forms()` when they need interaction.) Call this method without arguments to remove the callback function.

Method: FORMS object **delete_object** *()*

Delete the object.

Method: FORMS object **show_object** *()*

Show the object.

Method: FORMS object **hide_object** *()*

Hide the object.

Method: FORMS object **redraw_object** *()*

Redraw the object.

Method: FORMS object **freeze_object** *()*

Freeze the object.

Method: FORMS object **unfreeze_object** *()*

Unfreeze the object.

FORMS objects have these data attributes; see the FORMS documentation:

Name

Type -- *Meaning*

`objclass`

int (read-only) -- object class

`type`

int (read-only) -- object type

`boxtype`

int -- box type

`x`

float -- x origin

`y`

float -- y origin

`w`

float -- width

`h`

float -- height

`col1`

int -- primary color

`col2`

int -- secondary color

`align`

int -- alignment

`lcol`

int -- label color

`lsize`
: float -- label font size

`label`
: string -- label string

`lstyle`
: int -- label style

`pushed`
: int (read-only) -- (see FORMS docs)

`focus`
: int (read-only) -- (see FORMS docs)

`belowmouse`
: int (read-only) -- (see FORMS docs)

`frozen`
: int (read-only) -- (see FORMS docs)

`active`
: int (read-only) -- (see FORMS docs)

`input`
: int (read-only) -- (see FORMS docs)

`visible`
: int (read-only) -- (see FORMS docs)

`radio`
: int (read-only) -- (see FORMS docs)

`automatic`
: int (read-only) -- (see FORMS docs)

Standard Module `FL`

This module defines symbolic constants needed to use the built-in module `fl` (see above); they are equivalent to those defined in the C header file `` `<forms.h>' `` except that the name prefix `FL_' is omitted. Read the module source for a complete list of the defined names. Suggested use:

```
import fl
from FL import *
```

Standard Module `flp`

This module defines functions that can read form definitions created by the `form designer' (`fdesign`) program that comes with the FORMS library (see module `fl` above).

For now, see the file `` `flp.doc' `` in the Python library source directory for a description.

XXX A complete description should be inserted here!

Built-in Module `fm`

This module provides access to the IRIS *Font Manager* library. It is available only on Silicon Graphics machines. See also: 4Sight User's Guide, Section 1, Chapter 5: Using the IRIS Font Manager.

This is not yet a full interface to the IRIS Font Manager. Among the unsupported features are: matrix operations; cache operations; character operations (use string operations instead); some details of font info; individual glyph metrics; and printer matching.

It supports the following operations:

function of module fm: **init** *()*

Initialization function. Calls `fminit()`. It is normally not necessary to call this function, since it is called automatically the first time the `fm` module is imported.

function of module fm: **findfont** *(fontname)*

Return a font handle object. Calls `fmfindfont(fontname)`.

function of module fm: **enumerate** *()*

Returns a list of available font names. This is an interface to `fmenumerate()`.

function of module fm: **prstr** *(string)*

Render a string using the current font (see the `setfont()` font handle method below). Calls `fmprstr(string)`.

function of module fm: **setpath** *(string)*

Sets the font search path. Calls `fmsetpath(string)`. (XXX Does not work!?!)

function of module fm: **fontpath** *()*

Returns the current font search path.

Font handle objects support the following operations:

Method: font handle **scalefont** *(factor)*

Returns a handle for a scaled version of this font. Calls `fmscalefont(fh, factor)`.

Method: font handle **setfont** *()*

Makes this font the current font. Note: the effect is undone silently when the font handle object is deleted. Calls `fmsetfont(fh)`.

Method: font handle **getfontname** *()*

Returns this font's name. Calls `fmgetfontname(fh)`.

Method: font handle **getcomment** *()*

Returns the comment string associated with this font. Raises an exception if there is none. Calls `fmgetcomment(fh)`.

Method: font handle **getfontinfo** *()*

Returns a tuple giving some pertinent data about this font. This is an interface to `fmgetfontinfo()`. The returned tuple contains the following numbers: `(printermatched, fixed_width, xorig, yorig, xsize, ysize, height, nglyphs)`.

Method: font handle **getstrwidth** *(string)*

Returns the width, in pixels, of the string when drawn in this font. Calls `fmgetstrwidth(fh, string)`.

Built-in Module `gl`

This module provides access to the Silicon Graphics *Graphics Library*. It is available only on Silicon Graphics machines.

Warning: Some illegal calls to the GL library cause the Python interpreter to dump core. In particular, the use of most GL calls is unsafe before the first window is opened.

The module is too large to document here in its entirety, but the following should help you to get started. The parameter conventions for the C functions are translated to Python as follows:

- All (short, long, unsigned) int values are represented by Python integers.
- All float and double values are represented by Python floating point numbers. In most cases, Python integers are also allowed.
- All arrays are represented by one-dimensional Python lists. In most cases, tuples are also allowed.
- All string and character arguments are represented by Python strings, for instance, `winopen('Hi There!')` and `rotate(900, 'z')`.
- All (short, long, unsigned) integer arguments or return values that are only used to specify the length of an array argument are omitted. For example, the C call

  ```
  lmdef(deftype, index, np, props)
  ```

  is translated to Python as


```
lmdef(deftype, index, props)
```

- Output arguments are omitted from the argument list; they are transmitted as function return values instead. If more than one value must be returned, the return value is a tuple. If the C function has both a regular return value (that is not omitted because of the previous rule) and an output argument, the return value comes first in the tuple. Examples: the C call

  ```
  getmcolor(i, &red, &green, &blue)
  ```

  is translated to Python as

  ```
  red, green, blue = getmcolor(i)
  ```

The following functions are non-standard or have special argument conventions:

function of module gl: **varray** *(argument)*

Equivalent to but faster than a number of `v3d()` calls. The argument is a list (or tuple) of points. Each point must be a tuple of coordinates `(x, y, z)` or `(x, y)`. The points may be 2- or 3-dimensional but must all have the same dimension. Float and int values may be mixed however. The points are always converted to 3D double precision points by assuming `z = 0.0` if necessary (as indicated in the man page), and for each point `v3d()` is called.

function of module gl: **nvarray** *()*

Equivalent to but faster than a number of `n3f` and `v3f` calls. The argument is an array (list or tuple) of pairs of normals and points. Each pair is a tuple of a point and a normal for that point. Each point or normal must be a tuple of coordinates `(x, y, z)`. Three coordinates must be given. Float and int values may be mixed. For each pair, `n3f()` is called for the normal, and then `v3f()` is called for the point.

function of module gl: **vnarray** *()*

Similar to `nvarray()` but the pairs have the point first and the normal second.

function of module gl: **nurbssurface** *(s_k, t_k, ctl, s_ord, t_ord, type)*

Defines a nurbs surface. The dimensions of `ctl[][]` are computed as follows: `[len(s_k) - s_ord]`, `[len(t_k) - t_ord]`.

function of module gl: **nurbscurve** *(knots, ctlpoints, order, type)*

Defines a nurbs curve. The length of ctlpoints is `len(knots) - order`.

function of module gl: **pwlcurve** *(points, type)*

Defines a piecewise-linear curve. points is a list of points. type must be `N_ST`.

function of module gl: **pick** *(n)*

function of module gl: **select** *(n)*

The only argument to these functions specifies the desired size of the pick or select buffer.

function of module gl: **endpick** *()*

function of module gl: **endselect** *()*

These functions have no arguments. They return a list of integers representing the used part of the pick/select buffer. No method is provided to detect buffer overrun.

Here is a tiny but complete example GL program in Python:

```
import gl, GL, time

def main():
    gl.foreground()
    gl.prefposition(500, 900, 500, 900)
    w = gl.winopen('CrissCross')
    gl.ortho2(0.0, 400.0, 0.0, 400.0)
    gl.color(GL.WHITE)
    gl.clear()
    gl.color(GL.RED)
    gl.bgnline()
    gl.v2f(0.0, 0.0)
    gl.v2f(400.0, 400.0)
    gl.endline()
    gl.bgnline()
    gl.v2f(400.0, 0.0)
    gl.v2f(0.0, 400.0)
    gl.endline()
    time.sleep(5)

main()
```

Standard Modules `GL` and `DEVICE`

These modules define the constants used by the Silicon Graphics *Graphics Library* that C programmers find in the header files `` `<gl/gl.h>' `` and `` `<gl/device.h>' ``. Read the module source files for details.

Built-in Module `imgfile`

The imgfile module allows python programs to access SGI imglib image files (also known as `` `.rgb' `` files). The module is far from complete, but is provided anyway since the functionality that there is is enough in some cases. Currently, colormap files are not supported.

The module defines the following variables and functions:

exception: module imgfile **error**

This exception is raised on all errors, such as unsupported file type, etc.

function of module imgfile: **getsizes** *(file)*

This function returns a tuple `(x, y, z)` where x and y are the size of the image in pixels and z is the number of bytes per pixel. Only 3 byte RGB pixels and 1 byte greyscale pixels are currently supported.

function of module imgfile: **read** *(file)*

This function reads and decodes the image on the specified file, and returns it as a python string. The string has either 1 byte greyscale pixels or 4 byte RGBA pixels. The bottom left pixel is the first in the string. This format is suitable to pass to `gl.lrectwrite`, for instance.

function of module imgfile: **readscaled** *(file, x, y, filter[, blur])*

This function is identical to read but it returns an image that is scaled to the given x and y sizes. If the filter and blur parameters are omitted scaling is done by simply dropping or duplicating pixels, so the result will be less than perfect, especially for computer-generated images.

Alternatively, you can specify a filter to use to smoothen the image after scaling. The filter forms supported are `'impulse'`, `'box'`, `'triangle'`, `'quadratic'` and `'gaussian'`. If a filter is specified blur is an optional parameter specifying the blurriness of the filter. It defaults to `1.0`.

`readscaled` makes no attempt to keep the aspect ratio correct, so that is the users' responsibility.

function of module imgfile: **ttob** *(flag)*

This function sets a global flag which defines whether the scan lines of the image are read or written from bottom to top (flag is zero, compatible with SGI GL) or from top to bottom(flag is one, compatible with X). The default is zero.

function of module imgfile: **write** *(file, data, x, y, z)*

This function writes the RGB or greyscale data in data to image file file. x and y give the size of the image, z is 1 for 1 byte greyscale images or 3 for RGB images (which are stored as 4 byte values of which only the lower three bytes are used). These are the formats returned by `gl.lrectread`.

Go to the previous, next section.

Go to the previous, next section.

SunOS Specific Services

The modules described in this chapter provide interfaces to features that are unique to the SunOS operating system (versions 4 and 5; the latter is also known as Solaris version 2).

Built-in Module sunaudiodev

This module allows you to access the sun audio interface. The sun audio hardware is capable of recording and playing back audio data in U-LAW format with a sample rate of 8K per second. A full description can be gotten with `man audio'.

The module defines the following variables and functions:

exception: module sunaudiodev **error**

This exception is raised on all errors. The argument is a string describing what went wrong.

function of module sunaudiodev: **open** *(mode)*

This function opens the audio device and returns a sun audio device object. This object can then be used to do I/O on. The mode parameter is one of `'r'` for record-only access, `'w'` for play-only access, `'rw'` for both and `'control'` for access to the control device. Since only one process is allowed to have the recorder or player open at the same time it is a good idea to open the device only for the activity needed. See the audio manpage for details.

Audio Device Objects

The audio device objects are returned by `open` define the following methods (except `control` objects which only provide getinfo, setinfo and drain):

Method: audio device **close** *()*

This method explicitly closes the device. It is useful in situations where deleting the object does not immediately close it since there are other references to it. A closed device should not be used again.

Method: audio device **drain** *()*

This method waits until all pending output is processed and then returns. Calling this method is often not necessary: destroying the object will automatically close the audio device and this will do an implicit drain.

Method: audio device **flush** *()*

This method discards all pending output. It can be used avoid the slow response to a user's stop request (due to buffering of up to one second of sound).

Method: audio device **getinfo** *()*

This method retrieves status information like input and output volume, etc. and returns it in the form of an audio status object. This object has no methods but it contains a number of attributes describing the current device status. The names and meanings of the attributes are described in `/usr/include/sun/audioio.h' and in the audio man page. Member names are slightly different from their C counterparts: a status object is only a single structure. Members of the `play` substructure have `o_' prepended to their name and members of the `record` structure have `i_'. So, the C member `play.sample_rate` is accessed as `o_sample_rate`, `record.gain` as `i_gain` and `monitor_gain` plainly as `monitor_gain`.

Method: audio device **ibufcount** *()*

This method returns the number of samples that are buffered on the recording side, i.e. the program will not block on a `read` call of so many samples.

Method: audio device **obufcount** *()*

This method returns the number of samples buffered on the playback side. Unfortunately, this number cannot be used to determine a number of samples that can be written without blocking since the kernel output queue length seems to be variable.

Method: audio device **read** *(size)*

This method reads size samples from the audio input and returns them as a python string. The function blocks until enough data is available.

Method: audio device **setinfo** *(status)*

This method sets the audio device status parameters. The status parameter is an device status object as returned by `getinfo` and possibly modified by the program.

Method: audio device **write** *(samples)*

Write is passed a python string containing audio samples to be played. If there is enough buffer space free it will immediately return, otherwise it will block.

There is a companion module, `SUNAUDIODEV`, which defines useful symbolic constants like `MIN_GAIN`, `MAX_GAIN`, `SPEAKER`, etc. The names of the constants are the same names as used in the C include file `<sun/audioio.h>', with the leading string `AUDIO_' stripped.

Useability of the control device is limited at the moment, since there is no way to use the "wait for something to happen" feature the device provides.

Go to the previous, next section.

Go to the previous, next section.

Function Index

=

_

a

b

C

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

TOC

S

TOC

t

u

v

W

X

Z

Go to the previous, next section.

Go to the previous, next section.

Variable Index

a

b

c

d

e

f

g

h

i

k

l

m

n

o

p

r

S

t

u

v

w

X

- X11Stylesheet
- XRangeType

Z

- ZeroDivisionError

Go to the previous, next section.

Go to the previous, next section.

Module Index

_

a

c

d

f

- fm (built-in)
- ftplib (standard)

g

- gdbm (built-in)
- getopt (standard)
- gl (built-in)
- GL (standard)
- gopherlib (standard)
- grp (built-in)

h

- htmllib (standard)
- httplib (standard)

i

- imageop (built-in)
- imgfile (built-in)
- imp (built-in)

j

- jpeg (built-in)

m

- mac (built-in)
- macconsole (built-in)
- macdnr (built-in)
- macfs (built-in)
- macpath (standard)
- macspeech (built-in)
- mactcp (built-in)

n

o

p

r

- rotor (built-in)

s

- select (built-in)
- sgmllib (standard)
- shelve (standard)
- signal (built-in)
- socket (built-in)
- stdwin (built-in)
- stdwinevents (standard)
- string (standard)
- strop (built-in)
- struct (built-in)
- sunaudiodev (built-in)
- sys (built-in)

t

- tempfile (standard)
- termios (built-in)
- TERMIOS (standard)
- thread (built-in)
- time (built-in)
- traceback (standard)
- types (standard)

u

- urllib (standard)
- urlparse (standard)

W

Go to the previous, next section.

Go to the previous section.

Concept Index

a

b

c

d

e

f

g

h

i

l

m

n

o

p

r

s

t

u

v

w

Go to the previous section.

Python library reference

(1)

Most descriptions sorely lack explanations of the exceptions that may be raised -- this will be fixed in a future version of this manual.

(2)

As a consequence, the list `[1, 2]` is considered equal to `[1.0, 2.0]`, and similar for tuples.

(3)

They must have since the parser can't tell the type of the operands.

(4)

A tuple object in this case should be a singleton.

(5)

These numbers are fairly arbitrary. They are intended to avoid printing endless strings of meaningless digits without hampering correct use and without having to know the exact precision of floating point values on a particular machine.

(6)

The advantage of leaving the newline on is that an empty string can be returned to mean EOF without being ambiguous. Another advantage is that (in cases where it might matter, e.g. if you want to make an exact copy of a file while scanning its lines) you can tell whether the last line of a file ended in a newline or not (yes this happens!).

(7)

It is used relatively rarely so does not warrant being made into a statement.

(8)

This is ugly -- the language definition should require truncation towards zero.

(9)

Specifying a buffer size currently has no effect on systems that don't have `setvbuf()`. The interface to specify the buffer size is not done using a method that calls `setvbuf()`, because that may dump core when called after any I/O has been performed, and there's no reliable way to determine whether this is the case.

(10)

In the current implementation, local variable bindings cannot normally be affected this way, but variables retrieved from other scopes (e.g. modules) can be. This may change.

(11)

The name of this module stems from a bit of terminology used by the designers of Modula-3 (amongst others), who use the term "marshalling" for shipping of data around in a self-contained form. Strictly speaking, "to marshal" means to convert some data from internal to external form (in an RPC buffer for instance) and "unmarshalling" for the reverse process.

(12)

A solution would be to refuse such literals in the parser, since they are inherently non-portable. Another solution would be to let the `marshal` module raise an exception when an integer value would be truncated. At least one of these solutions will be implemented in a future version.

(13)

The problem with automatically passing on `environ` is that there is no portable way of changing the environment.

(14)

Updated and converted to LaTeX by Guido van Rossum. The references to the old profiler are left in the text, although it no longer exists.

(15)

This was once necessary, when Python would print any unused expression result that was not `None`. The method is still defined for backward compatibility.

(16)

The Python version of STDWIN does not support draw procedures; all drawing requests are reported as draw events.

GNU Readline Library

Edition 2.0, for `Readline Library` Version 2.0.

July 1994

Brian Fox, Free Software Foundation Chet Ramey, Case Western Reserve University

Go to the next section.

This document describes the GNU Readline Library, a utility which aids in the consistency of user interface across discrete programs that need to provide a command line interface.

Published by the Free Software Foundation
675 Massachusetts Avenue,
Cambridge, MA 02139 USA

Command Line Editing

This chapter describes the basic features of the GNU command line editing interface.

Introduction to Line Editing

The following paragraphs describe the notation used to represent keystrokes.

The text C-k is read as `Control-K' and describes the character produced when the Control key is depressed and the k key is struck.

The text M-k is read as `Meta-K' and describes the character produced when the meta key (if you have one) is depressed, and the k key is struck. If you do not have a meta key, the identical keystroke can be generated by typing ESC *first*, and then typing k. Either process is known as metafying the k key.

The text M-C-k is read as `Meta-Control-k' and describes the character produced by metafying C-k.

In addition, several keys have their own names. Specifically, DEL, ESC, LFD, SPC, RET, and TAB all stand for themselves when seen in this text, or in an init file (see section Readline Init File, for more info).

Readline Interaction

Often during an interactive session you type in a long line of text, only to notice that the first word on the line is misspelled. The Readline library gives you a set of commands for manipulating the text as you type it in, allowing you to just fix your typo, and not forcing you to retype the majority of the line. Using these editing commands, you move the cursor to the place that needs correction, and delete or insert the text of the corrections. Then, when you are satisfied with the line, you simply press RETURN. You do not have to be at the end of the line to press RETURN; the entire line is accepted regardless of the location of the cursor within the line.

Readline Bare Essentials

In order to enter characters into the line, simply type them. The typed character appears where the cursor was, and then the cursor moves one space to the right. If you mistype a character, you can use your erase character to back up and delete the mistyped character.

Sometimes you may miss typing a character that you wanted to type, and not notice your error until you have typed several other characters. In that case, you can type C-b to move the cursor to the left, and then correct your mistake. Afterwards, you can move the cursor to the right with C-f.

When you add text in the middle of a line, you will notice that characters to the right of the cursor are `pushed over' to make room for the text that you have inserted. Likewise, when you delete text behind the cursor, characters to the right of the cursor are `pulled back' to fill in the blank space created by the removal of the text. A list of the basic bare essentials for editing the text of an input line follows.

C-b
: Move back one character.

C-f
: Move forward one character.

DEL
: Delete the character to the left of the cursor.

C-d
: Delete the character underneath the cursor.

Printing characters
: Insert the character into the line at the cursor.

C-_
: Undo the last thing that you did. You can undo all the way back to an empty line.

Readline Movement Commands

The above table describes the most basic possible keystrokes that you need in order to do editing of the input line. For your convenience, many other commands have been added in addition to C-b, C-f, C-d, and DEL. Here are some commands for moving more rapidly about the line.

C-a

Move to the start of the line.

C-e

Move to the end of the line.

M-f

Move forward a word.

M-b

Move backward a word.

C-l

Clear the screen, reprinting the current line at the top.

Notice how C-f moves forward a character, while M-f moves forward a word. It is a loose convention that control keystrokes operate on characters while meta keystrokes operate on words.

Readline Killing Commands

Killing text means to delete the text from the line, but to save it away for later use, usually by yanking (re-inserting) it back into the line. If the description for a command says that it `kills' text, then you can be sure that you can get the text back in a different (or the same) place later.

When you use a kill command, the text is saved in a kill-ring. Any number of consecutive kills save all of the killed text together, so that when you yank it back, you get it all. The kill ring is not line specific; the text that you killed on a previously typed line is available to be yanked back later, when you are typing another line.

Here is the list of commands for killing text.

C-k

Kill the text from the current cursor position to the end of the line.

M-d

Kill from the cursor to the end of the current word, or if between words, to the end of the next word.

M-DEL

Kill from the cursor the start of the previous word, or if between words, to the start of the previous word.

C-w

Kill from the cursor to the previous whitespace. This is different than M-DEL because the word boundaries differ.

And, here is how to yank the text back into the line. Yanking means to copy the most-recently-killed text from the kill buffer.

C-y

Yank the most recently killed text back into the buffer at the cursor.

M-y

Rotate the kill-ring, and yank the new top. You can only do this if the prior command is C-y or M-y.

Readline Arguments

You can pass numeric arguments to Readline commands. Sometimes the argument acts as a repeat count, other times it is the *sign* of the argument that is significant. If you pass a negative argument to a command which normally acts in a forward direction, that command will act in a backward direction. For example, to kill text back to the start of the line, you might type M-- C-k.

The general way to pass numeric arguments to a command is to type meta digits before the command. If the first `digit' you type is a minus sign (-), then the sign of the argument will be negative. Once you have typed one meta digit to get the argument started, you can type the remainder of the digits, and then the command. For example, to give the C-d command an argument of 10, you could type M-1 0 C-d.

Readline Init File

Although the Readline library comes with a set of Emacs-like keybindings installed by default, it is possible that you would like to use a different set of keybindings. You can customize programs that use Readline by putting commands in an init file in your home directory. The name of this file is taken from the value of the environment variable `INPUTRC`. If that variable is unset, the default is `` `~/.inputrc' ``.

When a program which uses the Readline library starts up, the init file is read, and the key bindings are set.

In addition, the `C-x C-r` command re-reads this init file, thus incorporating any changes that you might have made to it.

Readline Init Syntax

There are only a few basic constructs allowed in the Readline init file. Blank lines are ignored. Lines beginning with a # are comments. Lines beginning with a $ indicate conditional constructs (see section Conditional Init Constructs). Other lines denote variable settings and key bindings.

Variable Settings

You can change the state of a few variables in Readline by using the `set` command within the init file. Here is how you would specify that you wish to use `vi` line editing commands:

```
set editing-mode vi
```

Right now, there are only a few variables which can be set; so few, in fact, that we just list them here:

`editing-mode`

The `editing-mode` variable controls which editing mode you are using. By default, Readline starts up in Emacs editing mode, where the keystrokes are most similar to Ema This variable can be set to either `emacs` or `vi`.

`horizontal-scroll-mode`

This variable can be set to either `On` or `Off`. Setting it to `On` means that the text of the lines that you edit will scroll horizontally on a single screen line when they are longer than the width of the screen, instead of wrapping onto a new screen line. By default, this variable is set to `Off`.

`mark-modified-lines`

This variable, when set to `On`, says to display an asterisk (`*') at the start of history lines which have been modified. This variable is `off` by default.

`bell-style`

Controls what happens when Readline wants to ring the terminal bell. If set to `none`, Readline never rings the bell. If set to `visible`, Readline uses a visible bell if one is available. If set to `audible` (the default), Readline attempts to ring the terminal's bell.

`comment-begin`

The string to insert at the beginning of the line when the `vi-comment` command is executed. The default value is `"#"`.

`meta-flag`

If set to `on`, Readline will enable eight-bit input (it will not strip the eighth bit from the characters it reads), regardless of what the terminal claims it can support. The default value is `off`.

`convert-meta`

If set to `on`, Readline will convert characters with the eigth bit set to an ASCII key sequence by stripping the eigth bit and prepending an ESC character, converting them to a meta-prefixed key sequence. The default value is `on`.

`output-meta`

If set to `on`, Readline will display characters with the eighth bit set directly rather than as a meta-prefixed escape sequence. The default is `off`.

`completion-query-items`

The number of possible completions that determines when the user is asked whether he wants to see the list of possibilities. If the number of possible completions is greater than this value, Readline will ask the user whether or not he wishes to view them; otherwise, they are simply listed. The default limit is `100`.

`keymap`

Sets Readline's idea of the current keymap for key binding commands. Acceptable `keymap` names are `emacs`, `emacs-standard`, `emacs-meta`, `emacs-ctlx`, `vi`, `vi-move`, `vi-command`, and `vi-insert`. `vi` is equivalent to `vi-command`; `emacs` is equivalent to `emacs-standard`. The default value is `emacs`. The value of the `editing-mode` variable also affects the default keymap.

`show-all-if-ambiguous`

This alters the default behavior of the completion functions. If set to `on`, words which h more than one possible completion cause the matches to be listed immediately instead o ringing the bell. The default value is `off`.

`expand-tilde`

If set to `on`, tilde expansion is performed when Readline attempts word completion. The default is `off`.

- Key Bindings The syntax for controlling key bindings in the init file is simple. First you have to know the name of the command that you want to change. The following pages contain tables of the command name, the default keybinding, and a short description of what the command does.

Once you know the name of the command, simply place the name of the key you wish to bind the command to, a colon, and then the name of the command on a line in the init file. The name of the key can be expressed in different ways, depending on which is most comfortable for you.

keyname: function-name or macro

keyname is the name of a key spelled out in English. For example:

```
Control-u: universal-argument
Meta-Rubout: backward-kill-word
Control-o: ">&output"
```

In the above example, `C-u' is bound to the function `universal-argument`, and `C-o' is bound to run the macro expressed on the right hand side (that is, to insert the text `>&output' into the line).

"keyseq": function-name or macro

keyseq differs from keyname above in that strings denoting an entire key sequence can be specified, by placing the key sequence in double quotes. Some GNU Emacs style key escapes can be used, as in the following example, but the special character names are not recognized.

```
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
"\e[11~": "Function Key 1"
```

In the above example, `C-u' is bound to the function `universal-argument` (just as it was in the first example), `C-x C-r' is bound to the function `re-read-init-file`, and `ESC [ 1 1 ~' is bound to insert the text `Function Key 1'. The following escape sequences are available when specifying key sequences:

`\C-`

control prefix

`\M-`

meta prefix

`\e`

an escape character

`\\`
: backslash

`\"`
: "

`\'`
: '

When entering the text of a macro, single or double quotes should be used to indicate a macro definition. Unquoted text is assumed to be a function name. Backslash will quote any character in the macro text, including " and '. For example, the following binding will make C-x \ insert a single \ into the line:

```
"\C-x\\": "\\"
```

Conditional Init Constructs

Readline implements a facility similar in spirit to the conditional compilation features of the C preprocessor which allows key bindings and variable settings to be performed as the result of tests. There are three parser directives used.

`$if`
: The `$if` construct allows bindings to be made based on the editing mode, the terminal being used, or the application using Readline. The text of the test extends to the end of the line; no characters are required to isolate it.

  `mode`
  : The `mode=` form of the `$if` directive is used to test whether Readline is in `emacs` or `vi` mode. This may be used in conjunction with the `set keymap' command, for instance, to set bindings in the `emacs-standard` and `emacs-ctlx` keymaps only if Readline is starting out in `emacs` mode.

  `term`
  : The `term=` form may be used to include terminal-specific key bindings, perhaps to bind the key sequences output by the terminal's function keys. The word on the right side of the `=' is tested against the full name of the terminal and the portion of the terminal name before the first `-'. This allows sun to match both sun and sun-cmd, for instance.

  `application`
  : The application construct is used to include application-specific settings. Each program using the Readline library sets the application name, and you can test for it. This could be used to bind key sequences to functions useful for a specific program. For instance, the following command adds a key sequence that quotes the current or previous word in Bash:

```
$if bash
# Quote the current or previous word
"\C-xq": "\eb\"\ef\""
```


```
$endif
```

- $endif This command, as you saw in the previous example, terminates an `$if` command.
- $else Commands in this branch of the `$if` directive are executed if the test fails.

Bindable Readline Commands

Commands For Moving

`beginning-of-line (C-a)`
Move to the start of the current line.

`end-of-line (C-e)`
Move to the end of the line.

`forward-char (C-f)`
Move forward a character.

`backward-char (C-b)`
Move back a character.

`forward-word (M-f)`
Move forward to the end of the next word. Words are composed of letters and digits.

`backward-word (M-b)`
Move back to the start of this, or the previous, word. Words are composed of letters and digits.

`clear-screen (C-l)`
Clear the screen and redraw the current line, leaving the current line at the top of the screen.

`redraw-current-line ()`
Refresh the current line. By default, this is unbound.

Commands For Manipulating The History

`accept-line (Newline, Return)`
Accept the line regardless of where the cursor is. If this line is non-empty, add it to the history list. If this line was a history line, then restore the history line to its original state.

`previous-history (C-p)`
Move `up' through the history list.

`next-history (C-n)`
Move `down' through the history list.

`beginning-of-history (M-<)`
Move to the first line in the history.

`end-of-history (M->)`
Move to the end of the input history, i.e., the line you are entering.

`reverse-search-history (C-r)`

Search backward starting at the current line and moving `up' through the history as necessary. This is an incremental search.

`forward-search-history (C-s)`

Search forward starting at the current line and moving `down' through the the history as necessary. This is an incremental search.

`non-incremental-reverse-search-history (M-p)`

Search backward starting at the current line and moving `up' through the history as necessary using a non-incremental search for a string supplied by the user.

`non-incremental-forward-search-history (M-n)`

Search forward starting at the current line and moving `down' through the the history as necessary using a non-incremental search for a string supplied by the user.

`history-search-forward ()`

Search forward through the history for the string of characters between the start of the current line and the current point. This is a non-incremental search. By default, this command is unbound.

`history-search-backward ()`

Search backward through the history for the string of characters between the start of the current line and the current point. This is a non-incremental search. By default, this command is unbound.

`yank-nth-arg (M-C-y)`

Insert the first argument to the previous command (usually the second word on the previous line). With an argument n, insert the nth word from the previous command (the words in the previous command begin with word 0). A negative argument inserts the nth word from the end of the previous command.

`yank-last-arg (M-., M-_)`

Insert last argument to the previous command (the last word on the previous line). With an argument, behave exactly like `yank-nth-arg`.

Commands For Changing Text

`delete-char (C-d)`

Delete the character under the cursor. If the cursor is at the beginning of the line, there are no characters in the line, and the last character typed was not C-d, then return EOF.

`backward-delete-char (Rubout)`

Delete the character behind the cursor. A numeric arg says to kill the characters instead of deleting them.

`quoted-insert (C-q, C-v)`

Add the next character that you type to the line verbatim. This is how to insert key sequences like C-q, for example.

`tab-insert (M-TAB)`

Insert a tab character.

`self-insert (a, b, A, 1, !, ...)`

Insert yourself.

`transpose-chars (C-t)`

Drag the character before the cursor forward over the character at the cursor, moving the cursor forward as well. If the insertion point is at the end of the line, then this transposes the last two characters of the line. Negative argumentss don't work.

`transpose-words (M-t)`

Drag the word behind the cursor past the word in front of the cursor moving the cursor over that word as well.

`upcase-word (M-u)`

Uppercase the current (or following) word. With a negative argument, do the previous word, but do not move the cursor.

`downcase-word (M-l)`

Lowercase the current (or following) word. With a negative argument, do the previous word, but do not move the cursor.

`capitalize-word (M-c)`

Capitalize the current (or following) word. With a negative argument, do the previous word, but do not move the cursor.

Killing And Yanking

`kill-line (C-k)`

Kill the text from the current cursor position to the end of the line.

`backward-kill-line (C-x Rubout)`

Kill backward to the beginning of the line.

`unix-line-discard (C-u)`

Kill backward from the cursor to the beginning of the current line. Save the killed text on the kill-ring.

`kill-whole-line ()`

Kill all characters on the current line, no matter where the cursor is. By default, this is unbound.

`kill-word (M-d)`

Kill from the cursor to the end of the current word, or if between words, to the end of the next word. Word boundaries are the same as `forward-word`.

`backward-kill-word (M-DEL)`

Kill the word behind the cursor. Word boundaries are the same as `backward-word`.

`unix-word-rubout (C-w)`

Kill the word behind the cursor, using white space as a word boundary. The killed text is saved on the kill-ring.

`delete-horizontal-space ()`

Delete all spaces and tabs around point. By default, this is unbound.

`yank (C-y)`

Yank the top of the kill ring into the buffer at the current cursor position.

`yank-pop (M-y)`

Rotate the kill-ring, and yank the new top. You can only do this if the prior command is yank or yank-pop.

Specifying Numeric Arguments

`digit-argument (M-0, M-1, ... M--)`

Add this digit to the argument already accumulating, or start a new argument. M-- starts a negative argument.

`universal-argument ()`

Each time this is executed, the argument count is multiplied by four. The argument count is initially one, so executing this function the first time makes the argument count four. By default, this is not bound to a key.

Letting Readline Type For You

`complete (TAB)`

Attempt to do completion on the text before the cursor. This is application-specific. Generally, if you are typing a filename argument, you can do filename completion; if you are typing a command, you can do command completion, if you are typing in a symbol to GDB, you can do symbol name completion, if you are typing in a variable to Bash, you can do variable name completion, and so on.

`possible-completions (M-?)`

List the possible completions of the text before the cursor.

`insert-completions ()`

Insert all completions of the text before point that would have been generated by `possible-completions`. By default, this is not bound to a key.

Keyboard Macros

`start-kbd-macro (C-x ()`

Begin saving the characters typed into the current keyboard macro.

`end-kbd-macro (C-x ))`

Stop saving the characters typed into the current keyboard macro and save the definition.

`call-last-kbd-macro (C-x e)`

Re-execute the last keyboard macro defined, by making the characters in the macro appear as if typed at the keyboard.

Some Miscellaneous Commands

`re-read-init-file (C-x C-r)`

Read in the contents of your init file, and incorporate any bindings or variable assignments found there.

`abort (C-g)`

Abort the current editing command and ring the terminal's bell (subject to the setting of `bell-style`).

`do-uppercase-version (M-a, M-b, ...)`

Run the command that is bound to the corresoponding uppercase character.

`prefix-meta (ESC)`

Make the next character that you type be metafied. This is for people without a meta key. Typing `ESC f' is equivalent to typing `M-f'.

`undo (C-_, C-x C-u)`

Incremental undo, separately remembered for each line.

`revert-line (M-r)`

Undo all changes made to this line. This is like typing the `undo` command enough times to get back to the beginning.

`tilde-expand (M-~)`

Perform tilde expansion on the current word.

`dump-functions ()`

Print all of the functions and their key bindings to the readline output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an inputrc file.

Readline vi Mode

While the Readline library does not have a full set of `vi` editing functions, it does contain enough to allow simple editing of the line. The Readline `vi` mode behaves as specified in the Posix 1003.2 standard.

In order to switch interactively between `Emacs` and `Vi` editing modes, use the command M-C-j (toggle-editing-mode). The Readline default is `emacs` mode.

When you enter a line in `vi` mode, you are already placed in `insertion' mode, as if you had typed an `i'. Pressing ESC switches you into `command' mode, where you can edit the text of the line with the standard `vi` movement keys, move to previous history lines with `k', and following lines with `j', and so forth.

Go to the next section.

Go to the previous, next section.

Programming with GNU Readline

This chapter describes the interface between the GNU Readline Library and other programs. If you are a programmer, and you wish to include the features found in GNU Readline such as completion, line editing, and interactive history manipulation in your own programs, this section is for you.

Basic Behavior

Many programs provide a command line interface, such as `mail`, `ftp`, and `sh`. For such programs, the default behaviour of Readline is sufficient. This section describes how to use Readline in the simplest way possible, perhaps to replace calls in your code to `gets()` or `fgets ()`.

The function `readline ()` prints a prompt and then reads and returns a single line of text from the user. The line `readline` returns is allocated with `malloc ()`; you should `free ()` the line when you are done with it. The declaration for `readline` in ANSI C is

```
char *readline (char *prompt);
```

So, one might say

```
char *line = readline ("Enter a line: ");
```

in order to read a line of text from the user. The line returned has the final newline removed, so only the text remains.

If `readline` encounters an `EOF` while reading the line, and the line is empty at that point, then `(char *)NULL` is returned. Otherwise, the line is ended just as if a newline had been typed.

If you want the user to be able to get at the line later, (with C-p for example), you must call `add_history ()` to save the line away in a history list of such lines.

```
add_history (line);
```

For full details on the GNU History Library, see the associated manual.

It is preferable to avoid saving empty lines on the history list, since users rarely have a burning need to reuse a blank line. Here is a function which usefully replaces the standard `gets ()` library function, and has the advantage of no static buffer to overflow:

```
/* A static variable for holding the line. */
static char *line_read = (char *)NULL;

/* Read a string, and return a pointer to it.  Returns NULL on EOF. */
char *
rl_gets ()
{
  /* If the buffer has already been allocated, return the memory
```


```
    to the free pool. */
  if (line_read)
    {
      free (line_read);
      line_read = (char *)NULL;
    }

  /* Get a line from the user. */
  line_read = readline ("");

  /* If the line has any text in it, save it on the history. */
  if (line_read && *line_read)
    add_history (line_read);

  return (line_read);
}
```

This function gives the user the default behaviour of TAB completion: completion on file names. If you do not want Readline to complete on filenames, you can change the binding of the TAB key with `rl_bind_key ()`.

```
int rl_bind_key (int key, int (*function)());
```

`rl_bind_key ()` takes two arguments: key is the character that you want to bind, and function is the address of the function to call when key is pressed. Binding TAB to `rl_insert ()` makes TAB insert itself. `rl_bind_key ()` returns non-zero if key is not a valid ASCII character code (between 0 and 255).

Thus, to disable the default TAB behavior, the following suffices:

```
rl_bind_key ('\t', rl_insert);
```

This code should be executed once at the start of your program; you might write a function called `initialize_readline ()` which performs this and other desired initializations, such as installing custom completers (see section Custom Completers).

Custom Functions

Readline provides many functions for manipulating the text of the line, but it isn't possible to anticipate the needs of all programs. This section describes the various functions and variables defined within the Readline library which allow a user program to add customized functionality to Readline.

The Function Type

For readabilty, we declare a new type of object, called Function. A `Function` is a C function which returns an `int`. The type declaration for `Function` is:

```
typedef int Function ();
```

The reason for declaring this new type is to make it easier to write code describing pointers to C functions. Let us say we had a variable called func which was a pointer to a function. Instead of the classic C declaration

```
int (*)()func;
```

we may write

```
Function *func;
```

Similarly, there are

```
typedef void VFunction ();
typedef char *CPFunction (); and
typedef char **CPPFunction ();
```

for functions returning no value, `pointer to char`, and `pointer to pointer to char`, respectively.

Writing a New Function

In order to write new functions for Readline, you need to know the calling conventions for keyboard-invoked functions, and the names of the variables that describe the current state of the line read so far.

The calling sequence for a command `foo` looks like

```
foo (int count, int key)
```

where count is the numeric argument (or 1 if defaulted) and key is the key that invoked this function.

It is completely up to the function as to what should be done with the numeric argument. Some functions use it as a repeat count, some as a flag, and others to choose alternate behavior (refreshing the current line as opposed to refreshing the screen, for example). Some choose to ignore it. In general, if a function uses the numeric argument as a repeat count, it should be able to do something useful with both negative and positive arguments. At the very least, it should be aware that it can be passed a negative argument.

Variable: char * **rl_line_buffer**

This is the line gathered so far. You are welcome to modify the contents of the line, but see section Allowing Undoing.

Variable: int **rl_point**

The offset of the current cursor position in `rl_line_buffer` (the *point*).

Variable: int **rl_end**

The number of characters present in `rl_line_buffer`. When `rl_point` is at the end of the line, `rl_point` and `rl_end` are equal.

Variable: int **rl_mark**

The mark (saved position) in the current line. If set, the mark and point define a *region*.

Variable: int **rl_done**

Setting this to a non-zero value causes Readline to return the current line immediately.

Variable: int **rl_pending_input**

Setting this to a value makes it the next keystroke read. This is a way to stuff a single character into the input stream.

Variable: char * **rl_prompt**

The prompt Readline uses. This is set from the argument to `readline ()`, and should not be assigned to directly.

Variable: char * **rl_terminal_name**

The terminal type, used for initialization.

Variable: char * **rl_readline_name**

This variable is set to a unique name by each application using Readline. The value allows conditional parsing of the inputrc file (see section Conditional Init Constructs).

Variable: FILE * **rl_instream**

The stdio stream from which Readline reads input.

Variable: FILE * **rl_outstream**

The stdio stream to which Readline performs output.

Variable: Function * **rl_startup_hook**

If non-zero, this is the address of a function to call just before `readline` prints the first prompt.

Readline Convenience Functions

Naming a Function

The user can dynamically change the bindings of keys while using Readline. This is done by representing the function with a descriptive name. The user is able to type the descriptive name when referring to the function. Thus, in an init file, one might find

```
Meta-Rubout:    backward-kill-word
```

This binds the keystroke Meta-Rubout to the function *descriptively* named `backward-kill-word`. You, as the programmer, should bind the functions you write to descriptive names as well. Readline provides a function for doing that:

Function: int **rl_add_defun** *(char *name, Function *function, int key)*

Add name to the list of named functions. Make function be the function that gets called. If key is not -1, then bind it to function using `rl_bind_key ()`.

Using this function alone is sufficient for most applications. It is the recommended way to add a few functions to the default functions that Readline has built in. If you need to do something other than adding a function to Readline, you may need to use the underlying functions described below.

Selecting a Keymap

Key bindings take place on a keymap. The keymap is the association between the keys that the user types and the functions that get run. You can make your own keymaps, copy existing keymaps, and tell Readline which keymap to use.

Function: Keymap **rl_make_bare_keymap** *()*

Returns a new, empty keymap. The space for the keymap is allocated with `malloc ()`; you should `free ()` it when you are done.

Function: Keymap **rl_copy_keymap** *(Keymap map)*

Return a new keymap which is a copy of map.

Function: Keymap **rl_make_keymap** *()*

Return a new keymap with the printing characters bound to rl_insert, the lowercase Meta characters bound to run their equivalents, and the Meta digits bound to produce numeric arguments.

Function: void **rl_discard_keymap** *(Keymap keymap)*

Free the storage associated with keymap.

Readline has several internal keymaps. These functions allow you to change which keymap is active.

Function: Keymap **rl_get_keymap** *()*

Returns the currently active keymap.

Function: void **rl_set_keymap** *(Keymap keymap)*

Makes keymap the currently active keymap.

Function: Keymap **rl_get_keymap_by_name** *(char *name)*

Return the keymap matching name. name is one which would be supplied in a `set keymap` inputrc line (see section Readline Init File).

Binding Keys

You associate keys with functions through the keymap. Readline has several internal keymaps: `emacs_standard_keymap`, `emacs_meta_keymap`, `emacs_ctlx_keymap`, `vi_movement_keymap`, and `vi_insertion_keymap`. `emacs_standard_keymap` is the default, and the examples in this manual assume that.

These functions manage key bindings.

Function: int **rl_bind_key** *(int key, Function *function)*

Binds key to function in the currently active keymap. Returns non-zero in the case of an invalid key.

Function: int **rl_bind_key_in_map** *(int key, Function *function, Keymap map)*

Bind key to function in map. Returns non-zero in the case of an invalid key.

Function: int **rl_unbind_key** *(int key)*

Bind key to the null function in the currently active keymap. Returns non-zero in case of error.

Function: int **rl_unbind_key_in_map** *(int key, Keymap map)*

Bind key to the null function in map. Returns non-zero in case of error.

Function: int **rl_generic_bind** *(int type, char *keyseq, char *data, Keymap map)*

Bind the key sequence represented by the string keyseq to the arbitrary pointer data. type says what kind of data is pointed to by data; this can be a function (`ISFUNC`), a macro (`ISMACR`), or a keymap (`ISKMAP`). This makes new keymaps as necessary. The initial keymap in which to do bindings is map.

Function: int **rl_parse_and_bind** *(char *line)*

Parse line as if it had been read from the `inputrc` file and perform any key bindings and variable assignments found (see section Readline Init File).

Associating Function Names and Bindings

These functions allow you to find out what keys invoke named functions and the functions invoked by a particular key sequence.

Function: Function * **rl_named_function** *(char *name)*

Return the function with name name.

Function: Function * **rl_function_of_keyseq** *(char *keyseq, Keymap map, int *type)*

Return the function invoked by keyseq in keymap map. If map is NULL, the current keymap is used. If type is not NULL, the type of the object is returned in it (one of `ISFUNC`, `ISKMAP`, or `ISMACR`).

Function: char ** **rl_invoking_keyseqs** *(Function *function)*

Return an array of strings representing the key sequences used to invoke function in the current keymap.

Function: char ** **rl_invoking_keyseqs_in_map** *(Function *function, Keymap map)*

Return an array of strings representing the key sequences used to invoke function in the keymap map.

Allowing Undoing

Supporting the undo command is a painless thing, and makes your functions much more useful. It is certainly easy to try something if you know you can undo it. I could use an undo function for the stock market.

If your function simply inserts text once, or deletes text once, and uses `rl_insert_text ()` or `rl_delete_text ()` to do it, then undoing is already done for you automatically.

If you do multiple insertions or multiple deletions, or any combination of these operations, you should group them together into one operation. This is done with `rl_begin_undo_group ()` and `rl_end_undo_group ()`.

The types of events that can be undone are:

```
enum undo_code { UNDO_DELETE, UNDO_INSERT, UNDO_BEGIN, UNDO_END };
```

Notice that `UNDO_DELETE` means to insert some text, and `UNDO_INSERT` means to delete some text. That is, the undo code tells undo what to undo, not how to undo it. `UNDO_BEGIN` and `UNDO_END` are tags added by `rl_begin_undo_group ()` and `rl_end_undo_group ()`.

Function: int **rl_begin_undo_group** *()*

Begins saving undo information in a group construct. The undo information usually comes from calls to

`rl_insert_text ()` and `rl_delete_text ()`, but could be the result of calls to `rl_add_undo ()`.

Function: int **rl_end_undo_group** *()*

Closes the current undo group started with `rl_begin_undo_group ()`. There should be one call to `rl_end_undo_group ()` for each call to `rl_begin_undo_group ()`.

Function: void **rl_add_undo** *(enum undo_code what, int start, int end, char *text)*

Remember how to undo an event (according to what). The affected text runs from start to end, and encompasses text.

Function: void **free_undo_list** *()*

Free the existing undo list.

Function: int **rl_do_undo** *()*

Undo the first thing on the undo list. Returns `0` if there was nothing to undo, non-zero if something was undone.

Finally, if you neither insert nor delete text, but directly modify the existing text (e.g., change its case), call `rl_modifying ()` once, just before you modify the text. You must supply the indices of the text range that you are going to modify.

Function: int **rl_modifying** *(int start, int end)*

Tell Readline to save the text between start and end as a single undo unit. It is assumed that you will subsequently modify that text.

Redisplay

Function: int **rl_redisplay** *()*

Change what's displayed on the screen to reflect the current contents of `rl_line_buffer`.

Function: int **rl_forced_update_display** *()*

Force the line to be updated and redisplayed, whether or not Readline thinks the screen display is correct.

Function: int **rl_on_new_line** *()*

Tell the update routines that we have moved onto a new (empty) line, usually after ouputting a newline.

Function: int **rl_reset_line_state** *()*

Reset the display state to a clean state and redisplay the current line starting on a new line.

Function: int **rl_message** *(va_alist)*

The arguments are a string as would be supplied to `printf`. The resulting string is displayed in the echo area. The echo area is also used to display numeric arguments and search strings.

Function: int **rl_clear_message** *()*

Clear the message in the echo area.

Modifying Text

Function: int **rl_insert_text** *(char *text)*

Insert text into the line at the current cursor position.

Function: int **rl_delete_text** *(int start, int end)*

Delete the text between start and end in the current line.

Function: char * **rl_copy_text** *(int start, int end)*

Return a copy of the text between start and end in the current line.

Function: int **rl_kill_text** *(int start, int end)*

Copy the text between start and end in the current line to the kill ring, appending or prepending to the last kill if the last command was a kill command. The text is deleted. If start is less than end, the text is appended, otherwise prepended. If the last command was not a kill, a new kill ring slot is used.

Utility Functions

Function: int **rl_reset_terminal** *(char *terminal_name)*

Reinitialize Readline's idea of the terminal settings using terminal_name as the terminal type (e.g., `vt100`).

Function: int **alphabetic** *(int c)*

Return 1 if c is an alphabetic character.

Function: int **numeric** *(int c)*

Return 1 if c is a numeric character.

Function: int **ding** *()*

Ring the terminal bell, obeying the setting of `bell-style`.

The following are implemented as macros, defined in `chartypes.h`.

Function: int **uppercase_p** *(int c)*

Return 1 if c is an uppercase alphabetic character.

Function: int **lowercase_p** *(int c)*

Return 1 if c is a lowercase alphabetic character.

Function: int **digit_p** *(int c)*

Return 1 if c is a numeric character. Function:

Function: int **to_upper** *(int c)*

If c is a lowercase alphabetic character, return the corresponding uppercase character.

Function: int **to_lower** *(int c)*

If c is an uppercase alphabetic character, return the corresponding lowercase character.

Function: int **digit_value** *(int c)*

If c is a number, return the value it represents.

An Example

Here is a function which changes lowercase characters to their uppercase equivalents, and uppercase characters to lowercase. If this function was bound to `M-c', then typing `M-c' would change the case of the character under point. Typing `M-1 0 M-c' would change the case of the following 10 characters, leaving the cursor on the last character changed.

```
/* Invert the case of the COUNT following characters. */
int
invert_case_line (count, key)
     int count, key;
{
  register int start, end, i;

  start = rl_point;

  if (rl_point >= rl_end)
    return (0);

  if (count < 0)
    {
      direction = -1;
      count = -count;
    }
  else
    direction = 1;

  /* Find the end of the range to modify. */
  end = start + (count * direction);

  /* Force it to be within range. */
  if (end > rl_end)
    end = rl_end;
  else if (end < 0)
    end = 0;

  if (start == end)
    return (0);

  if (start > end)
    {
      int temp = start;
      start = end;
      end = temp;
    }
```


```
  /* Tell readline that we are modifying the line, so it will save
     the undo information. */
  rl_modifying (start, end);

  for (i = start; i != end; i++)
    {
      if (uppercase_p (rl_line_buffer[i]))
        rl_line_buffer[i] = to_lower (rl_line_buffer[i]);
      else if (lowercase_p (rl_line_buffer[i]))
        rl_line_buffer[i] = to_upper (rl_line_buffer[i]);
    }
  /* Move point to on top of the last character changed. */
  rl_point = (direction == 1) ? end - 1 : start;
  return (0);
}
```

Custom Completers

Typically, a program that reads commands from the user has a way of disambiguating commands and data. If your program is one of these, then it can provide completion for commands, data, or both. The following sections describe how your program and Readline cooperate to provide this service.

How Completing Works

In order to complete some text, the full list of possible completions must be available. That is, it is not possible to accurately expand a partial word without knowing all of the possible words which make sense in that context. The Readline library provides the user interface to completion, and two of the most common completion functions: filename and username. For completing other types of text, you must write your own completion function. This section describes exactly what such functions must do, and provides an example.

There are three major functions used to perform completion:

1. The user-interface function `rl_complete ()`. This function is called with the same arguments as other Readline functions intended for interactive use: count and invoking_key. It isolates the word to be completed and calls `completion_matches ()` to generate a list of possible completions. It then either lists the possible completions, inserts the possible completions, or actually performs the completion, depending on which behavior is desired.
2. The internal function `completion_matches ()` uses your generator function to generate the list of possible matches, and then returns the array of these matches. You should place the address of your generator function in `rl_completion_entry_function`.
3. The generator function is called repeatedly from `completion_matches ()`, returning a string each time. The arguments to the generator function are text and state. text is the partial word to be completed. state is zero the first time the function is called, allowing the generator to perform any necessary initialization, and a positive non-zero integer for each subsequent call. When the generator function returns `(char *)NULL` this signals `completion_matches ()` that there are no more possibilities left. Usually the generator function computes the list of possible completions when state is zero, and returns them one at a time on subsequent calls. Each string the generator function returns as a match must be allocated with `malloc()`; Readline frees the strings when it has finished with them.

Function: int **rl_complete** *(int ignore, int invoking_key)*

Complete the word at or before point. You have supplied the function that does the initial simple matching selection algorithm (see `completion_matches ()`). The default is to do filename completion.

Variable: Function * **rl_completion_entry_function**

This is a pointer to the generator function for `completion_matches ()`. If the value of `rl_completion_entry_function` is `(Function *)NULL` then the default filename generator function, `filename_entry_function ()`, is used.

Completion Functions

Here is the complete list of callable completion functions present in Readline.

Function: int **rl_complete_internal** *(int what_to_do)*

Complete the word at or before point. what_to_do says what to do with the completion. A value of `?' means list the possible completions. `TAB' means do standard completion. `*' means insert all of the possible completions. `!' means to display all of the possible completions, if there is more than one, as well as performing partial completion.

Function: int **rl_complete** *(int ignore, int invoking_key)*

Complete the word at or before point. You have supplied the function that does the initial simple matching selection algorithm (see `completion_matches ()` and `rl_completion_entry_function`). The default is to do filename completion. This calls `rl_complete_internal ()` with an argument depending on invoking_key.

Function: int **rl_possible_completions** *(int count, int invoking_key))*

List the possible completions. See description of `rl_complete ()`. This calls `rl_complete_internal ()` with an argument of `?'.

Function: int **rl_insert_completions** *(int count, int invoking_key))*

Insert the list of possible completions into the line, deleting the partially-completed word. See description of `rl_complete ()`. This calls `rl_complete_internal ()` with an argument of `*'.

Function: char ** **completion_matches** *(char *text, CPFunction *entry_func)*

Returns an array of `(char *)` which is a list of completions for text. If there are no completions, returns `(char **)NULL`. The first entry in the returned array is the substitution for text. The remaining entries are the possible completions. The array is terminated with a `NULL` pointer.

entry_func is a function of two args, and returns a `(char *)`. The first argument is text. The second is a state argument; it is zero on the first call, and non-zero on subsequent calls. entry_func returns a `NULL` pointer to the caller when there are no more matches.

Function: char * **filename_completion_function** *(char *text, int state)*

A generator function for filename completion in the general case. Note that completion in Bash is a little different because of all the pathnames that must be followed when looking up completions for a command. The Bash source is a useful reference for writing custom completion functions.

Function: char * **username_completion_function** *(char *text, int state)*

A completion generator for usernames. text contains a partial username preceded by a random character (usua
`~'). As with all completion generators, state is zero on the first call and non-zero for subsequent calls.

Completion Variables

Variable: Function * **rl_completion_entry_function**

A pointer to the generator function for `completion_matches ()`. `NULL` means to use `filename_entry_function ()`, the default filename completer.

Variable: CPPFunction * **rl_attempted_completion_function**

A pointer to an alternative function to create matches. The function is called with text, start, and end. start and end are indices in `rl_line_buffer` saying what the boundaries of text are. If this function exists and returns `NULL`, or if this variable is set to `NULL`, then `rl_complete ()` will call the value of `rl_completion_entry_function` to generate matches, otherwise the array of strings returned will be used.

Variable: int **rl_completion_query_items**

Up to this many items will be displayed in response to a possible-completions call. After that, we ask the user if she is sure she wants to see them all. The default value is 100.

Variable: char * **rl_basic_word_break_characters**

The basic list of characters that signal a break between words for the completer routine. The default value of this variable is the characters which break words for completion in Bash, i.e., `" \t\n\"\\'`@$><=;|&{(`".

Variable: char * **rl_completer_word_break_characters**

The list of characters that signal a break between words for `rl_complete_internal ()`. The default list is the value of `rl_basic_word_break_characters`.

Variable: char * **rl_special_prefixes**

The list of characters that are word break characters, but should be left in text when it is passed to the completion function. Programs can use this to help determine what kind of completing to do. For instance, Bash sets this variable to "$@" so that it can complete shell variables and hostnames.

Variable: int **rl_ignore_completion_duplicates**

If non-zero, then disallow duplicates in the matches. Default is 1.

Variable: int **rl_filename_completion_desired**

Non-zero means that the results of the matches are to be treated as filenames. This is *always* zero on entry, and can only be changed within a completion entry generator function. If it is set to a non-zero value, directory names have a slash appended and Readline attempts to quote completed filenames if they contain any embedded word break characters.

Variable: int **rl_filename_quoting_desired**

Non-zero means that the results of the matches are to be quoted using double quotes (or an application-specific quoting mechanism) if the completed filename contains any characters in `rl_completer_word_break_chars`. This is *always* non-zero on entry, and can only be changed within a completion entry generator function.

Variable: Function * **rl_ignore_some_completions_function**

This function, if defined, is called by the completer when real filename completion is done, after all the matching names have been generated. It is passed a `NULL` terminated array of matches. The first element (`matches[0]`) is the maximal substring common to all matches. This function can re-arrange the list of matches as required, but each element deleted from the array must be freed.

Variable: char * **rl_completer_quote_characters**

List of characters which can be used to quote a substring of the line. Completion occurs on the entire substring, and within the substring `rl_completer_word_break_characters` are treated as any other character, unless they also appear within this list.

A Short Completion Example

Here is a small application demonstrating the use of the GNU Readline library. It is called `fileman`, and the source code resides in `` `examples/fileman.c' ``. This sample application provides completion of command names, line editing features, and access to the history list.

```
/* fileman.c -- A tiny application which demonstrates how to use the
   GNU Readline library.  This application interactively allows users
   to manipulate files and their modes. */

#include <stdio.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/stat.h>
#include <sys/errno.h>

#include <readline/readline.h>
#include <readline/history.h>

extern char *getwd ();
extern char *xmalloc ();

/* The names of functions that actually do the manipulation. */
int com_list (), com_view (), com_rename (), com_stat (), com_pwd ();
int com_delete (), com_help (), com_cd (), com_quit ();

/* A structure which contains information on the commands this program
   can understand. */

typedef struct {
  char *name;                   /* User printable name of the function. */
  Function *func;               /* Function to call to do the job. */
  char *doc;                    /* Documentation for this function.  */
} COMMAND;

COMMAND commands[] = {
  { "cd", com_cd, "Change to directory DIR" },
```


```
  { "delete", com_delete, "Delete FILE" },
  { "help", com_help, "Display this text" },
  { "?", com_help, "Synonym for `help'" },
  { "list", com_list, "List files in DIR" },
  { "ls", com_list, "Synonym for `list'" },
  { "pwd", com_pwd, "Print the current working directory" },
  { "quit", com_quit, "Quit using Fileman" },
  { "rename", com_rename, "Rename FILE to NEWNAME" },
  { "stat", com_stat, "Print out statistics on FILE" },
  { "view", com_view, "View the contents of FILE" },
  { (char *)NULL, (Function *)NULL, (char *)NULL }
};

/* Forward declarations. */
char *stripwhite ();
COMMAND *find_command ();

/* The name of this program, as taken from argv[0]. */
char *progname;

/* When non-zero, this global means the user is done using this program. */
int done;

char *
dupstr (s)
     int s;
{
  char *r;

  r = xmalloc (strlen (s) + 1);
  strcpy (r, s);
  return (r);
}

main (argc, argv)
     int argc;
     char **argv;
{
  char *line, *s;

  progname = argv[0];

  initialize_readline ();       /* Bind our completer. */

  /* Loop reading and executing lines until the user quits. */
  for ( ; done == 0; )
    {
      line = readline ("FileMan: ");

      if (!line)
```


```
      break;

    /* Remove leading and trailing whitespace from the line.
       Then, if there is anything left, add it to the history list
       and execute it. */
    s = stripwhite (line);

    if (*s)
      {
        add_history (s);
        execute_line (s);
      }

    free (line);
  }
  exit (0);
}

/* Execute a command line. */
int
execute_line (line)
     char *line;
{
  register int i;
  COMMAND *command;
  char *word;

  /* Isolate the command word. */
  i = 0;
  while (line[i] && whitespace (line[i]))
    i++;
  word = line + i;

  while (line[i] && !whitespace (line[i]))
    i++;

  if (line[i])
    line[i++] = '\0';

  command = find_command (word);

  if (!command)
    {
      fprintf (stderr, "%s: No such command for FileMan.\n", word);
      return (-1);
    }

  /* Get argument to command, if any. */
  while (whitespace (line[i]))
    i++;
```


```
  word = line + i;

  /* Call the function. */
  return ((*(command->func)) (word));
}

/* Look up NAME as the name of a command, and return a pointer to that
   command.  Return a NULL pointer if NAME isn't a command name. */
COMMAND *
find_command (name)
     char *name;
{
  register int i;

  for (i = 0; commands[i].name; i++)
    if (strcmp (name, commands[i].name) == 0)
      return (&commands[i]);

  return ((COMMAND *)NULL);
}

/* Strip whitespace from the start and end of STRING.  Return a pointer
   into STRING. */
char *
stripwhite (string)
     char *string;
{
  register char *s, *t;

  for (s = string; whitespace (*s); s++)
    ;

  if (*s == 0)
    return (s);

  t = s + strlen (s) - 1;
  while (t > s && whitespace (*t))
    t--;
  *++t = '\0';

  return s;
}

/* ********************************************************************** */
/*                                                                        */
/*                  Interface to Readline Completion                      */
/*                                                                        */
/* ********************************************************************** */
```


```
char *command_generator ();
char **fileman_completion ();

/* Tell the GNU Readline library how to complete.  We want to try to complete
   on command names if this is the first word in the line, or on filenames
   if not. */
initialize_readline ()
{
  /* Allow conditional parsing of the ~/.inputrc file. */
  rl_readline_name = "FileMan";

  /* Tell the completer that we want a crack first. */
  rl_attempted_completion_function = (CPPFunction *)fileman_completion;
}

/* Attempt to complete on the contents of TEXT.  START and END show the
   region of TEXT that contains the word to complete.  We can use the
   entire line in case we want to do some simple parsing.  Return the
   array of matches, or NULL if there aren't any. */
char **
fileman_completion (text, start, end)
     char *text;
     int start, end;
{
  char **matches;

  matches = (char **)NULL;

  /* If this word is at the start of the line, then it is a command
     to complete.  Otherwise it is the name of a file in the current
     directory. */
  if (start == 0)
    matches = completion_matches (text, command_generator);

  return (matches);
}

/* Generator function for command completion.  STATE lets us know whether
   to start from scratch; without any state (i.e. STATE == 0), then we
   start at the top of the list. */
char *
command_generator (text, state)
     char *text;
     int state;
{
  static int list_index, len;
  char *name;

  /* If this is a new word to complete, initialize now.  This includes
     saving the length of TEXT for efficiency, and initializing the index
```


```
     variable to 0. */
  if (!state)
    {
      list_index = 0;
      len = strlen (text);
    }

  /* Return the next name which partially matches from the command list. */
  while (name = commands[list_index].name)
    {
      list_index++;

      if (strncmp (name, text, len) == 0)
        return (dupstr(name));
    }

  /* If no names matched, then return NULL. */
  return ((char *)NULL);
}

/* ********************************************************************** */
/*                                                                        */
/*                        FileMan Commands                                */
/*                                                                        */
/* ********************************************************************** */

/* String to pass to system ().  This is for the LIST, VIEW and RENAME
   commands. */
static char syscom[1024];

/* List the file(s) named in arg. */
com_list (arg)
     char *arg;
{
  if (!arg)
    arg = "";

  sprintf (syscom, "ls -FClg %s", arg);
  return (system (syscom));
}

com_view (arg)
     char *arg;
{
  if (!valid_argument ("view", arg))
    return 1;

  sprintf (syscom, "more %s", arg);
  return (system (syscom));
}
```


```
com_rename (arg)
     char *arg;
{
  too_dangerous ("rename");
  return (1);
}

com_stat (arg)
     char *arg;
{
  struct stat finfo;

  if (!valid_argument ("stat", arg))
    return (1);

  if (stat (arg, &finfo) == -1)
    {
      perror (arg);
      return (1);
    }

  printf ("Statistics for `%s':\n", arg);

  printf ("%s has %d link%s, and is %d byte%s in length.\n", arg,
          finfo.st_nlink,
          (finfo.st_nlink == 1) ? "" : "s",
          finfo.st_size,
          (finfo.st_size == 1) ? "" : "s");
  printf ("Inode Last Change at: %s", ctime (&finfo.st_ctime));
  printf ("      Last access at: %s", ctime (&finfo.st_atime));
  printf ("    Last modified at: %s", ctime (&finfo.st_mtime));
  return (0);
}

com_delete (arg)
     char *arg;
{
  too_dangerous ("delete");
  return (1);
}

/* Print out help for ARG, or for all of the commands if ARG is
   not present. */
com_help (arg)
     char *arg;
{
  register int i;
  int printed = 0;
```


```
  for (i = 0; commands[i].name; i++)
    {
      if (!*arg || (strcmp (arg, commands[i].name) == 0))
        {
          printf ("%s\t\t%s.\n", commands[i].name, commands[i].doc);
          printed++;
        }
    }

  if (!printed)
    {
      printf ("No commands match `%s'.  Possibilties are:\n", arg);

      for (i = 0; commands[i].name; i++)
        {
          /* Print in six columns. */
          if (printed == 6)
            {
              printed = 0;
              printf ("\n");
            }

          printf ("%s\t", commands[i].name);
          printed++;
        }

      if (printed)
        printf ("\n");
    }
  return (0);
}

/* Change to the directory ARG. */
com_cd (arg)
     char *arg;
{
  if (chdir (arg) == -1)
    {
      perror (arg);
      return 1;
    }

  com_pwd ("");
  return (0);
}

/* Print out the current working directory. */
com_pwd (ignore)
     char *ignore;
{
```


```
  char dir[1024], *s;

  s = getwd (dir);
  if (s == 0)
    {
      printf ("Error getting pwd: %s\n", dir);
      return 1;
    }

  printf ("Current directory is %s\n", dir);
  return 0;
}

/* The user wishes to quit using this program.  Just set DONE non-zero. */
com_quit (arg)
     char *arg;
{
  done = 1;
  return (0);
}

/* Function which tells you that you can't do this. */
too_dangerous (caller)
     char *caller;
{
  fprintf (stderr,
           "%s: Too dangerous for me to distribute.  Write it yourself.\n",
           caller);
}

/* Return non-zero if ARG is a valid argument for CALLER, else print
   an error message and return zero. */
int
valid_argument (caller, arg)
     char *caller, *arg;
{
  if (!arg || !*arg)
    {
      fprintf (stderr, "%s: Argument required.\n", caller);
      return (0);
    }

  return (1);
}
```

Go to the previous, next section.

Go to the previous, next section.

Concept Index

i

k

r

y

Go to the previous, next section.

Go to the previous section.

Function and Variable Index

d

e

f

h

i

k

l

m

n

o

p

q

r

s

t

u

y

- yank-nth-arg (M-C-y)
- yank-pop (M-y)

Go to the previous section.

Texinfo

The GNU Documentation Format

Edition 2.18, for Texinfo Version Two

March 1993

by Robert J. Chassell and Richard M. Stallman

Go to the next section.

This is the second edition of the Texinfo documentation,
and is consistent with version 2 of `texinfo.tex'.

Published by the Free Software Foundation
675 Massachusetts Avenue,
Cambridge, MA 02139 USA
Printed copies are available for $15 each.
ISBN-1882114-12-4

Cover art by Etienne Suvasa.

Texinfo Copying Conditions

The programs currently being distributed that relate to Texinfo include portions of GNU Emacs, plus other separate programs (including `makeinfo`, `info`, `texindex`, and `texinfo.tex`). These programs are free; this means that everyone is free to use them and free to redistribute them on a free basis. The Texinfo-related programs are not in the public domain; they are copyrighted and there are restrictions on their distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of these programs that they might get from you.

Specifically, we want to make sure that you have the right to give away copies of the programs that relate to Texinfo, that you receive source code or else can get it if you want it, that you can change these programs or use pieces of them in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of the Texinfo related programs, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for the programs that relate to Texinfo. If these programs are modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems

introduced by others will not reflect on our reputation.

The precise conditions of the licenses for the programs currently being distributed that relate to Texinfo are found in the General Public Licenses that accompany them.

Go to the next section.

Go to the previous, next section.

Overview of Texinfo

Texinfo(1) is a documentation system that uses a single source file to produce both on-line information and printed output. This means that instead of writing two different documents, one for the on-line help or other on-line information and the other for a typeset manual or other printed work, you need write only one document. When the work is revised, you need revise only one document. (You can read the on-line information, known as an Info file, with an Info documentation-reading program.)

Using Texinfo, you can create a printed document with the normal features of a book, including chapters, sections, cross references, and indices. From the same Texinfo source file, you can create a menu-driven, on-line Info file with nodes, menus, cross references, and indices. You can, if you wish, make the chapters and sections of the printed document correspond to the nodes of the on-line information; and you use the same cross references and indices for both the Info file and the printed work. The GNU Emacs Manual is a good example of a Texinfo file, as is this manual.

To make a printed document, you process a Texinfo source file with the TeX typesetting program. This creates a DVI file that you can typeset and print as a book or report. (Note that the Texinfo language is completely different from TeX's usual language, PlainTeX, which Texinfo replaces.) If you do not have TeX, but do have `troff` or `nroff`, you can use the `texi2roff` program instead.

To make an Info file, you process a Texinfo source file with the `makeinfo` utility or Emacs's `texinfo-format-buffer` command; this creates an Info file that you can install on-line.

TeX and `texi2roff` work with many types of printer; similarly, Info works with almost every type of computer terminal. This power makes Texinfo a general purpose system, but brings with it a constraint, which is that a Texinfo file may contain only the customary "typewriter" characters (letters, numbers, spaces, and punctuation marks) but no special graphics.

A Texinfo file is a plain ASCII file containing text and @-commands (words preceded by an `@') that tell the typesetting and formatting programs what to do. You may edit a Texinfo file with any text editor; but it is especially convenient to use GNU Emacs since that editor has a special mode, called Texinfo mode, that provides various Texinfo-related features. (See section Using Texinfo Mode.)

Before writing a Texinfo source file, you should become familiar with the Info documentation reading program and learn about nodes, menus, cross references, and the rest. (@inforef{Top, info, info}, for more information.)

You can use Texinfo to create both on-line help and printed manuals; moreover, Texinfo is freely redistributable. For these reasons, Texinfo is the format in which documentation for GNU utilities and libraries is written.

Info files

An Info file is a Texinfo file formatted so that the Info documentation reading program can operate on it. (`makeinfo` and `texinfo-format-buffer` are two commands that convert a Texinfo file into an Info file.)

Info files are divided into pieces called nodes, each of which contains the discussion of one topic. Each node has a name, and contains both text for the user to read and pointers to other nodes, which are identified by their names. The Info program displays one node at a time, and provides commands with which the user can move to other related nodes.

Each node of an Info file may have any number of child nodes that describe subtopics of the node's topic. The names of child nodes are listed in a menu within the parent node; this allows you to use certain Info commands to move to one of the child nodes. Generally, an Info file is organized like a book. If a node is at the logical level of a chapter, its child nodes are at the level of sections; likewise, the child nodes of sections are at the level of subsections. All the children of any one parent are linked together in a bidirectional chain of `Next' and `Previous' pointers. The `Next' pointer provides a link to the next section, and the `Previous' pointer provides a link to the previous section. This means that all the nodes that are at the level of sections within a chapter are linked together. Normally the order in this chain is the same as the order of the children in the parent's menu. Each child node records the parent node name as its `Up' pointer. The last child has no `Next' pointer, and the first child has the parent both as its `Previous' and as its `Up' pointer.(2)

The book-like structuring of an Info file into nodes that correspond to chapters, sections, and the like is a matter of convention, not a requirement. The `Up', `Previous', and `Next' pointers of a node can point to any other nodes, and a menu can contain any other nodes. Thus, the node structure can be any directed graph. But it is usually more comprehensible to follow a structure that corresponds to the structure of chapters and sections in a printed book or report.

In addition to menus and to `Next', `Previous', and `Up' pointers, Info provides pointers of another kind, called references, that can be sprinkled throughout the text. This is usually the best way to represent links that do not fit a hierarchical structure.

Usually, you will design a document so that its nodes match the structure of chapters and sections in the printed output. But there are times when this is not right for the material being discussed. Therefore, Texinfo uses separate commands to specify the node structure for the Info file and the section structure for the printed output.

Generally, you enter an Info file through a node that by convention is called `Top'. This node normally contains just a brief summary of the file's purpose, and a large menu through which the rest of the file is reached. From this node, you can either traverse the file systematically by going from node to node, or you can go to a specific node listed in the main menu, or you can search the index menus and then go directly to the node that has the information you want.

If you want to read through an Info file in sequence, as if it were a printed manual, you can get the whole file with the advanced Info command g* RET. (@inforef{Expert, Advanced Info commands, info}.)

The `dir' file in the `info' directory serves as the departure point for the whole Info system. From

it, you can reach the `Top' nodes of each of the documents in a complete Info system.

Printed Books

A Texinfo file can be formatted and typeset as a printed book or manual. To do this, you need TeX, a powerful, sophisticated typesetting program written by Donald Knuth.(3)

A Texinfo-based book is similar to any other typeset, printed work: it can have a title page, copyright page, table of contents, and preface, as well as chapters, numbered or unnumbered sections and subsections, page headers, cross references, footnotes, and indices.

You can use Texinfo to write a book without ever having the intention of converting it into on-line information. You can use Texinfo for writing a printed novel, and even to write a printed memo, although this latter application is not recommended since electronic mail is so much easier.

TeX is a general purpose typesetting program. Texinfo provides a file called `texinfo.tex' that contains information (definitions or macros) that TeX uses when it typesets a Texinfo file. (`texinfo.tex' tells TeX how to convert the Texinfo @-commands to TeX commands, which TeX can then process to create the typeset document.) `texinfo.tex' contains the specifications for printing a document.

Most often, documents are printed on 8.5 inch by 11 inch pages (216mm by 280mm; this is the default size), but you can also print for 7 inch by 9.25 inch pages (178mm by 235mm; the `@smallbook` size) or on European A4 size paper (`@afourpaper`). (See section Printing "Small" Books. Also, see section Printing on A4 Paper.)

By changing the parameters in `texinfo.tex', you can change the size of the printed document. In addition, you can change the style in which the printed document is formatted; for example, you can change the sizes and fonts used, the amount of indentation for each paragraph, the degree to which words are hyphenated, and the like. By changing the specifications, you can make a book look dignified, old and serious, or light-hearted, young and cheery.

TeX is freely distributable. It is written in a dialect of Pascal called WEB and can be compiled either in Pascal or (by using a conversion program that comes with the TeX distribution) in C. (See section 'TeX Mode' in The GNU Emacs Manual, for information about TeX.)

TeX is very powerful and has a great many features. Because a Texinfo file must be able to present information both on a character-only terminal in Info form and in a typeset book, the formatting commands that Texinfo supports are necessarily limited.

See section How to Obtain TeX.

@-commands

In a Texinfo file, the commands that tell TeX how to typeset the printed manual and tell `makeinfo` and `texinfo-format-buffer` how to create an Info file are preceded by `@'; they are called

@-commands. For example, `@node` is the command to indicate a node and `@chapter` is the comm to indicate the start of a chapter.

> **Please note:** All the @-commands, with the exception of the `@TeX{}` command, must be written entirely in lower case.

The Texinfo @-commands are a strictly limited set of constructs. The strict limits make it possible for Texinfo files to be understood both by TeX and by the code that converts them into Info files. You can display Info files on any terminal that displays alphabetic and numeric characters. Similarly, you can print the output generated by TeX on a wide variety of printers.

Depending on what they do or what arguments(4) they take, you need to write @-commands on lines of their own or as part of sentences:

- Write a command such as `@noindent` at the beginning of a line as the only text on the line. (`@noindent` prevents the beginning of the next line from being indented as the beginning of a paragraph.)
- Write a command such as `@chapter` at the beginning of a line followed by the command's arguments, in this case the chapter title, on the rest of the line. (`@chapter` creates chapter titles.)
- Write a command such as `@dots{}` wherever you wish but usually within a sentence. (`@dots{}` creates dots ...)
- Write a command such as `@code{sample-code}` wherever you wish (but usually within a sentence) with its argument, sample-code in this example, between the braces. (`@code` marks text as being code.)
- Write a command such as `@example` at the beginning of a line of its own; write the body-text on following lines; and write the matching `@end` command, `@end example` in this case, at the beginning of a line of its own after the body-text. (`@example` ... `@end example` indents and typesets body-text as an example.)

As a general rule, a command requires braces if it mingles among other text; but it does not need braces if it starts a line of its own. The non-alphabetic commands, such as `@:`, are exceptions to the rule; they do not need braces.

As you gain experience with Texinfo, you will rapidly learn how to write the different commands: the different ways to write commands make it easier to write and read Texinfo files than if all commands followed exactly the same syntax. (For details about @-command syntax, see section @-Command Syntax.)

General Syntactic Conventions

All ASCII printing characters except `@', `{' and `}' can appear in a Texinfo file and stand for themselves. `@' is the escape character which introduces commands. `{' and `}' should be used only to surround arguments to certain commands. To put one of these special characters into the document, put an `@' character in front of it, like this: `@@', `@{', and `@}'.

It is customary in TeX to use doubled single-quote characters to begin and end quotations: @tt{ '' } and @w{@tt{ '' }}. This convention should be followed in Texinfo files. TeX converts doubled single-quote

characters to left- and right-hand doubled quotation marks, "like this", and Info converts doubled single-quote characters to ASCII double-quotes: @tt{ " } and @tt{ " } to @w{@tt{ " }}.

Use three hyphens in a row, `---', for a dash--like this. In TeX, a single or even a double hyphen produces a printed dash that is shorter than the usual typeset dash. Info reduces three hyphens to two for display on the screen.

To prevent a paragraph from being indented in the printed manual, put the command `@noindent` on a line by itself before the paragraph.

If you mark off a region of the Texinfo file with the `@iftex` and `@end iftex` commands, that region will appear only in the printed copy; in that region, you can use certain commands borrowed from PlainTeX that you cannot use in Info. Likewise, if you mark off a region with the `@ifinfo` and `@end ifinfo` commands, that region will appear only in the Info file; in that region, you can use Info commands that you cannot use in TeX. (See section Conditionally Visible Text.)

> **Caution:** Do not use tabs in a Texinfo file! TeX uses variable-width fonts, which means that it cannot predefine a tab to work in all circumstances. Consequently, TeX treats tabs like single spaces, and that is not what they look like.
>
> To avoid this problem, Texinfo mode causes GNU Emacs to insert multiple spaces when you press the TAB key.
>
> Also, you can run `untabify` in Emacs to convert tabs in a region to multiple spaces.

Comments

You can write comments in a Texinfo file that will not appear in either the Info file or the printed manual by using the `@comment` command (which may be abbreviated to `@c`). Such comments are for the person who reads the Texinfo file. All the text on a line that follows either `@comment` or `@c` is a comment; the rest of the line does not appear in either the Info file or the printed manual. (Often, you can write the `@comment` or `@c` in the middle of a line, and only the text that follows after the `@comment` or `@c` command does not appear; but some commands, such as `@settitle` and `@setfilename`, work on a whole line. You cannot use `@comment` or `@c` in a line beginning with such a command.)

You can write long stretches of text that will not appear in either the Info file or the printed manual by using the `@ignore` and `@end ignore` commands. Write each of these commands on a line of its own, starting each command at the beginning of the line. Text between these two commands does not appear in the processed output. You can use `@ignore` and `@end ignore` for writing comments. Often, `@ignore` and `@end ignore` is used to enclose a part of the copying permissions that applies to the Texinfo source file of a document, but not to the Info or printed version of the document.

What a Texinfo File Must Have

By convention, the names of Texinfo files end with one of the extensions `.texinfo', `.texi', or `.tex'. The longer extension is preferred since it describes more clearly to a human reader the nature of the file. The shorter extensions are for operating systems that cannot handle long file names.

In order to be made into a printed manual and an Info file, a Texinfo file **must** begin with lines like this:

```
\input texinfo
@setfilename info-file-name
@settitle name-of-manual
```

The contents of the file follow this beginning, and then you **must** end a Texinfo file with a line like this:

```
@bye
```

The `\input texinfo' line tells TeX to use the `texinfo.tex' file, which tells TeX how to translate the Texinfo @-commands into TeX typesetting commands. (Note the use of the backslash, `\'; this is correct for TeX.) The `@setfilename' line provides a name for the Info file and the `@settitle' line specifies a title for the page headers (or footers) of the printed manual.

The `@bye' line at the end of the file on a line of its own tells the formatters that the file is ended and to stop formatting.

Usually, you will not use quite such a spare format, but will include mode setting and start-of-header and end-of-header lines at the beginning of a Texinfo file, like this:

```
\input texinfo   @c -*-texinfo-*-
@c %**start of header
@setfilename info-file-name
@settitle name-of-manual
@c %**end of header
```

In the first line, `-*-texinfo-*-' causes Emacs to switch into Texinfo mode when you edit the file.

The `@c' lines which surround the `@setfilename' and `@settitle' lines are optional, but you need them in order to run TeX or Info on just part of the file. (See section Start of Header, for more information.)

Furthermore, you will usually provide a Texinfo file with a title page, indices, and the like. But the minimum, which can be useful for short documents, is just the three lines at the beginning and the one line at the end.

Six Parts of a Texinfo File

Generally, a Texinfo file contains more than the minimal beginning and end--it usually contains six parts:

1. Header

   The Header names the file, tells TeX which definitions' file to use, and performs other "housekeeping" tasks.

2. Summary Description and Copyright

   The Summary Description and Copyright segment describes the document and contains the copyright notice and copying permissions for the Info file. The segment must be enclosed between

`@ifinfo` and `@end ifinfo` commands so that the formatters place it only in the Info file.

3. Title and Copyright

   The Title and Copyright segment contains the title and copyright pages and copying permissions for the printed manual. The segment must be enclosed between `@titlepage` and `@end titlepage` commands. The title and copyright page appear only in the printed manual.

4. `Top' Node and Master Menu

   The Master Menu contains a complete menu of all the nodes in the whole Info file. It appears only in the Info file, in the `Top' node.

5. Body

   The Body of the document may be structured like a traditional book or encyclopedia or it may be free form.

6. End

   The End contains commands for printing indices and generating the table of contents, and the `@bye` command on a line of its own.

A Short Sample Texinfo File

Here is a complete but very short Texinfo file, in 6 parts. The first three parts of the file, from `\input texinfo' through to `@end titlepage', look more intimidating than they are. Most of the material is standard boilerplate; when you write a manual, simply insert the names for your own manual in this segment. (See section Beginning a Texinfo File.)

In the following, the sample text is *indented*; comments on it are not. The complete file, without any comments, is shown in section A Sample Texinfo File.

Part 1: Header

The header does not appear in either the Info file or the printed output. It sets various parameters, including the name of the Info file and the title used in the header.

```
\input texinfo   @c -*-texinfo-*-
@c %**start of header
@setfilename sample.info
@settitle Sample Document
@c %**end of header

@setchapternewpage odd
```

Part 2: Summary Description and Copyright

The summary description and copyright segment does not
appear in the printed document.

```
@ifinfo
This is a short example of a complete Texinfo file.

Copyright @copyright{} 1990 Free Software Foundation, Inc.
@end ifinfo
```

Part 3: Titlepage and Copyright

The titlepage segment does not appear in the Info file.

```
@titlepage
@sp 10
@comment The title is printed in a large font.
@center @titlefont{Sample Title}

@c The following two commands start the copyright page.
@page
@vskip 0pt plus 1filll
Copyright @copyright{} 1990 Free Software Foundation, Inc.
@end titlepage
```

Part 4: `Top' Node and Master Menu

The `Top' node contains the master menu for the Info file.
Since a printed manual uses a table of contents rather than
a menu, the master menu appears only in the Info file.

```
@node    Top,       First Chapter, (dir),    (dir)
@comment node-name, next,          previous, up

@menu
* First Chapter::    The first chapter is the
                     only chapter in this sample.
* Concept Index::    This index has two entries.
@end menu
```

Part 5: The Body of the Document

The body segment contains all the text of the document, but not the indices or table of contents. This example illustrates a node and a chapter containing an enumerated list.

```
@node    First Chapter, Concept Index, Top,      Top
@comment node-name,     next,          previous, up
@chapter First Chapter
@cindex Sample index entry

This is the contents of the first chapter.
@cindex Another sample index entry

Here is a numbered list.

@enumerate
@item
This is the first item.

@item
This is the second item.
@end enumerate

The @code{makeinfo} and @code{texinfo-format-buffer}
commands transform a Texinfo file such as this into
an Info file; and @TeX{} typesets it for a printed
manual.
```

Part 6: The End of the Document

The end segment contains commands both for generating an index in a node and unnumbered chapter of its own and for generating the table of contents; and it contains the `@bye` command that marks the end of the document.

```
@node    Concept Index,    ,  First Chapter, Top
@comment node-name,    next,  previous,      up
@unnumbered Concept Index

@printindex cp

@contents
@bye
```

The Results

Here is what the contents of the first chapter of the sample look like:

> This is the contents of the first chapter.
>
> Here is a numbered list.
>
> 1. This is the first item.
> 2. This is the second item.
>
> The `makeinfo` and `texinfo-format-buffer` commands transform a Texinfo file such as this into an Info file; and TeX typesets it for a printed manual.

Acknowledgements

Richard M. Stallman wrote Edition 1.0 of this manual. Robert J. Chassell revised and extended it, starting with Edition 1.1.

Our thanks go out to all who helped improve this work, particularly to Francois Pinard and David D. Zuhn, who tirelessly recorded and reported mistakes and obscurities; our special thanks go to Melissa Weisshaus for her frequent and often tedious reviews of nearly similar editions. Our mistakes are our own.

Go to the previous, next section.

Go to the previous, next section.

TOC

Using Texinfo Mode

You may edit a Texinfo file with any text editor you choose. A Texinfo file is no different from any other ASCII file. However, GNU Emacs comes with a special mode, called Texinfo mode, that provides Emacs commands and tools to help ease your work.

This chapter describes features of GNU Emacs' Texinfo mode but not any features of the Texinfo formatting language. If you are reading this manual straight through from the beginning, you may want to skim through this chapter briefly and come back to it after reading succeeding chapters which describe the Texinfo formatting language in detail.

Texinfo mode provides special features for working with Texinfo files:

- Insert frequently used @commands.
- Automatically create `@node` lines.
- Show the structure of a Texinfo source file.
- Automatically create or update the `Next', `Previous', and `Up' pointers of a node.
- Automatically create or update menus.
- Automatically create a master menu.
- Format a part or all of a file for Info.
- Typeset and print part or all of a file.

Perhaps the two most helpful features are those for inserting frequently used @-commands and for creating node pointers and menus.

The Usual GNU Emacs Editing Commands

In most cases, the usual Text mode commands work the same in Texinfo mode as they do in Text mode. Texinfo mode adds new editing commands and tools to GNU Emacs' general purpose editing features. The major difference concerns filling. In Texinfo mode, the paragraph separation variable and syntax table are redefined so that Texinfo commands that should be on lines of their own are not inadvertently included in paragraphs. Thus, the M-q (`fill-paragraph`) command will refill a paragraph but not mix an indexing command on a line adjacent to it into the paragraph.

In addition, Texinfo mode sets the `page-delimiter` variable to the value of `texinfo-chapter-level-regexp`; by default, this is a regular expression matching the commands for chapters and their equivalents, such as appendices. With this value for the page delimiter, you can jump from chapter title to chapter title with the C-x] (`forward-page`) and C-x [(`backward-page`) commands and narrow to a chapter with the C-x p (`narrow-to-page`) command. (See section 'Pages' in The GNU Emacs Manual, for details about the page commands.)

You may name a Texinfo file however you wish, but the convention is to end a Texinfo file name wi one of the three extensions `.texinfo', `.texi', or `.tex'. A longer extension is preferred, since it is explicit, but a shorter extension may be necessary for operating systems that limit the length of file names. GNU Emacs automatically enters Texinfo mode when you visit a file with a `.texinfo' or `.texi' extension. Also, Emacs switches to Texinfo mode when you visit a file that has `-*-texinfo-*-' in its first line. If ever you are in another mode and wish to switch to Texinfo mode, type `M-x texinfo-mode'.

Like all other Emacs features, you can customize or enhance Texinfo mode as you wish. In particular, the keybindings are very easy to change. The keybindings described here are the default or standard ones.

Inserting Frequently Used Commands

Texinfo mode provides commands to insert various frequently used @-commands into the buffer. You can use these commands to save keystrokes.

The insert commands are invoked by typing C-c twice and then the first letter of the @-command:

C-c C-c c

M-x texinfo-insert-@code

> Insert `@code{}` and put the cursor between the braces.

C-c C-c d

M-x texinfo-insert-@dfn

> Insert `@dfn{}` and put the cursor between the braces.

C-c C-c e

M-x texinfo-insert-@end

> Insert `@end` and attempt to insert the correct following word, such as `example' or `table'. (This command does not handle nested lists correctly, but inserts the word appropriate to the immediately preceding list.)

C-c C-c i

M-x texinfo-insert-@item

> Insert `@item` and put the cursor at the beginning of the next line.

C-c C-c k

M-x texinfo-insert-@kbd

> Insert `@kbd{}` and put the cursor between the braces.

C-c C-c n

M-x texinfo-insert-@node

> Insert `@node` and a comment line listing the sequence for the `Next', `Previous', and `Up' nodes. Leave point after the `@node`.

C-c C-c o

M-x texinfo-insert-@noindent

Insert `@noindent` and put the cursor at the beginning of the next line.

C-c C-c s

M-x texinfo-insert-@samp

Insert `@samp{}` and put the cursor between the braces.

C-c C-c t

M-x texinfo-insert-@table

Insert `@table` followed by a SPC and leave the cursor after the SPC.

C-c C-c v

M-x texinfo-insert-@var

Insert `@var{}` and put the cursor between the braces.

C-c C-c x

M-x texinfo-insert-@example

Insert `@example` and put the cursor at the beginning of the next line.

C-c C-c {

M-x texinfo-insert-braces

Insert `{}` and put the cursor between the braces.

C-c C-c }

C-c C-c]

M-x up-list

Move from between a pair of braces forward past the closing brace. Typing C-c C-c] is easier than typing C-c C-c }, which is, however, more mnemonic; hence the two keybindings. (Also, you can move out from between braces by typing C-f.)

To put a command such as `@code{...}` around an *existing* word, position the cursor in front of the word and type C-u 1 C-c C-c c. This makes it easy to edit existing plain text. The value of the prefix argument tells Emacs how many words following point to include between braces--1 for one word, 2 for two words, and so on. Use a negative argument to enclose the previous word or words. If you do not specify a prefix argument, Emacs inserts the @-command string and positions the cursor between the braces. This feature works only for those @-commands that operate on a word or words within one line, such as `@kbd` and `@var`.

This set of insert commands was created after analyzing the frequency with which different @-commands are used in the GNU Emacs Manual and the GDB Manual. If you wish to add your own insert commands, you can bind a keyboard macro to a key, use abbreviations, or extend the code in `` `texinfo.el' ``.

C-c C-c C-d (`texinfo-start-menu-description`) is an insert command that works differently from the other insert commands. It inserts a node's section or chapter title in the space for the description in a menu entry line. (A menu entry has three parts, the entry name, the node name, and the description. Only the node name is required, but a description helps explain what the node is about. See section The Parts of a Menu.)

To use `texinfo-start-menu-description`, position point in a menu entry line and type C-c C-d. The command looks for and copies the title that goes with the node name, and inserts the title as a description; it positions point at beginning of the inserted text so you can edit it. The function does not insert the title if the menu entry line already contains a description.

This command is only an aid to writing descriptions; it does not do the whole job. You must edit the inserted text since a title tends to use the same words as a node name but a useful description uses different words.

Showing the Section Structure of a File

You can show the section structure of a Texinfo file by using the C-c C-s command (`texinfo-show-structure`). This command shows the section structure of a Texinfo file by listing the lines that begin with the @-commands for `@chapter`, `@section`, and the like. It constructs what amounts to a table of contents. These lines are displayed in another buffer called the `*Occur*' buffer. In that buffer, you can position the cursor over one of the lines and use the C-c C-c command (`occur-mode-goto-occurrence`), to jump to the corresponding spot in the Texinfo file.

C-c C-s

M-x texinfo-show-structure

> Show the `@chapter`, `@section`, and such lines of a Texinfo file.

C-c C-c

M-x occur-mode-goto-occurrence

> Go to the line in the Texinfo file corresponding to the line under the cursor in the `*Occur*' buffer.

If you call `texinfo-show-structure` with a prefix argument by typing C-u C-c C-s, it will list not only those lines with the @-commands for `@chapter`, `@section`, and the like, but also the `@node` lines. (This is how the `texinfo-show-structure` command worked without an argument in the first version of Texinfo. It was changed because `@node` lines clutter up the `*Occur*' buffer and are usually not needed.) You can use `texinfo-show-structure` with a prefix argument to check whether the `Next', `Previous', and `Up' pointers of an `@node` line are correct.

Often, when you are working on a manual, you will be interested only in the structure of the current chapter. In this case, you can mark off the region of the buffer that you are interested in with the C-x n (`narrow-to-region`) command and `texinfo-show-structure` will work on only that region. To see the whole buffer again, use C-x w (`widen`). (See section 'Narrowing' in The GNU Emacs Manual, for more information about the narrowing commands.)

In addition to providing the `texinfo-show-structure` command, Texinfo mode sets the value of the page delimiter variable to match the chapter-level @-commands. This enables you to use the C-x] (`forward-page`) and C-x [(`backward-page`) commands to move forward and backward by chapter, and to use the C-x p (`narrow-to-page`) command to narrow to a chapter. See section 'Pages' in The GNU Emacs Manual, for more information about the page commands.

Updating Nodes and Menus

Texinfo mode provides commands for automatically creating or updating menus and node pointers. The commands are called "update" commands because their most frequent use is for updating a Texinfo file after you have worked on it; but you can use them to insert the `Next', `Previous', and `Up' pointers into an `@node` line that has none and to create menus in a file that has none.

If you do not use the updating commands, you need to write menus and node pointers by hand, which is a tedious task.

You can use the updating commands

- to insert or update the `Next', `Previous', and `Up' pointers of a node,
- to insert or update the menu for a section, and
- to create a master menu for a Texinfo source file.

You can also use the commands to update all the nodes and menus in a region or in a whole Texinfo file.

The updating commands work only with conventional Texinfo files, which are structured hierarchically like books. In such files, a structuring command line must follow closely after each `@node` line, except for the `Top' `@node` line. (A structuring command line is a line beginning with `@chapter`, `@section`, or other similar command.)

You can write the structuring command line on the line that follows immediately after an `@node` line or else on the line that follows after a single `@comment` line or a single `@ifinfo` line. You cannot interpose more than one line between the `@node` line and the structuring command line; and you may interpose only an `@comment` line or an `@ifinfo` line.

Commands which work on a whole buffer require that the `Top' node be followed by a node with an `@chapter` or equivalent-level command. Note that the menu updating commands will not create a main or master menu for a Texinfo file that has only `@chapter`-level nodes! The menu updating commands only create menus *within* nodes for lower level nodes. To create a menu of chapters, you must provide a `Top' node.

The menu updating commands remove menu entries that refer to other Info files since they do not refer to nodes within the current buffer. This is a deficiency. Rather than use menu entries, you can use cross references to refer to other Info files. None of the updating commands affect cross references.

Texinfo mode has five updating commands that are used most often: two are for updating the node pointers or menu of a single node (or a region); two are for updating every node pointer and menu in a file; and one, the `texinfo-master-menu` command, is for creating a master menu for a complete file, and optionally, for updating every node and menu in the whole Texinfo file.

The `texinfo-master-menu` command is the primary command:

C-c C-u m

M-x texinfo-master-menu

> Create or update a master menu that includes all the other menus (incorporating the descriptions from pre-existing menus, if any).

With an argument (prefix argument, C-u, if interactive), first create or update all the nodes and the regular menus in the buffer before constructing the master menu. (See section The `Top' Node and Master Menu, for more about a master menu.)

For `texinfo-master-menu` to work, the Texinfo file must have a `Top' node and at least one subsequent node.

After extensively editing a Texinfo file, you can type the following:

```
C-u M-x texinfo-master-menu
or
C-u C-c C-u m
```

This updates all the nodes and menus completely and all at once.

The other major updating commands do smaller jobs and are designed for the person who updates nodes and menus as he or she writes a Texinfo file.

The commands are:

C-c C-u C-n

M-x texinfo-update-node

Insert the `Next', `Previous', and `Up' pointers for the node that point is within (i.e., for the `@node` line preceding point). If the `@node` line has pre-existing `Next', `Previous', or `Up' pointers in it, the old pointers are removed and new ones inserted. With an argument (prefix argument, C-u, if interactive), this command updates all `@node` lines in the region (which is the text between point and mark).

C-c C-u C-m

M-x texinfo-make-menu

Create or update the menu in the node that point is within. With an argument (C-u as prefix argument, if interactive), the command makes or updates menus for the nodes which are either within or a part of the region.

Whenever `texinfo-make-menu` updates an existing menu, the descriptions from that menu are incorporated into the new menu. This is done by copying descriptions from the existing menu to the entries in the new menu that have the same node names. If the node names are different, the descriptions are not copied to the new menu.

C-c C-u C-e

M-x texinfo-every-node-update

Insert or update the `Next', `Previous', and `Up' pointers for every node in the buffer.

C-c C-u C-a

M-x texinfo-all-menus-update

Create or update all the menus in the buffer. With an argument (C-u as prefix argument, if interactive), first insert or update all the node pointers before working on the menus.

If a master menu exists, the `texinfo-all-menus-update` command updates it; but the command does not create a new master menu if none already exists. (Use the `texinfo-master-menu` command for that.)

When working on a document that does not merit a master menu, you can type the following:

```
C-u C-c C-u C-a
or
C-u M-x texinfo-all-menus-update
```

This updates all the nodes and menus.

The `texinfo-column-for-description` variable specifies the column to which menu descriptions are indented. By default, the value is 32 although it is often useful to reduce it to as low as 24. You can set the variable with the M-x edit-options command (see section 'Editing Variable Values' in The GNU Emacs Manual) or with the M-x set-variable command (see section 'Examining and Setting Variables' in The GNU Emacs Manual).

Also, the `texinfo-indent-menu-description` command may be used to indent existing menu descriptions to a specified column. Finally, if you wish, you can use the `texinfo-insert-node-lines` command to insert missing `@node` lines into a file. (See section Other Updating Commands, for more information.)

Updating Requirements

To use the updating commands, you must organize the Texinfo file hierarchically with chapters, sections, subsections, and the like. When you construct the hierarchy of the manual, do not `jump down' more than one level at a time: you can follow the `Top' node with a chapter, but not with a section; you can follow a chapter with a section, but not with a subsection. However, you may `jump up' any number of levels at one time--for example, from a subsection to a chapter.

Each `@node` line, with the exception of the line for the `Top' node, must be followed by a line with a structuring command such as `@chapter`, `@section`, or `@unnumberedsubsec`.

Each `@node` line/structuring-command line combination must look either like this:

```
@node     Comments,  Minimum, Conventions, Overview
@comment  node-name, next,    previous,    up
@section Comments
```

or like this (without the `@comment` line):

```
@node Comments, Minimum, Conventions, Overview
@section Comments
```

In this example, `Comments' is the name of both the node and the section. The next node is called `Minimum' and the previous node is called `Conventions'. The `Comments' section is within the

`Overview' node, which is specified by the `Up' pointer. (Instead of an `@comment` line, you can wri
`@ifinfo` line.)

If a file has a `Top' node, it must be called `top' or `Top' and be the first node in the file.

The menu updating commands create a menu of sections within a chapter, a menu of subsections within a section, and so on. This means that you must have a `Top' node if you want a menu of chapters.

Incidentally, the `makeinfo` command will create an Info file for a hierarchically organized Texinfo file that lacks `Next', `Previous' and `Up' pointers. Thus, if you can be sure that your Texinfo file will be formatted with `makeinfo`, you have no need for the `update node' commands. (See section Creating an Info File, for more information about `makeinfo`.) However, both `makeinfo` and the `texinfo-format-...` commands require that you insert menus in the file.

Other Updating Commands

In addition to the five major updating commands, Texinfo mode possesses several less frequently used updating commands:

M-x texinfo-insert-node-lines

> Insert `@node` lines before the `@chapter`, `@section`, and other sectioning commands wherever they are missing throughout a region in a Texinfo file.
>
> With an argument (C-u as prefix argument, if interactive), the `texinfo-insert-node-lines` command not only inserts `@node` lines but also inserts the chapter or section titles as the names of the corresponding nodes. In addition, it inserts the titles as node names in pre-existing `@node` lines that lack names. Since node names should be more concise than section or chapter titles, you must manually edit node names so inserted.
>
> For example, the following marks a whole buffer as a region and inserts `@node` lines and titles throughout:
>
> ```
> C-x h C-u M-x texinfo-insert-node-lines
> ```
>
> (Note that this command inserts titles as node names in `@node` lines; the `texinfo-start-menu-description` command (see section Inserting Frequently Used Commands) inserts titles as descriptions in menu entries, a different action. However, in both cases, you need to edit the inserted text.)

M-x texinfo-multiple-files-update

> Update nodes and menus in a document built from several separate files. With C-u as a prefix argument, create and insert a master menu in the outer file. With a numeric prefix argument, such as C-u 2, first update all the menus and all the `Next', `Previous', and `Up' pointers of all the included files before creating and inserting a master menu in the outer file. The `texinfo-multiple-files-update` command is described in the appendix on `@include` files. See section `texinfo-multiple-files-update`}.

M-x texinfo-indent-menu-description

Indent every description in the menu following point to the specified column. You can use this command to give yourself more space for descriptions. With an argument (C-u as prefix argument, if interactive), the `texinfo-indent-menu-description` command indents every description in every menu in the region. However, this command does not indent the second and subsequent lines of a multi-line description.

M-x texinfo-sequential-node-update

Insert the names of the nodes immediately following and preceding the current node as the `Next' or `Previous' pointers regardless of those nodes' hierarchical level. This means that the `Next' node of a subsection may well be the next chapter. Sequentially ordered nodes are useful for novels and other documents that you read through sequentially. (However, in Info, the `g* RET` command lets you look through the file sequentially, so sequentially ordered nodes are not strictly necessary.) With an argument (prefix argument, if interactive), the `texinfo-sequential-node-update` command sequentially updates all the nodes in the region.

Formatting for Info

Texinfo mode provides several commands for formatting part or all of a Texinfo file for Info. Often, when you are writing a document, you want to format only part of a file--that is, a region.

You can use either the `texinfo-format-region` or the `makeinfo-region` command to format a region:

C-c C-e C-r

M-x texinfo-format-region

C-c C-m C-r

M-x makeinfo-region

Format the current region for Info.

You can use either the `texinfo-format-buffer` or the `makeinfo-buffer` command to format a whole buffer:

C-c C-e C-b

M-x texinfo-format-buffer

C-c C-m C-b

M-x makeinfo-buffer

Format the current buffer for Info.

For example, after writing a Texinfo file, you can type the following:

```
C-u C-c C-u m
or
C-u M-x texinfo-master-menu
```

This updates all the nodes and menus. Then type the following to create an Info file:


```
C-c C-m C-b
or
M-x makeinfo-buffer
```

For the Info formatting commands to work, the file *must* include a line that has `@setfilename` in its header.

Not all systems support the `makeinfo`-based formatting commands.

See section Creating an Info File, for details about Info formatting.

Formatting and Printing

Typesetting and printing a Texinfo file is a multi-step process in which you first create a file for printing (called a DVI file), and then print the file. Optionally, you may also create indices. To do this, you must run the `texindex` command after first running the `tex` typesetting command; and then you must run the `tex` command again.

Often, when you are writing a document, you want to typeset and print only part of a file to see what it will look like. You can use the `texinfo-tex-region` and related commands for this purpose. Use the `texinfo-tex-buffer` command to format all of a buffer.

C-c C-t C-r

M-x texinfo-tex-region

> Run TeX on the region.

C-c C-t C-b

M-x texinfo-tex-buffer

> Run TeX on the buffer.

C-c C-t C-i

M-x texinfo-texindex

> Run `texindex` to sort the indices of a Texinfo file formatted with `texinfo-tex-region` or `texinfo-tex-buffer`. You must run the `tex` command a second time after sorting the raw index files.

C-c C-t C-p

M-x texinfo-tex-print

> Print the file (or the part of the file) previously formatted with `texinfo-tex-buffer` or `texinfo-tex-region`.

For `texinfo-tex-region` or `texinfo-tex-buffer` to work, the file *must* start with a `\input texinfo' line and must include an `@settitle` line. The file must end with `@bye` on a line by itself. (When you use `texinfo-tex-region`, you must surround the `@settitle` line with start-of-header and end-of-header lines.)

See section Format and Print Hardcopy, for a description of the other TeX related commands, such as

`tex-show-print-queue`.

Texinfo Mode Summary

In Texinfo mode, each set of commands has default keybindings that begin with the same keys. All the commands that are custom-created for Texinfo mode begin with C-c. The keys are somewhat mnemonic.

Insert Commands

The insert commands are invoked by typing C-c twice and then the first letter of the @-command to be inserted. (It might make more sense mnemonically to use C-c C-i, for `custom insert', but C-c C-c is quick to type.)

```
C-c C-c c       Insert `@code'.
C-c C-c d       Insert `@dfn'.
C-c C-c e       Insert `@end'.
C-c C-c i       Insert `@item'.
C-c C-c n       Insert `@node'.
C-c C-c s       Insert `@samp'.
C-c C-c v       Insert `@var'.
C-c C-c {       Insert braces.
C-c C-c ]
C-c C-c }       Move out of enclosing braces.

C-c C-c C-d     Insert a node's section title
                in the space for the description
                in a menu entry line.
```

Show Structure

The `texinfo-show-structure` command is often used within a narrowed region.

```
C-c C-s         List all the headings.
```

The Master Update Command

The `texinfo-master-menu` command creates a master menu; and can be used to update every node and menu in a file as well.

```
C-c C-u m
M-x texinfo-master-menu
                Create or update a master menu.

C-u C-c C-u m   With C-u as a prefix argument, first
```


```
                create or update all nodes and regular
                menus, and then create a master menu.
```

Update Pointers

The update pointer commands are invoked by typing C-c C-u and then either typing C-n for `texinfo-update-node` or typing C-e for `texinfo-every-node-update`.

```
C-c C-u C-n     Update a node.
C-c C-u C-e     Update every node in the buffer.
```

Update Menus

Invoke the update menu commands by typing C-c C-u and then either C-m for `texinfo-make-menu` or C-a for `texinfo-all-menus-update`. To update both nodes and menus at the same time, precede C-c C-u C-a with C-u.

```
C-c C-u C-m     Make or update a menu.

C-c C-u C-a     Make or update all
                menus in a buffer.

C-u C-c C-u C-a With C-u as a prefix argument,
                first create or update all nodes and
                then create or update all menus.
```

Format for Info

The Info formatting commands that are written in Emacs Lisp are invoked by typing C-c C-e and then either C-r for a region or C-b for the whole buffer.

The Info formatting commands that are written in C and based on the `makeinfo` program are invoked by typing C-c C-m and then either C-r for a region or C-b for the whole buffer.

Use the `texinfo-format...` commands:

```
C-c C-e C-r     Format the region.
C-c C-e C-b     Format the buffer.
```

Use `makeinfo`:

```
C-c C-m C-r     Format the region.
C-c C-m C-b     Format the buffer.
C-c C-m C-l     Recenter the makeinfo output buffer.
C-c C-m C-k     Kill the makeinfo formatting job.
```

Typeset and Print

The TeX typesetting and printing commands are invoked by typing C-c C-t and then another control command: C-r for `texinfo-tex-region`, C-b for `texinfo-tex-buffer`, and so on.

```
C-c C-t C-r     Run TeX on the region.
C-c C-t C-b     Run TeX on the buffer.
C-c C-t C-i     Run texindex.
C-c C-t C-p     Print the DVI file.
C-c C-t C-q     Show the print queue.
C-c C-t C-d     Delete a job from the print queue.
C-c C-t C-k     Kill the current TeX formatting job.
C-c C-t C-x     Quit a currently stopped TeX formatting job.
C-c C-t C-l     Recenter the output buffer.
```

Other Updating Commands

The `other updating commands' do not have standard keybindings because they are rarely used.

```
M-x texinfo-insert-node-lines
                Insert missing @node lines in region.
                With C-u as a prefix argument,
                use section titles as node names.

M-x texinfo-multiple-files-update
                Update a multi-file document.
                With C-u 2 as a prefix argument,
                create or update all nodes and menus
                in all included files first.

M-x texinfo-indent-menu-description
                Indent descriptions.

M-x texinfo-sequential-node-update
                Insert node pointers in strict sequence.
```

Go to the previous, next section.

Go to the previous, next section.

TOC

Beginning a Texinfo File

Certain pieces of information must be provided at the beginning of a Texinfo file, such as the name of the file and the title of the document.

Generally, the beginning of a Texinfo file has four parts:

1. The header, delimited by special comment lines, that includes the commands for naming the Texinfo file and telling TeX what definitions' file to use when processing the Texinfo file.
2. A short statement of what the file is about, with a copyright notice and copying permissions. This is enclosed in `@ifinfo` and `@end ifinfo` commands so that the formatters place it only in the Info file.
3. A title page and copyright page, with a copyright notice and copying permissions. This is enclosed between `@titlepage` and `@end titlepage` commands. The title and copyright page appear only in the printed manual.
4. The `Top' node that contains a menu for the whole Info file. The contents of this node appear only in the Info file.

Also, optionally, you may include the copying conditions for a program and a warranty disclaimer. The copying section will be followed by an introduction or else by the first chapter of the manual.

Since the copyright notice and copying permissions for the Texinfo document (in contrast to the copying permissions for a program) are in parts that appear only in the Info file or only in the printed manual, this information must be given twice.

Sample Texinfo File Beginning

The following sample shows what is needed.

```
\input texinfo   @c -*-texinfo-*-
@c %**start of header
@setfilename name-of-info-file
@settitle name-of-manual
@setchapternewpage odd
@c %**end of header

@ifinfo
This file documents ...

Copyright year copyright-owner

Permission is granted to ...
```


```
@end ifinfo

@c  This title page illustrates only one of the
@c  two methods of forming a title page.

@titlepage
@title name-of-manual-when-printed
@subtitle subtitle-if-any
@subtitle second-subtitle
@author author

@c  The following two commands
@c  start the copyright page.
@page
@vskip 0pt plus 1filll
Copyright @copyright{} year copyright-owner

Published by ...

Permission is granted to ...
@end titlepage

@node Top, Overview, (dir), (dir)

@ifinfo
This document describes ...

This document applies to version ...
of the program named ...
@end ifinfo

@menu
* Copying::          Your rights and freedoms.
* First Chapter::    Getting started ...
* Second Chapter::             ...
  ...
  ...
@end menu

@node    First Chapter, Second Chapter, top,      top
@comment node-name,     next,           previous, up
@chapter First Chapter
@cindex Index entry for First Chapter
```

The Texinfo File Header

Texinfo files start with at least three lines that provide Info and TeX with necessary information. These are the `\input texinfo` line, the `@settitle` line, and the `@setfilename` line. If you want to run TeX on just a part of the Texinfo File, you must write the `@settitle` and `@setfilename` lines between start-of-header and end-of-header lines.

Thus, the beginning of a Texinfo file looks like this:

```
\input texinfo   @c -*-texinfo-*-
@setfilename sample.info
@settitle Sample Document
```

or else like this:

```
\input texinfo   @c -*-texinfo-*-
@c %**start of header
@setfilename sample.info
@settitle Sample Document
@c %**end of header
```

The First Line of a Texinfo File

Every Texinfo file that is to be the top-level input to TeX must begin with a line that looks like this:

```
\input texinfo   @c -*-texinfo-*-
```

This line serves two functions:

1. When the file is processed by TeX, the `\input texinfo` command tells TeX to load the macros needed for processing a Texinfo file. These are in a file called `` `texinfo.tex' ``, which is usually located in the `` `/usr/lib/tex/macros' `` directory. TeX uses the backslash, `\', to mark the beginning of a command, just as Texinfo uses @. The `` `texinfo.tex' `` file causes the switch from `\' to `@'; before the switch occurs, TeX requires `\', which is why it appears at the beginning of the file.
2. When the file is edited in GNU Emacs, the `-*-texinfo-*-' mode specification tells Emacs to use Texinfo mode.

Start of Header

Write a start-of-header line on the second line of a Texinfo file. Follow the start-of-header line with `@setfilename` and `@settitle` lines and, optionally, with other command lines, such as `@smallbook` or `@footnotestyle`; and then by an end-of-header line (see section End of Header).

With these lines, you can format part of a Texinfo file for Info or typeset part for printing.

A start-of-header line looks like this:

```
@c %**start of header
```

The odd string of characters, `%**', is to ensure that no other comment is accidentally taken for a start-of-header line.

@setfilename

In order to be made into an Info file, a Texinfo file must contain a line that looks like this:

```
@setfilename info-file-name
```

Write the `@setfilename` command at the beginning of a line and follow it on the same line by the Info file name. Do not write anything else on the line; anything on the line after the command is considered part of the file name, including a comment.

The `@setfilename` line specifies the name of the Info file to be generated. This name should be different from the name of the Texinfo file. The convention is to write a name with a `.info' extension, to produce an Info file name such as `` `texinfo.info' ``.

Some operating systems cannot handle long file names. You can run into a problem even when the file name you specify is itself short enough. This occurs because the Info formatters split a long Info file into short indirect subfiles, and name them by appending `-1', `-2', ..., `-10', `-11', and so on, to the original file name. (See section Tag Files and Split Files.) The subfile name `` `texinfo.info-10' ``, for example, is too long for some systems; so the Info file name for this document is actually `` `texinfo' `` rather than `` `texinfo.info' ``.

The Info formatting commands ignore everything written before the `@setfilename` line, which is why the very first line of the file (the `\input` line) does not need to be commented out. The `@setfilename` line is ignored when you typeset a printed manual.

@settitle

In order to be made into a printed manual, a Texinfo file must contain a line that looks like this:

```
@settitle title
```

Write the `@settitle` command at the beginning of a line and follow it on the same line by the title. This tells TeX the title to use in a header or footer. Do not write anything else on the line; anything on the line after the command is considered part of the title, including a comment.

Conventionally, TeX formats a Texinfo file for double-sided output so as to print the title in the left-hand (even-numbered) page headings and the current chapter titles in the right-hand (odd-numbered) page headings. (TeX learns the title of each chapter from each `@chapter` command.) Page footers are not printed.

Even if you are printing in a single-sided style, TeX looks for an `@settitle` command line, in case you include the manual title in the heading.

The `@settitle` command should precede everything that generates actual output in TeX.

Although the title in the `@settitle` command is usually the same as the title on the title page, it does not affect the title as it appears on the title page. Thus, the two do not need not match exactly; and the title in the `@settitle` command can be a shortened or expanded version of the title as it appears on the title page. (See section `@titlepage`.)

TeX prints page headings only for that text that comes after the `@end titlepage` command in the Texinfo file, or that comes after an `@headings` command that turns on headings. (See section The `@headings` Command, for more information.)

You may, if you wish, create your own, customized headings and footings. See section Page Headings, for a detailed discussion of this process.

`@setchapternewpage`

In a book or a manual, text is usually printed on both sides of the paper, chapters start on right-hand pages, and right-hand pages have odd numbers. But in short reports, text often is printed only on one side of the paper. Also in short reports, chapters sometimes do not start on new pages, but are printed on the same page as the end of the preceding chapter, after a small amount of vertical whitespace.

You can use the `@setchapternewpage` command with various arguments to specify how TeX should start chapters and whether it should typeset pages for printing on one or both sides of the paper (single-sided or double-sided printing).

Write the `@setchapternewpage` command at the beginning of a line followed by its argument.

For example, you would write the following to cause each chapter to start on a fresh odd-numbered page:

```
@setchapternewpage odd
```

You can specify one of three alternatives with the `@setchapternewpage` command:

`@setchapternewpage off`

> Cause TeX to typeset a new chapter on the same page as the last chapter, after skipping some vertical whitespace. Also, cause TeX to format page headers for single-sided printing. (You can override the headers format with the `@headings double` command; see section The `@headings` Command.)

`@setchapternewpage on`

> Cause TeX to start new chapters on new pages and to typeset page headers for single-sided printing. This is the form most often used for short reports.
>
> This alternative is the default.

`@setchapternewpage odd`

Cause TeX to start new chapters on new, odd-numbered pages (right-handed pages) and to typ
for double-sided printing. This is the form most often used for books and manuals.

Texinfo does not have an `@setchapternewpage even` command.

(You can countermand or modify an `@setchapternewpage` command with an `@headings` command. See section The `@headings` Command Command}.)

At the beginning of a manual or book, pages are not numbered--for example, the title and copyright pages of a book are not numbered. By convention, table of contents pages are numbered with roman numerals and not in sequence with the rest of the document.

Since an Info file does not have pages, the `@setchapternewpage` command has no effect on it.

Usually, you do not write an `@setchapternewpage` command for single-sided printing, but accept the default which is to typeset for single-sided printing and to start new chapters on new pages. Usually, you write an `@setchapternewpage odd` command for double-sided printing.

Paragraph Indenting

The Info formatting commands may insert spaces at the beginning of the first line of each paragraph, thereby indenting that paragraph. You can use the `@paragraphindent` command to specify the indentation. Write an `@paragraphindent` command at the beginning of a line followed by either `asis' or a number. The template is:

```
@paragraphindent indent
```

The Info formatting commands indent according to the value of indent:

- If the value of indent is `asis', the Info formatting commands do not change the existing indentation.
- If the value of indent is 0, the Info formatting commands delete existing indentation.
- If the value of indent is greater than 0, the Info formatting commands indent the paragraph by that number of spaces.

The default value of indent is `asis'.

Write the `@paragraphindent` command before or shortly after the end-of-header line at the beginning of a Texinfo file. (If you write the command between the start-of-header and end-of-header lines, the region formatting commands indent paragraphs as specified.)

A peculiarity of `texinfo-format-buffer` and `texinfo-format-region` is that they do not indent (nor fill) paragraphs that contain `@w` or `@*` commands. See section Refilling Paragraphs, for a detailed description of what goes on.

End of Header

Follow the header lines with an end-of-header line. An end-of-header line looks like this:

```
@c %**end of header
```

If you include the `@setchapternewpage` command between the start-of-header and end-of-header lines, TeX will typeset a region as that command specifies. Similarly, if you include an `@smallbook` command between the start-of-header and end-of-header lines, TeX will typeset a region in the "small" book format.

See section Start of Header.

Summary and Copying Permissions for Info

The title page and the copyright page appear only in the printed copy of the manual; therefore, the same information must be inserted in a section that appears only in the Info file. This section usually contains a brief description of the contents of the Info file, a copyright notice, and copying permissions.

The copyright notice should read:

```
Copyright year copyright-owner
```

and be put on a line by itself.

Standard text for the copyright permissions is contained in an appendix to this manual; see section `ifinfo' Copying Permissions, for the complete text.

The permissions text appears in an Info file *before* the first node. This mean that a reader does *not* see this text when reading the file using Info, except when using the advanced Info command g *.

The Title and Copyright Pages

A manual's name and author are usually printed on a title page. Sometimes copyright information is printed on the title page as well; more often, copyright information is printed on the back of the title page.

The title and copyright pages appear in the printed manual, but not in the Info file. Because of this, it is possible to use several slightly obscure TeX typesetting commands that cannot be used in an Info file. In addition, this part of the beginning of a Texinfo file contains the text of the copying permissions that will appear in the printed manual.

See section Titlepage Copying Permissions, for the standard text for the copyright permissions.

@titlepage

Start the material for the title page and following copyright page with `@titlepage` on a line by itself and end it with `@end titlepage` on a line by itself.

The `@end titlepage` command starts a new page and turns on page numbering. (See section Page Headings, for details about how to generate of page headings.) All the material that you want to appear on unnumbered pages should be put between the `@titlepage` and `@end titlepage` commands. By using the `@page` command you can force a page break within the region delineated by the `@titlepage` and `@end titlepage` commands and thereby create more than one unnumbered page. This is how the copyright page is produced. (The `@titlepage` command might perhaps have been better named the `@titleandadditionalpages` command, but that would have been rather long!)

When you write a manual about a computer program, you should write the version of the program to which the manual applies on the title page. If the manual changes more frequently than the program or is independent of it, you should also include an edition number(5) for the manual. This helps readers keep track of which manual is for which version of the program. (The `Top' node should also contain this information; see section `@top'.)

Texinfo provides two methods for creating a title page. One method uses the `@titlefont`, `@sp`, and `@center` commands to generate a title page in which the words on the page are centered.

The second method uses the `@title`, `@subtitle`, and `@author` commands to create a title page with black rules under the title and author lines and the subtitle text set flush to the right hand side of the page. With this method, you do not specify any of the actual formatting of the title page. You specify the text you want, and Texinfo does the formatting. You may use either method.

@titlefont, @center, and @sp

You can use the `@titlefont`, `@sp`, and `@center` commands to create a title page for a printed document. (This is the first of the two methods for creating a title page in Texinfo.)

Use the `@titlefont` command to select a large font suitable for the title itself.

For example:

```
@titlefont{Texinfo}
```

Use the `@center` command at the beginning of a line to center the remaining text on that line. Thus,

```
@center @titlefont{Texinfo}
```

centers the title, which in this example is "Texinfo" printed in the title font.

Use the `@sp` command to insert vertical space. For example:

```
@sp 2
```

This inserts two blank lines on the printed page. (See section @sp n: Insert Blank Lines}, for more information about the @sp command.)

A template for this method looks like this:

```
@titlepage
@sp 10
@center @titlefont{name-of-manual-when-printed}
@sp 2
@center subtitle-if-any
@sp 2
@center author
...
@end titlepage
```

The spacing of the example fits an 8 1/2 by 11 inch manual.

@title, @subtitle, and @author

You can use the @title, @subtitle, and @author commands to create a title page in which the vertical and horizontal spacing is done for you automatically. This contrasts with the method described in the previous section, in which the @sp command is needed to adjust vertical spacing.

Write the @title, @subtitle, or @author commands at the beginning of a line followed by the title, subtitle, or author.

The @title command produces a line in which the title is set flush to the left-hand side of the page in a larger than normal font. The title is underlined with a black rule.

The @subtitle command sets subtitles in a normal-sized font flush to the right-hand side of the page.

The @author command sets the names of the author or authors in a middle-sized font flush to the left-hand side of the page on a line near the bottom of the title page. The names are underlined with a black rule that is thinner than the rule that underlines the title. (The black rule only occurs if the @author command line is followed by an @page command line.)

There are two ways to use the @author command: you can write the name or names on the remaining part of the line that starts with an @author command:

```
@author by Jane Smith and John Doe
```

or you can write the names one above each other by using two (or more) @author commands:

```
@author Jane Smith
@author John Doe
```

(Only the bottom name is underlined with a black rule.)

A template for this method looks like this:

```
@titlepage
@title name-of-manual-when-printed
@subtitle subtitle-if-any
@subtitle second-subtitle
@author author
@page
...
@end titlepage
```

Copyright Page and Permissions

By international treaty, the copyright notice for a book should be either on the title page or on the back of the title page. The copyright notice should include the year followed by the name of the organization or person who owns the copyright.

When the copyright notice is on the back of the title page, that page is customarily not numbered. Therefore, in Texinfo, the information on the copyright page should be within `@titlepage` and `@end titlepage` commands.

Use the `@page` command to cause a page break. To push the copyright notice and the other text on the copyright page towards the bottom of the page, you can write a somewhat mysterious line after the `@page` command that reads like this:

```
@vskip 0pt plus 1filll
```

This is a TeX command that is not supported by the Info formatting commands. The `@vskip` command inserts whitespace. The `0pt plus 1filll' means to put in zero points of mandatory whitespace, and as much optional whitespace as needed to push the following text to the bottom of the page. Note the use of three `l's in the word `filll'; this is the correct usage in TeX.

In a printed manual, the `@copyright{}` command generates a `c' inside a circle. (In Info, it generates `(C)'.) The copyright notice itself has the following legally defined sequence:

```
Copyright (C) year copyright-owner
```

It is customary to put information on how to get a manual after the copyright notice, followed by the copying permissions for the manual.

Note that permissions must be given here as well as in the summary segment within `@ifinfo` and `@end ifinfo` that immediately follows the header since this text appears only in the printed manual and the `ifinfo' text appears only in the Info file.

See section Sample Permissions, for the standard text.

Heading Generation

An `@end titlepage` command on a line by itself not only marks the end of the title and copyright pages, but also causes TeX to start generating page headings and page numbers.

To repeat what is said elsewhere, Texinfo has two standard page heading formats, one for documents which are printed on one side of each sheet of paper (single-sided printing), and the other for documents which are printed on both sides of each sheet (double-sided printing). (See section `@setchapternewpage`.) You can specify these formats in different ways:

- The conventional way is to write an `@setchapternewpage` command before the title page commands, and then have the `@end titlepage` command start generating page headings in the manner desired. (See section `@setchapternewpage`.)
- Alternatively, you can use the `@headings` command to prevent page headings from being generated or to start them for either single or double-sided printing. (Write an `@headings` command immediately after the `@end titlepage` command. See section The `@headings` Command Command}, for more information.)
- Or, you may specify your own page heading and footing format. See section Page Headings, for detailed information about page headings and footings.

Most documents are formatted with the standard single-sided or double-sided format, using `@setchapternewpage odd` for double-sided printing and no `@setchapternewpage` command for single-sided printing.

The `@headings` Command

The `@headings` command is rarely used. It specifies what kind of page headings and footings to print on each page. Usually, this is controlled by the `@setchapternewpage` command. You need the `@headings` command only if the `@setchapternewpage` command does not do what you want, or if you want to turn off pre-defined page headings prior to defining your own. Write an `@headings` command immediately after the `@end titlepage` command.

There are four ways to use the `@headings` command:

`@headings off`

: Turn off printing of page headings.

`@headings single`

: Turn on page headings appropriate for single-sided printing.

`@headings double`

`@headings on`

: Turn on page headings appropriate for double-sided printing. The two commands, `@headings on` and `@headings double`, are synonymous.

For example, suppose you write `@setchapternewpage off` before the `@titlepage` command to tell TeX to start a new chapter on the same page as the end of the last chapter. This command also causes

TeX to typeset page headers for single-sided printing. To cause TeX to typeset for double sided print, write `@headings double` after the `@end titlepage` command.

You can stop TeX from generating any page headings at all by writing `@headings off` on a line of its own immediately after the line containing the `@end titlepage` command, like this:

```
@end titlepage
@headings off
```

The `@headings off` command overrides the `@end titlepage` command, which would otherwise cause TeX to print page headings.

You can also specify your own style of page heading and footing. See section Page Headings, for more information.

The `Top' Node and Master Menu

The `Top' node is the node from which you enter an Info file.

A `Top' node should contain a brief description of the Info file and an extensive, master menu for the whole Info file. This helps the reader understand what the Info file is about. Also, you should write the version number of the program to which the Info file applies; or, at least, the edition number.

The contents of the `Top' node should appear only in the Info file; none of it should appear in printed output, so enclose it between `@ifinfo` and `@end ifinfo` commands. (TeX does not print either an `@node` line or a menu; they appear only in Info; strictly speaking, you are not required to enclose these parts between `@ifinfo` and `@end ifinfo`, but it is simplest to do so. See section Conditionally Visible Text.)

Sometimes, you will want to place an `@top` sectioning command line containing the title of the document immediately after the `@node Top` line (see section The `@top` Sectioning Command, for more information).

For example, the beginning of the Top node of this manual contains an `@top` sectioning command, a short description, and edition and version information. It looks like this:

```
...
@end titlepage

@ifinfo
@node Top, Copying, (dir), (dir)
@top Texinfo

Texinfo is a documentation system...

This is edition...
```


```
...
@end ifinfo

@menu
* Copying::                Texinfo is freely
                             redistributable.
* Overview::               What is Texinfo?
...
@end menu
```

In a `Top' node, the `Previous', and `Up' nodes usually refer to the top level directory of the whole Info system, which is called `(dir)'. The `Next' node refers to the first node that follows the main or master menu, which is usually the copying permissions, introduction, or first chapter.

Parts of a Master Menu

A master menu is a detailed main menu listing all the nodes in a file.

A master menu is enclosed in `@menu` and `@end menu` commands and does not appear in the printed document.

Generally, a master menu is divided into parts.

- The first part contains the major nodes in the Texinfo file: the nodes for the chapters, chapter-like sections, and the appendices.
- The second part contains nodes for the indices.
- The third and subsequent parts contain a listing of the other, lower level nodes, often ordered by chapter. This way, rather than go through an intermediary menu, an inquirer can go directly to a particular node when searching for specific information. These menu items are not required; add them if you think they are a convenience.

Each section in the menu can be introduced by a descriptive line. So long as the line does not begin with an asterisk, it will not be treated as a menu entry. (See section Writing a Menu, for more information.)

For example, the master menu for this manual looks like the following (but has many more entries):

```
@menu
* Copying::              Texinfo is freely
                           redistributable.
* Overview::             What is Texinfo?
* Texinfo Mode::         Special features in GNU Emacs.
...
...
* Command and Variable Index::
                        An entry for each @-command.
* Concept Index::        An entry for each concept.
```


```
 -- The Detailed Node Listing ---

Overview of Texinfo

* Info Files::           What is an Info file?
* Printed Manuals::      Characteristics of
                           a printed manual.
...
...

Using Texinfo Mode

* Info on a Region::     Formatting part of a file
                           for Info.
...
...
@end menu
```

Software Copying Permissions

If the Texinfo file has a section containing the "General Public License" and the distribution information and a warranty disclaimer for the software that is documented, this section usually follows the `Top' node. The General Public License is very important to Project GNU software. It ensures that you and others will continue to have a right to use and share the software.

The copying and distribution information and the disclaimer are followed by an introduction or else by the first chapter of the manual.

Although an introduction is not a required part of a Texinfo file, it is very helpful. Ideally, it should state clearly and concisely what the file is about and who would be interested in reading it. In general, an introduction would follow the licensing and distribution information, although sometimes people put it earlier in the document. Usually, an introduction is put in an `@unnumbered` section. (See section `@unnumbered`, `@appendix` Commands}.)

Go to the previous, next section.

Go to the previous, next section.

Ending a Texinfo File

The end of a Texinfo file should include the commands that create indices and generate detailed and summary tables of contents. And it must include the `@bye` command that marks the last line processed by TeX.

For example:

```
@node    Concept Index,     , Variables Index, Top
@c        node-name,    next, previous,        up
@unnumbered Concept Index

@printindex cp

@contents
@bye
```

Index Menus and Printing an Index

To print an index means to include it as part of a manual or Info file. This does not happen automatically just because you use `@cindex` or other index-entry generating commands in the Texinfo file; those just cause the raw data for the index to be accumulated. To generate an index, you must include the `@printindex` command at the place in the document where you want the index to appear. Also, as part of the process of creating a printed manual, you must run a program called `texindex` (see section Format and Print Hardcopy) to sort the raw data to produce a sorted index file. The sorted index file is what is actually used to print the index.

Texinfo offers six different types of predefined index: the concept index, the function index, the variables index, the keystroke index, the program index, and the data type index (see section Predefined Indices). Each index type has a two-letter name: `cp', `fn', `vr', `ky', `pg', and `tp'. You may merge indices, or put them into separate sections (see section Combining Indices); or you may define your own indices (see section Defining New Indices).

The `@printindex` command takes a two-letter index name, reads the corresponding sorted index file and formats it appropriately into an index.

The `@printindex` command does not generate a chapter heading for the index. Consequently, you should precede the `@printindex` command with a suitable section or chapter command (usually `@unnumbered`) to supply the chapter heading and put the index into the table of contents. Precede the `@unnumbered` command with an `@node` line.

For example:


```
@node Variable Index, Concept Index, Function Index, Top
@comment    node-name,         next,       previous, up
@unnumbered Variable Index

@printindex vr

@node     Concept Index,     , Variable Index, Top
@comment      node-name, next,       previous, up
@unnumbered Concept Index

@printindex cp

@summarycontents
@contents
@bye
```

(Readers often prefer that the concept index come last in a book, since that makes it easiest to find.)

Generating a Table of Contents

The `@chapter`, `@section`, and other structuring commands supply the information to make up a table of contents, but they do not cause an actual table to appear in the manual. To do this, you must use the `@contents` and `@summarycontents` commands:

`@contents`

> Generate a table of contents in a printed manual, including all chapters, sections, subsections, etc., as well as appendices and unnumbered chapters. (Headings generated by the `@heading` series of commands do not appear in the table of contents.) The `@contents` command should be written on a line by itself.

`@shortcontents`

`@summarycontents`

> (`@summarycontents` is a synonym for `@shortcontents`; the two commands are exactly the same.)
>
> Generate a short or summary table of contents that lists only the chapters (and appendices and unnumbered chapters). Omit sections, subsections and subsubsections. Only a long manual needs a short table of contents in addition to the full table of contents.
>
> Write the `@shortcontents` command on a line by itself right *before* the `@contents` command.

The table of contents commands automatically generate a chapter-like heading at the top of the first table of contents page. Write the table of contents commands at the very end of a Texinfo file, just before the `@bye` command, following any index sections--anything in the Texinfo file after the table of contents commands will be omitted from the table of contents.

When you print a manual with a table of contents, the table of contents are printed last and numbered with roman numerals. You need to place those pages in their proper place, after the title page, yourself. (This is the only collating you need to do for a printed manual. The table of contents is printed last because it is generated after the rest of the manual is typeset.)

Here is an example of where to write table of contents commands:

```
indices...
@shortcontents
@contents
@bye
```

Since an Info file uses menus instead of tables of contents, the Info formatting commands ignore the `@contents` and `@shortcontents` commands.

@bye File Ending

An `@bye` command terminates TeX or Info formatting. None of the formatting commands see any of the file following `@bye`. The `@bye` command should be on a line by itself.

If you wish, you may follow the `@bye` line with notes. These notes will not be formatted and will not appear in either Info or a printed manual; it is as if text after `@bye` were within `@ignore` ... `@end ignore`. Also, you may follow the `@bye` line with a local variables list. See section Using the Local Variables List, for more information.

Go to the previous, next section.

Go to the previous, next section.

Chapter Structuring

The chapter structuring commands divide a document into a hierarchy of chapters, sections, subsections, and subsubsections. These commands generate large headings; they also provide information for the table of contents of a printed manual (see section Generating a Table of Contents).

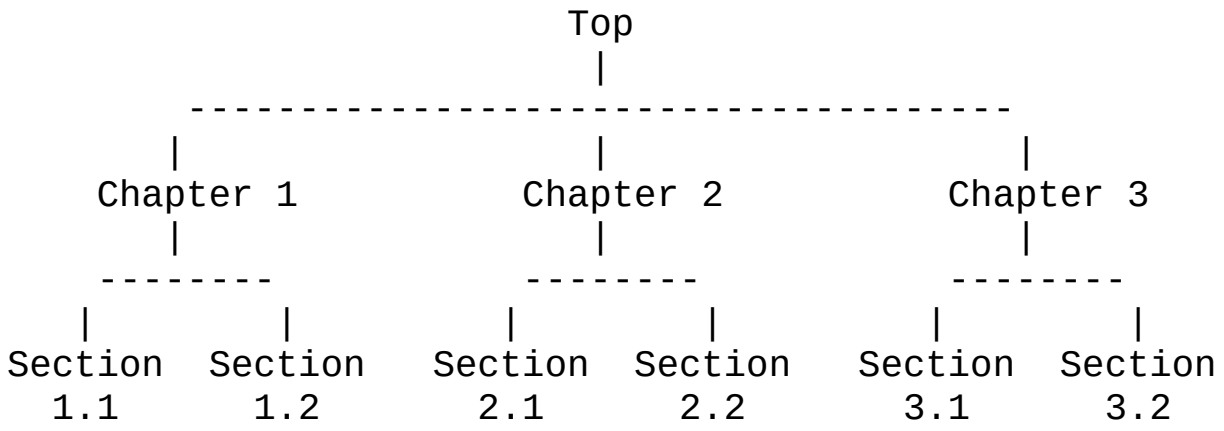
The chapter structuring commands do not create an Info node structure, so normally you should put an `@node` command immediately before each chapter structuring command (see section Nodes). The only time you are likely to use the chapter structuring commands without using the node structuring commands is if you are writing a document that contains no cross references and will never be transformed into Info format.

It is unlikely that you will ever write a Texinfo file that is intended only as an Info file and not as a printable document. If you do, you might still use chapter structuring commands to create a heading at the top of each node--but you don't need to.

Tree Structure of Sections

A Texinfo file is usually structured like a book with chapters, sections, subsections, and the like. This structure can be visualized as a tree (or rather as an upside-down tree) with the root at the top and the levels corresponding to chapters, sections, subsection, and subsubsections.

Here is a diagram that shows a Texinfo file with three chapters, each of which has two sections.



In a Texinfo file that has this structure, the beginning of Chapter 2 looks like this:

```
@node    Chapter 2,  Chapter 3, Chapter 1, top
@chapter Chapter 2
```

The chapter structuring commands are described in the sections that follow; the `@node` and `@menu`

commands are described in following chapters. (See section Nodes, and see section Menus.)

Types of Structuring Command

The chapter structuring commands fall into four groups or series, each of which contains structuring commands corresponding to the hierarchical levels of chapters, sections, subsections, and subsubsections.

The four groups are the `@chapter` series, the `@unnumbered` series, the `@appendix` series, and the `@heading` series.

Each command produces titles that have a different appearance on the printed page or Info file; only some of the commands produce titles that are listed in the table of contents of a printed book or manual.

- The `@chapter` and `@appendix` series of commands produce numbered or lettered entries both in the body of a printed work and in its table of contents.
- The `@unnumbered` series of commands produce unnumbered entries both in the body of a printed work and in its table of contents. The `@top` command, which has a special use, is a member of this series (see section `@top`).
- The `@heading` series of commands produce unnumbered headings that do not appear in a table of contents. The heading commands never start a new page.
- The `@majorheading` command produces results similar to using the `@chapheading` command but generates a larger vertical whitespace before the heading.
- When an `@setchapternewpage` command says to do so, the `@chapter`, `@unnumbered`, and `@appendix` commands start new pages in the printed manual; the `@heading` commands do not.

Here are the four groups of chapter structuring commands:

@top

The `@top` command is a special sectioning command that you use only after an `@node Top` line at the beginning of a Texinfo file. The `@top` command tells the `makeinfo` formatter which node is the `Top' node. It has the same typesetting effect as `@unnumbered` (see section `@unnumbered, @appendix`). For detailed information, see section The `@top` Sectioning Command.

@chapter

`@chapter` identifies a chapter in the document. Write the command at the beginning of a line and follow it on the same line by the title of the chapter.

For example, this chapter in this manual is entitled "Chapter Structuring"; the `@chapter` line looks like this:

`@chapter Chapter Structuring`

In TeX, the `@chapter` command creates a chapter in the document, specifying the chapter title. The chapter is numbered automatically.

In Info, the `@chapter` command causes the title to appear on a line by itself, with a line of asterisks inserted underneath. Thus, in Info, the above example produces the following output:

```
Chapter Structuring
*******************
```

@unnumbered, @appendix

Use the `@unnumbered` command to create a chapter that appears in a printed manual without chapter numbers of any kind. Use the `@appendix` command to create an appendix in a printed manual that is labelled by letter instead of by number.

For Info file output, the `@unnumbered` and `@appendix` commands are equivalent to `@chapter`: the title is printed on a line by itself with a line of asterisks underneath. (See section `@chapter`}.)

To create an appendix or an unnumbered chapter, write an `@appendix` or `@unnumbered` command at the beginning of a line and follow it on the same line by the title, as you would if you were creating a chapter.

@majorheading, @chapheading

The `@majorheading` and `@chapheading` commands put chapter-like headings in the body of a document.

However, neither command causes TeX to produce a numbered heading or an entry in the table of contents; and neither command causes TeX to start a new page in a printed manual.

In TeX, an `@majorheading` command generates a larger vertical whitespace before the heading than an `@chapheading` command but is otherwise the same.

In Info, the `@majorheading` and `@chapheading` commands are equivalent to `@chapter`: the title is printed on a line by itself with a line of asterisks underneath. (See section `@chapter`.)

@section

In a printed manual, an `@section` command identifies a numbered section within a chapter. The section title appears in the table of contents. In Info, an `@section` command provides a title for a segment of text, underlined with `=`'.

This section is headed with an `@section` command and looks like this in the Texinfo file:

`@section @code{@@section}`

To create a section, write the `@section` command at the beginning of a line and follow it on the same line by the section title.

Thus,

```
@section This is a section
```

produces

```
This is a section
==================
```

in Info.

@unnumberedsec, @appendixsec, @heading

The `@unnumberedsec`, `@appendixsec`, and `@heading` commands are, respectively, the unnumbered, appendix-like, and heading-like equivalents of the `@section` command. (See section `@section`.)

`@unnumberedsec`

> The `@unnumberedsec` command may be used within an unnumbered chapter or within a regular chapter or appendix to provide an unnumbered section.

`@appendixsec`

`@appendixsection`

> `@appendixsection` is a longer spelling of the `@appendixsec` command; the two are synonymous.
>
> Conventionally, the `@appendixsec` or `@appendixsection` command is used only within appendices.

`@heading`

> You may use the `@heading` command anywhere you wish for a section-style heading that will not appear in the table of contents.

The @subsection Command

Subsections are to sections as sections are to chapters. (See section `@section`.) In Info, subsection titles are underlined with `-'. For example,

```
@subsection This is a subsection
```

produces

```
This is a subsection
--------------------
```

In a printed manual, subsections are listed in the table of contents and are numbered three levels deep.

The `@subsection`-like Commands

The `@unnumberedsubsec`, `@appendixsubsec`, and `@subheading` commands are, respectively, the unnumbered, appendix-like, and heading-like equivalents of the `@subsection` command. (See section The `@subsection` Command.)

In Info, the `@subsection`-like commands generate a title underlined with hyphens. In a printed manual, an `@subheading` command produces a heading like that of a subsection except that it is not numbered and does not appear in the table of contents. Similarly, an `@unnumberedsubsec` command produces an unnumbered heading like that of a subsection and an `@appendixsubsec` command produces a subsection-like heading labelled with a letter and numbers; both of these commands produce headings that appear in the table of contents.

The `subsub' Commands

The fourth and lowest level sectioning commands in Texinfo are the `subsub' commands. They are:

`@subsubsection`

Subsubsections are to subsections as subsections are to sections. (See section The `@subsection` Command.) In a printed manual, subsubsection titles appear in the table of contents and are numbered four levels deep.

`@unnumberedsubsubsec`

Unnumbered subsubsection titles appear in the table of contents of a printed manual, but lack numbers. Otherwise, unnumbered subsubsections are the same as subsubsections. In Info, unnumbered subsubsections look exactly like ordinary subsubsections.

`@appendixsubsubsec`

Conventionally, appendix commands are used only for appendices and are lettered and numbered appropriately in a printed manual. They also appear in the table of contents. In Info, appendix subsubsections look exactly like ordinary subsubsections.

`@subsubheading`

The `@subsubheading` command may be used anywhere that you need a small heading that will not appear in the table of contents. In Info, subsubheadings look exactly like ordinary subsubsection headings.

In Info, `subsub' titles are underlined with periods. For example,


```
@subsubsection This is a subsubsection
```

produces

```
This is a subsubsection
.......................
```

Go to the previous, next section.

Go to the previous, next section.

Nodes

Nodes are the primary segments of a Texinfo file. They do not themselves impose a hierarchic or any other kind of structure on a file. Nodes contain node pointers that name other nodes, and can contain menus which are lists of nodes. In Info, the movement commands can carry you to a pointed-to node or to a node listed in a menu. Node pointers and menus provide structure for Info files just as chapters, sections, subsections, and the like, provide structure for printed books.

The node and menu commands and the chapter structuring commands are independent of each other:

- In Info, node and menu commands provide structure. The chapter structuring commands generate headings with different kinds of underlining--asterisks for chapters, hyphens for sections, and so on; they do nothing else.
- In TeX, the chapter structuring commands generate chapter and section numbers and tables of contents. The node and menu commands provide information for cross references; they do nothing else.

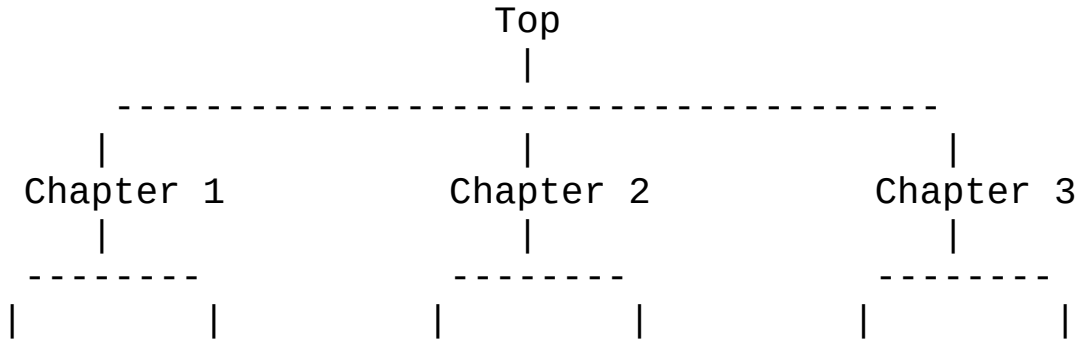
You can use node pointers and menus to structure an Info file any way you want; and you can write a Texinfo file so that its Info output has a different structure than its printed output. However, most Texinfo files are written such that the structure for the Info output corresponds to the structure for the printed output. It is not convenient to do otherwise.

Generally, printed output is structured in a tree-like hierarchy in which the chapters are the major limbs from which the sections branch out. Similarly, node pointers and menus are organized to create a matching structure in the Info output.

Node and Menu Illustration

Here is a copy of the diagram shown earlier that illustrates a Texinfo file with three chapters, each of which contains two sections.

Note that the "root" is at the top of the diagram and the "leaves" are at the bottom. This is how such a diagram is drawn conventionally; it illustrates an upside-down tree. For this reason, the root node is called the `Top' node, and `Up' node pointers carry you closer to the root.




```
 Section  Section   Section  Section   Section  Section
   1.1      1.2       2.1      2.2       3.1      3.2
```

Write the beginning of the node for Chapter 2 like this:

```
@node     Chapter 2,  Chapter 3, Chapter 1, top
@comment  node-name,  next,      previous,  up
```

This `@node` line says that the name of this node is "Chapter 2", the name of the `Next' node is "Chapter 3", the name of the `Previous' node is "Chapter 1", and the name of the `Up' node is "Top".

> **Please Note:** `Next' refers to the next node at the same hierarchical level in the manual, not necessarily to the next node within the Texinfo file. In the Texinfo file, the subsequent node may be at a lower level--a section-level node may follow a chapter-level node, and a subsection-level node may follow a section-level node. `Next' and `Previous' refer to nodes at the *same* hierarchical level. (The `Top' node contains the exception to this rule. Since the `Top' node is the only node at that level, `Next' refers to the first following node, which is almost always a chapter or chapter-level node.)

To go to Sections 2.1 and 2.2 using Info, you need a menu inside Chapter 2. (See section Menus.) You would write the menu just before the beginning of Section 2.1, like this:

```
@menu
* Sect. 2.1::    Description of this section.
* Sect. 2.2::
@end menu
```

Write the node for Sect. 2.1 like this:

```
@node     Sect. 2.1, Sect. 2.2, Chapter 2, Chapter 2
@comment  node-name, next,      previous,  up
```

In Info format, the `Next' and `Previous' pointers of a node usually lead to other nodes at the same level--from chapter to chapter or from section to section (sometimes, as shown, the `Previous' pointer points up); an `Up' pointer usually leads to a node at the level above (closer to the `Top' node); and a `Menu' leads to nodes at a level below (closer to `leaves'). (A cross reference can point to a node at any level; see section Cross References.)

Usually, an `@node` command and a chapter structuring command are used in sequence, along with indexing commands. (You may follow the `@node` line with a comment line that reminds you which pointer is which.)

Here is the beginning of the chapter in this manual called "Ending a Texinfo File". This shows an `@node` line followed by a comment line, an `@chapter` line, and then by indexing lines.

```
@node    Ending a File, Structuring, Beginning a File, Top
@comment node-name,     next,        previous,         up
```


```
@chapter Ending a Texinfo File
@cindex Ending a Texinfo file
@cindex Texinfo file ending
@cindex File ending
```

The @node Command

A node is a segment of text that begins at an `@node` command and continues until the next `@node` command. The definition of node is different from that for chapter or section. A chapter may contain sections and a section may contain subsections; but a node cannot contain subnodes; the text of a node continues only until the next `@node` command in the file. A node usually contains only one chapter structuring command, the one that follows the `@node` line. On the other hand, in printed output nodes are used only for cross references, so a chapter or section may contain any number of nodes. Indeed, a chapter usually contains several nodes, one for each section, subsection, and subsubsection.

To create a node, write an `@node` command at the beginning of a line, and follow it with four arguments, separated by commas, on the rest of the same line. These arguments are the name of the node, and the names of the `Next', `Previous', and `Up' pointers, in that order. You may insert spaces before each pointer if you wish; the spaces are ignored. You must write the name of the node, and the names of the `Next', `Previous', and `Up' pointers, all on the same line. Otherwise, the formatters fail. (@inforef{Top, info, info}, for more information about nodes in Info.)

Usually, you write one of the chapter-structuring command lines immediately after an `@node` line--for example, an `@section` or `@subsection` line. (See section Types of Structuring Command.)

> **Please note:** The GNU Emacs Texinfo mode updating commands work only with Texinfo files in which `@node` lines are followed by chapter structuring lines. See section Updating Requirements.

TeX uses `@node` lines to identify the names to use for cross references. For this reason, you must write `@node` lines in a Texinfo file that you intend to format for printing, even if you do not intend to format it for Info. (Cross references, such as the one at the end of this sentence, are made with `@xref` and its related commands; see section Cross References.)

The name of a node identifies the node. The pointers enable you to reach other nodes and consist of the names of those nodes.

Normally, a node's `Up' pointer contains the name of the node whose menu mentions that node. The node's `Next' pointer contains the name of the node that follows that node in that menu and its `Previous' pointer contains the name of the node that precedes it in that menu. When a node's `Previous' node is the same as its `Up' node, both node pointers name the same node.

Usually, the first node of a Texinfo file is the `Top' node, and its `Up' and `Previous' pointers point to the `dir' file, which contains the main menu for all of Info.

The `Top' node itself contains the main or master menu for the manual. Also, it is helpful to include a brief description of the manual in the `Top' node. See section The First Node, for information on how to

write the first node of a Texinfo file.

How to Write an @node Line

The easiest way to write an `@node` line is to write `@node` at the beginning of a line and then the name of the node, like this:

```
@node node-name
```

If you are using GNU Emacs, you can use the update node commands provided by Texinfo mode to insert the names of the pointers; or you can leave the pointers out of the Texinfo file and let `makeinfo` insert node pointers into the Info file it creates. (See section Using Texinfo Mode, and section Creating Pointers with `makeinfo`.)

Alternatively, you can insert the `Next', `Previous', and `Up' pointers yourself. If you do this, you may find it helpful to use the Texinfo mode keyboard command C-c C-c n. This command inserts `@node' and a comment line listing the names of the pointers in their proper order. The comment line helps you keep track of which arguments are for which pointers. This comment line is especially useful if you are not familiar with Texinfo.

The template for a node line with `Next', `Previous', and `Up' pointers looks like this:

```
@node node-name, next, previous, up
```

If you wish, you can ignore `@node` lines altogether in your first draft and then use the `texinfo-insert-node-lines` command to create `@node` lines for you. However, we do not recommend this practice. It is better to name the node itself at the same time that you write a segment so you can easily make cross references. A large number of cross references are an especially important feature of a good Info file.

After you have inserted an `@node` line, you should immediately write an @-command for the chapter or section and insert its name. Next (and this is important!), put in several index entries. Usually, you will find at least two and often as many as four or five ways of referring to the node in the index. Use them all. This will make it much easier for people to find the node.

@node Line Tips

Here are three suggestions:

- Try to pick node names that are informative but short.

  In the Info file, the file name, node name, and pointer names are all inserted on one line, which may run into the right edge of the window. (This does not cause a problem with Info, but is ugly.)
- Try to pick node names that differ from each other near the beginnings of their names. This way, it is easy to use automatic name completion in Info.
- By convention, node names are capitalized just as they would be for section or chapter titles--initial and significant words are capitalized; others are not.

@node Line Requirements

Here are several requirements for `@node` lines:

- All the node names for a single Info file must be unique.

  Duplicates confuse the Info movement commands. This means, for example, that if you end every chapter with a summary, you must name each summary node differently. You cannot just call each one "Summary". You may, however, duplicate the titles of chapters, sections, and the like. Thus you can end each chapter in a book with a section called "Summary", so long as the node names for those sections are all different.
- A pointer name must be the name of a node.

  The node to which a pointer points may come before or after the node containing the pointer.
- You cannot use any of the Texinfo @-commands in a node name; @-commands confuse Info.

  Thus, the beginning of the section called `@chapter` looks like this:

```
@node  chapter, unnumbered & appendix, makeinfo top, Structuring
@comment  node-name,  next,  previous,  up
@section @code{@@chapter}
@findex chapter
```

- You cannot use commas, colons, or apostrophes within a node name; these confuse TeX or the Info formatters.

  For example, the following is a section title:

```
@code{@@unnumberedsec}, @code{@@appendixsec}, @code{@@heading}
```

  The corresponding node name is:

```
unnumberedsec appendixsec heading
```

- Case is significant.

The First Node

The first node of a Texinfo file is the `Top' node, except in an included file (see section Include Files).

The `Top' node (which must be named `top' or `Top') should have as its `Up' and `Previous' nodes the name of a node in another file, where there is a menu that leads to this file. Specify the file name in parentheses. If the file is to be installed directly in the Info directory file, use `(dir)' as the parent of the `Top' node; this is short for `(dir)top', and specifies the `Top' node in the `dir' file, which contains the main menu for Info. For example, the `@node Top` line of this manual looks like this:

```
@node Top, Overview, (dir), (dir)
```

(You may use the Texinfo updating commands or the `makeinfo` utility to insert these `Next' and `(pointers automatically.)

See section Installing an Info File, for more information about installing an Info file in the `info' directory.

The `Top' node contains the main or master menu for the document.

The `@top` Sectioning Command

A special sectioning command, `@top`, has been created for use with the `@node Top` line. The `@top` sectioning command tells `makeinfo` that it marks the `Top' node in the file. It provides the information that `makeinfo` needs to insert node pointers automatically. Write the `@top` command at the beginning of the line immediately following the `@node Top` line. Write the title on the remaining part of the same line as the `@top` command.

In Info, the `@top` sectioning command causes the title to appear on a line by itself, with a line of asterisks inserted underneath.

In TeX and `texinfo-format-buffer`, the `@top` sectioning command is merely a synonym for `@unnumbered`. Neither of these formatters require an `@top` command, and do nothing special with it. You can use `@chapter` or `@unnumbered` after the `@node Top` line when you use these formatters. Also, you can use `@chapter` or `@unnumbered` when you use the Texinfo updating commands to create or update pointers and menus.

Whatever sectioning command follows an `@node Top` line, whether it be `@top` or `@chapter`, the `@node Top` line and the immediately following line and any additional text must be enclosed between `@ifinfo` and `@end ifinfo` commands. (See section Conditionally Visible Text.) This prevents the title and the accompanying text from appearing in printed output. Write the `@ifinfo` command before the `@node` line and write the `@end ifinfo` command after the `@top` or other sectioning command and after any additional text. (You can write the `@end ifinfo` command after the `@end menu` command if you like.)

The `Top' Node Summary

You can help readers by writing a summary in the `Top' node, after the `@top` line, before the main or master menu. The summary should briefly describe the Info file. You should also write the version number of the program to which the manual applies in this section. This helps the reader keep track of which manual is for which version of the program. If the manual changes more frequently than the program or is independent of it, you should also include an edition number for the manual. (The title page should also contain this information: see section `@titlepage`.)

Put the whole of the `Top' node, including the `@top` sectioning command line if you have one, between `@ifinfo` and `@end ifinfo` so none of the text appears in the printed output (see section Conditionally Visible Text). (You may want to repeat the brief description from the `Top' node within `@iftex` ... `@end iftex` at the beginning of the first chapter, for those who read the printed manual.)

Creating Pointers with `makeinfo`

The `makeinfo` program has a feature for automatically creating node pointers for a hierarchically organized file that lacks them.

When you take advantage of this feature, you do not need to write the `Next', `Previous', and `Up' pointers after the name of a node. However, you must write a sectioning command, such as `@chapter` or `@section`, on the line immediately following each truncated `@node` line. You cannot write a comment line after a node line; the section line must follow it immediately.

In addition, you must follow the `Top' `@node` line with a line beginning with `@top` to mark the `Top' node in the file. See section `@top`}.

Finally, you must write the name of each node (except for the `Top' node) in a menu that is one or more hierarchical levels above the node's hierarchical level.

This node pointer insertion feature in `makeinfo` is an alternative to the menu and pointer creation and update commands in Texinfo mode. (See section Updating Nodes and Menus.) It is especially helpful to people who do not use GNU Emacs for writing Texinfo documents.

Go to the previous, next section.

Go to the previous, next section.

Menus

Menus contain pointers to subordinate nodes.(6) In Info, you use menus to go to such nodes. Menus have no effect in printed manuals and do not appear in them.

By convention, a menu is put at the end of a node since a reader who uses the menu may not see text that follows it.

A node that has a menu should not contain much text. If you have a lot of text and a menu, move most of the text into a new subnode--all but a few lines. Otherwise, a reader with a terminal that displays only a few lines may miss the menu and its associated text. As a practical matter, you should locate a menu within 20 lines of the beginning of the node.

The short text before a menu may look awkward in a printed manual. To avoid this, you can write a menu near the beginning of its node and follow the menu by an `@node` line, and then an `@heading` line located within `@ifinfo` and `@end ifinfo`. This way, the menu, `@node` line, and title appear only in the Info file, not the printed document.

For example, the preceding two paragraphs follow an Info-only menu, `@node` line, and heading, and look like this:

```
@menu
* Menu Location::              Put a menu in a short node.
* Writing a Menu::             What is a menu?
* Menu Parts::                 A menu entry has three parts.
* Less Cluttered Menu Entry:: Two part menu entry.
* Menu Example::               Two and three part entries.
* Other Info Files::           How to refer to a different
                                 Info file.
@end menu

@node Menu Location, Writing a Menu,  , Menus
@ifinfo
@heading Menus Need Short Nodes
@end ifinfo
```

The Texinfo file for this document contains more than a dozen examples of this procedure. One is at the beginning of this chapter; another is at the beginning of the "Cross References" chapter.

Writing a Menu

A menu consists of an `@menu` command on a line by itself followed by menu entry lines or menu comment lines and then by an `@end menu` command on a line by itself.

A menu looks like this:

```
@menu
Larger Units of Text

* Files::                      All about handling files.
* Multiples: Buffers.          Multiple buffers; editing
                                 several files at once.
@end menu
```

In a menu, every line that begins with an `* ' is a menu entry. (Note the space after the asterisk.) A line that does not start with an `* ' may also appear in a menu. Such a line is not a menu entry but is a menu comment line that appears in the Info file. In the example above, the line `Larger Units of Text' is a menu comment line; the two lines starting with `* ' are menu entries.

The Parts of a Menu

A menu entry has three parts, only the second of which is required:

1. The menu entry name.
2. The name of the node (required).
3. A description of the item.

The template for a menu entry looks like this:

```
* menu-entry-name: node-name.   description
```

Follow the menu entry name with a single colon and follow the node name with tab, comma, period, or newline.

In Info, a user selects a node with the m (`Info-menu`) command. The menu entry name is what the user types after the m command.

The third part of a menu entry is a descriptive phrase or sentence. Menu entry names and node names are often short; the description explains to the reader what the node is about. The description, which is optional, can spread over two or more lines. A useful description complements the node name rather than repeats it.

Less Cluttered Menu Entry

When the menu entry name and node name are the same, you can write the name immediately after the asterisk and space at the beginning of the line and follow the name with two colons.

For example, write

```
* Name::                                  description
```

instead of

```
* Name: Name.                             description
```

You should use the node name for the menu entry name whenever possible, since it reduces visual clutter in the menu.

A Menu Example

A menu looks like this in Texinfo:

```
@menu
* menu entry name: Node name.   A short description.
* Node name::                   This form is preferred.
@end menu
```

This produces:

```
* menu:

* menu entry name: Node name.   A short description.
* Node name::                   This form is preferred.
```

Here is an example as you might see it in a Texinfo file:

```
@menu
Larger Units of Text

* Files::                       All about handling files.
* Multiples: Buffers.           Multiple buffers; editing
                                  several files at once.
@end menu
```

This produces:

```
* menu:
```


```
Larger Units of Text

* Files::                      All about handling files.
* Multiples: Buffers.          Multiple buffers; editing
                                 several files at once.
```

In this example, the menu has two entries. `Files' is both a menu entry name and the name of the node referred to by that name. `Multiples' is the menu entry name; it refers to the node named `Buffers'. The line `Larger Units of Text' is a comment; it appears in the menu, but is not an entry.

Since no file name is specified with either `Files' or `Buffers', they must be the names of nodes in the same Info file (see section Referring to Other Info Files).

Referring to Other Info Files

You can create a menu entry that enables a reader in Info to go to a node in another Info file by writing the file name in parentheses just before the node name. In this case, you should use the three-part menu entry format, which saves the reader from having to type the file name.

The format looks like this:

```
@menu
* first-entry-name:(filename)nodename.     description
* second-entry-name:(filename)second-node. description
@end menu
```

For example, to refer directly to the `Outlining' and `Rebinding' nodes in the Emacs Manual, you would write a menu like this:

```
@menu
* Outlining: (emacs)Outline Mode. The major mode for
                                  editing outlines.
* Rebinding: (emacs)Rebinding.    How to redefine the
                                  meaning of a key.
@end menu
```

If you do not list the node name, but only name the file, then Info presumes that you are referring to the `Top' node.

The `dir' file that contains the main menu for Info has menu entries that list only file names. These take you directly to the `Top' nodes of each Info document. (See section Installing an Info File.)

For example:

```
* Info: (info).          Documentation browsing system.
* Emacs: (emacs).        The extensible, self-documenting
```


```
text editor.
```

(The `dir' top level directory for the Info system is an Info file, not a Texinfo file, but a menu entry looks the same in both types of file.)

Note that the GNU Emacs Texinfo mode menu updating commands only work with nodes within the current buffer, so you cannot use them to create menus that refer to other files. You must write such menus by hand.

Go to the previous, next section.

Go to the previous, next section.

Cross References

Cross references are used to refer the reader to other parts of the same or different Texinfo files. In Texinfo, nodes are the places to which cross references can refer.

Often, but not always, a printed document should be designed so that it can be read sequentially. People tire of flipping back and forth to find information that should be presented to them as they need it.

However, in any document, some information will be too detailed for the current context, or incidental to it; use cross references to provide access to such information. Also, an on-line help system or a reference manual is not like a novel; few read such documents in sequence from beginning to end. Instead, people look up what they need. For this reason, such creations should contain many cross references to help readers find other information that they may not have read.

In a printed manual, a cross reference results in a page reference, unless it is to another manual altogether, in which case the cross reference names that manual.

In Info, a cross reference results in an entry that you can follow using the Info `f' command. (@inforef{Help-Adv, Some advanced Info commands, info}.)

The various cross reference commands use nodes to define cross reference locations. This is evident in Info, in which a cross reference takes you to the specified node. TeX also uses nodes to define cross reference locations, but the action is less obvious. When TeX generates a DVI file, it records nodes' page numbers and uses the page numbers in making references. Thus, if you are writing a manual that will only be printed, and will not be used on-line, you must nonetheless write `@node` lines to name the places to which you make cross references.

Different Cross Reference Commands

There are four different cross reference commands:

`@xref`

: Used to start a sentence in the printed manual saying `See ...' or an entry in the Info file saying `*Note ...'.

`@ref`

: Used within or, more often, at the end of a sentence; same as `@xref` for Info; produces just the reference in the printed manual without a preceding `See'.

`@pxref`

: Used within parentheses to make a reference that suits both an Info file and a printed book. Starts with a lower case `see' within the printed manual. (`p' is for `parenthesis'.)

`@inforef`

: Used to make a reference to an Info file for which there is no printed manual.

(The `@cite` command is used to make references to books and manuals for which there is no corresponding Info file and, therefore, no node to which to point. See section `@cite`{reference}.)

Parts of a Cross Reference

A cross reference command requires only one argument, which is the name of the node to which it refers. But a cross reference command may contain up to four additional arguments. By using these arguments, you can provide a cross reference name for Info, a topic description or section title for the printed output, the name of a different Info file, and the name of a different printed manual.

Here is a simple cross reference example:

```
@xref{Node name}.
```

which produces

```
*Note Node name::.
```

and

> See Section nnn [Node name], page ppp.

Here is an example of a full five-part cross reference:

```
@xref{Node name, Cross Reference Name, Particular Topic,
info-file-name, A Printed Manual}, for details.
```

which produces

```
*Note Cross Reference Name: (info-file-name)Node name,
for details.
```

in Info and

> See section "Particular Topic" in *A Printed Manual*, for details.

in a printed book.

The five possible arguments for a cross reference are:

1. The node name (required). This is the node to which the cross reference takes you. In a printed document, the location of the node provides the page reference only for references within the same document.
2. The cross reference name for the Info reference, if it is to be different from the node name. If you include this argument, it argument becomes the first part of the cross reference. It is usually omitted.
3. A topic description or section name. Often, this is the title of the section. This is used as the name of the reference in the printed manual. If omitted, the node name is used.

4. The name of the Info file in which the reference is located, if it is different from the current fil
5. The name of a printed manual from a different Texinfo file.

The template for a full five argument cross reference looks like this:

```
@xref{node-name, cross-reference-name, title-or-topic,
info-file-name, printed-manual-title}.
```

Cross references with one, two, three, four, and five arguments are described separately following the description of `@xref`.

Write a node name in a cross reference in exactly the same way as in the `@node` line, including the same capitalization; otherwise, the formatters may not find the reference.

You can write cross reference commands within a paragraph, but note how Info and TeX format the output of each of the various commands: write `@xref` at the beginning of a sentence; write `@pxref` only within parentheses, and so on.

@xref

The `@xref` command generates a cross reference for the beginning of a sentence. The Info formatting commands convert it into an Info cross reference, which the Info `f' command can use to bring you directly to another node. The TeX typesetting commands convert it into a page reference, or a reference to another book or manual.

Most often, an Info cross reference looks like this:

```
*Note node-name::.
```

or like this

```
*Note cross-reference-name: node-name.
```

In TeX, a cross reference looks like this:

```
See Section section-number [node-name], page page.
```

or like this

```
See Section section-number [title-or-topic], page page.
```

The `@xref` command does not generate a period or comma to end the cross reference in either the Info file or the printed output. You must write that period or comma yourself; otherwise, Info will not recognize the end of the reference. (The `@pxref` command works differently. See section `@pxref`.)

> **Please note:** A period or comma **must** follow the closing brace of an `@xref`. It is required to terminate the cross reference. This period or comma will appear in the output, both in the

Info file and in the printed manual.

`@xref` must refer to an Info node by name. Use `@node` to define the node (see section How to Write an `@node` Line).

`@xref` is followed by several arguments inside braces, separated by commas. Whitespace before and after these commas is ignored.

A cross reference requires only the name of a node; but it may contain up to four additional arguments. Each of these variations produces a cross reference that looks somewhat different.

> **Please note:** Commas separate arguments in a cross reference; avoid including them in the title or other part lest the formatters mistake them for separators.

`@xref` with One Argument

The simplest form of `@xref` takes one argument, the name of another node in the same Info file. The Info formatters produce output that the Info readers can use to jump to the reference; TeX produces output that specifies the page and section number for you.

For example,

```
@xref{Tropical Storms}.
```

produces

```
*Note Tropical Storms::.
```

and

> See Section 3.1 [Tropical Storms], page 24.

(Note that in the preceding example the closing brace is followed by a period.)

You can write a clause after the cross reference, like this:

```
@xref{Tropical Storms}, for more info.
```

which produces

```
*Note Tropical Storms::, for more info.
```

> See Section 3.1 [Tropical Storms], page 24, for more info.

(Note that in the preceding example the closing brace is followed by a comma, and then by the clause, which is followed by a period.)

@xref with Two Arguments

With two arguments, the second is used as the name of the Info cross reference, while the first is still the name of the node to which the cross reference points.

The template is like this:

```
@xref{node-name, cross-reference-name}.
```

For example,

```
@xref{Electrical Effects, Lightning}.
```

produces:

```
*Note Lightning: Electrical Effects.
```

and

> See Section 5.2 [Electrical Effects], page 57.

(Note that in the preceding example the closing brace is followed by a period; and that the node name is printed, not the cross reference name.)

You can write a clause after the cross reference, like this:

```
@xref{Electrical Effects, Lightning}, for more info.
```

which produces

```
*Note Lightning: Electrical Effects, for more info.
```

and

> See Section 5.2 [Electrical Effects], page 57, for more info.

(Note that in the preceding example the closing brace is followed by a comma, and then by the clause, which is followed by a period.)

@xref with Three Arguments

A third argument replaces the node name in the TeX output. The third argument should be the name of the section in the printed output, or else state the topic discussed by that section. Often, you will want to use initial upper case letters so it will be easier to read when the reference is printed. Use a third argument when the node name is unsuitable because of syntax or meaning.

Remember to avoid placing a comma within the title or topic section of a cross reference, or within any other section. The formatters divide cross references into arguments according to the commas; a comma within a title or other section will divide it into two arguments. In a reference, you need to write a title

such as "Clouds, Mist, and Fog" without the commas.

Also, remember to write a comma or period after the closing brace of a `@xref` to terminate the cross reference. In the following examples, a clause follows a terminating comma.

The template is like this:

```
@xref{node-name, cross-reference-name, title-or-topic}.
```

For example,

```
@xref{Electrical Effects, Lightning, Thunder and Lightning},
for details.
```

produces

```
*Note Lightning: Electrical Effects, for details.
```

and

> See Section 5.2 [Thunder and Lightning], page 57, for details.

If a third argument is given and the second one is empty, then the third argument serves both. (Note how two commas, side by side, mark the empty second argument.)

```
@xref{Electrical Effects, , Thunder and Lightning},
for details.
```

produces

```
*Note Thunder and Lightning: Electrical Effects, for details.
```

and

> See Section 5.2 [Thunder and Lightning], page 57, for details.

As a practical matter, it is often best to write cross references with just the first argument if the node name and the section title are the same, and with the first and third arguments if the node name and title are different.

Here are several examples from The GAWK Manual:

```
@xref{Sample Program}.
@xref{Glossary}.
@xref{Case-sensitivity, ,Case-sensitivity in Matching}.
@xref{Close Output, , Closing Output Files and Pipes},
   for more information.
@xref{Regexp, , Regular Expressions as Patterns}.
```

@xref with Four and Five Arguments

In a cross reference, a fourth argument specifies the name of another Info file, different from the file in which the reference appears, and a fifth argument specifies its title as a printed manual.

Remember that a comma or period must follow the closing brace of an `@xref` command to terminate the cross reference. In the following examples, a clause follows a terminating comma.

The template is:

```
@xref{node-name, cross-reference-name, title-or-topic,
info-file-name, printed-manual-title}.
```

For example,

```
@xref{Electrical Effects, Lightning, Thunder and Lightning,
weather, An Introduction to Meteorology}, for details.
```

produces

```
*Note Lightning: (weather)Electrical Effects, for details.
```

The name of the Info file is enclosed in parentheses and precedes the name of the node.

In a printed manual, the reference looks like this:

> See section "Thunder and Lightning" in *An Introduction to Meteorology*, for details.

The title of the printed manual is typeset in italics; and the reference lacks a page number since TeX cannot know to which page a reference refers when that reference is to another manual.

Often, you will leave out the second argument when you use the long version of `@xref`. In this case, the third argument, the topic description, will be used as the cross reference name in Info.

The template looks like this:

```
@xref{node-name, , title-or-topic, info-file-name,
printed-manual-title}, for details.
```

which produces

```
*Note title-or-topic: (info-file-name)node-name, for details.
```

and

> See section title-or-topic in printed-manual-title, for details.

For example,

```
@xref{Electrical Effects, , Thunder and Lightning,
```


```
weather, An Introduction to Meteorology}, for details.
```

produces

```
*Note Thunder and Lightning: (weather)Electrical Effects,
for details.
```

and

> See section "Thunder and Lightning" in *An Introduction to Meteorology*, for details.

On rare occasions, you may want to refer to another Info file that is within a single printed manual--when multiple Texinfo files are incorporated into the same TeX run but make separate Info files. In this case, you need to specify only the fourth argument, and not the fifth.

Naming a `Top' Node

In a cross reference, you must always name a node. This means that in order to refer to a whole manual, you must identify the `Top' node by writing it as the first argument to the `@xref` command. (This is different from the way you write a menu entry; see section Referring to Other Info Files.) At the same time, to provide a meaningful section topic or title in the printed cross reference (instead of the word `Top'), you must write an appropriate entry for the third argument to the `@xref` command.

Thus, to make a cross reference to The GNU Make Manual, write:

```
@xref{Top, , Overview, make, The GNU Make Manual}.
```

which produces

```
*Note Overview: (make)Top.
```

and

> See section "Overview" in *The GNU Make Manual*.

In this example, `Top' is the name of the first node, and `Overview' is the name of the first section of the manual.

@ref

`@ref` is nearly the same as `@xref` except that it does not generate a `See' in the printed output, just the reference itself. This makes it useful as the last part of a sentence.

For example,

```
For more information, see @ref{Hurricanes}.
```

produces


```
For more information, see *Note Hurricanes.
```

and

> For more information, see Section 8.2 [Hurricanes], page 123.

The `@ref` command sometimes leads writers to express themselves in a manner that is suitable for a printed manual but looks awkward in the Info format. Bear in mind that your audience will be using both the printed and the Info format.

For example,

```
Sea surges are described in @ref{Hurricanes}.
```

produces

> Sea surges are described in Section 6.7 [Hurricanes], page 72.

in a printed document, and the following in Info:

```
Sea surges are described in *Note Hurricanes::.
```

> **Caution:** You *must* write a period or comma immediately after an `@ref` command with two or more arguments. Otherwise, Info will not find the end of the cross reference entry and its attempt to follow the cross reference will fail. As a general rule, you should write a period or comma after every `@ref` command. This looks best in both the printed and the Info output.

@pxref

The parenthetical reference command, `@pxref`, is nearly the same as `@xref`, but you use it *only* inside parentheses and you do *not* type a comma or period after the command's closing brace. The command differs from `@xref` in two ways:

1. TeX typesets the reference for the printed manual with a lower case `see' rather than an upper case `See'.
2. The Info formatting commands automatically end the reference with a closing colon or period.

Because one type of formatting automatically inserts closing punctuation and the other does not, you should use `@pxref` *only* inside parentheses as part of another sentence. Also, you yourself should not insert punctuation after the reference, as you do with `@xref`.

`@pxref` is designed so that the output looks right and works right between parentheses both in printed output and in an Info file. In a printed manual, a closing comma or period should not follow a cross reference within parentheses; such punctuation is wrong. But in an Info file, suitable closing punctuation must follow the cross reference so Info can recognize its end. `@pxref` spares you the need to use complicated methods to put a terminator into one form of the output and not the other.

With one argument, a parenthetical cross reference looks like this:


```
... storms cause flooding (@pxref{Hurricanes}) ...
```

which produces

```
... storms cause flooding (*Note Hurricanes::) ...
```

and

> ... storms cause flooding (see Section 6.7 [Hurricanes], page 72) ...

With two arguments, a parenthetical cross reference has this template:

```
... (@pxref{node-name, cross-reference-name}) ...
```

which produces

```
... (*Note cross-reference-name: node-name.) ...
```

and

> ... (see Section nnn [node-name], page ppp) ...

`@pxref` can be used with up to five arguments just like `@xref` (see section `@xref`).

> **Please note:** Use `@pxref` only as a parenthetical reference. Do not try to use `@pxref` as a clause in a sentence. It will look bad in either the Info file, the printed output, or both.
>
> Also, parenthetical cross references look best at the ends of sentences. Although you may write them in the middle of a sentence, that location breaks up the flow of text.

`@inforef`

`@inforef` is used for cross references to Info files for which there are no printed manuals. Even in a printed manual, `@inforef` generates a reference directing the user to look in an Info file.

The command takes either two or three arguments, in the following order:

1. The node name.
2. The cross reference name (optional).
3. The Info file name.

Separate the arguments with commas, as with `@xref`. Also, you must terminate the reference with a comma or period after the `}', as you do with `@xref`.

The template is:

```
@inforef{node-name, cross-reference-name, info-file-name},
```

Thus,


```
@inforef{Expert, Advanced Info commands, info},
for more information.
```

produces

```
*Note Advanced Info commands: (info)Expert,
for more information.
```

and

> See Info file `info', node `Expert', for more information.

Similarly,

```
@inforef{Expert, , info}, for more information.
```

produces

```
*Note (info)Expert::, for more information.
```

and

> See Info file `info', node `Expert', for more information.

The converse of `@inforef` is `@cite`, which is used to refer to printed works for which no Info form exists. See section `@cite{reference}`.

Go to the previous, next section.

Go to the previous, next section.

Marking Words and Phrases

In Texinfo, you can mark words and phrases in a variety of ways. The Texinfo formatters use this information to determine how to highlight the text. You can specify, for example, whether a word or phrase is a defining occurrence, a metasyntactic variable, or a symbol used in a program. Also, you can emphasize text.

Indicating Definitions, Commands, etc.

Texinfo has commands for indicating just what kind of object a piece of text refers to. For example, metasyntactic variables are marked by `@var`, and code by `@code`. Since the pieces of text are labelled by commands that tell what kind of object they are, it is easy to change the way the Texinfo formatters prepare such text. (Texinfo is an *intentional* formatting language rather than a *typesetting* formatting language.)

For example, in a printed manual, code is usually illustrated in a typewriter font; `@code` tells TeX to typeset this text in this font. But it would be easy to change the way TeX highlights code to use another font, and this change would not effect how keystroke examples are highlighted. If straight typesetting commands were used in the body of the file and you wanted to make a change, you would need to check every single occurrence to make sure that you were changing code and not something else that should not be changed.

The highlighting commands can be used to generate useful information from the file, such as lists of functions or file names. It is possible, for example, to write a program in Emacs Lisp (or a keyboard macro) to insert an index entry after every paragraph that contains words or phrases marked by a specified command. You could do this to construct an index of functions if you had not already made the entries.

The commands serve a variety of purposes:

`@code{sample-code}`
: Indicate text that is a literal example of a piece of a program.

`@kbd{keyboard-characters}`
: Indicate keyboard input.

`@key{key-name}`
: Indicate the conventional name for a key on a keyboard.

`@samp{text}`
: Indicate text that is a literal example of a sequence of characters.

`@var{metasyntactic-variable}`
: Indicate a metasyntactic variable.

`@file{file-name}`

Indicate the name of a file.

`@dfn{term}`

Indicate the introductory or defining use of a term.

`@cite{reference}`

Indicate the name of a book.

@code{sample-code}

Use the `@code` command to indicate text that is a piece of a program and which consists of entire syntactic tokens. Enclose the text in braces.

Thus, you should use `@code` for an expression in a program, for the name of a variable or function used in a program, or for a keyword. Also, you should use `@code` for the name of a program, such as `diff`, that is a name used in the machine. (You should write the name of a program in the ordinary text font if you regard it as a new English word, such as `Emacs' or `Bison'.)

Use `@code` for environment variables such as `TEXINPUTS`, and other variables.

Use `@code` for command names in command languages that resemble programming languages, such as Texinfo or the shell. For example, `@code` and `@samp` are produced by writing `@code{@@code}' and `@code{@@samp}' in the Texinfo source, respectively.

Note, however, that you should not use `@code` for shell options such as `-c' when such options stand alone. (Use `@samp`.) Also, an entire shell command often looks better if written using `@samp` rather than `@code`. In this case, the rule is to choose the more pleasing format.

It is incorrect to alter the case of a word inside an `@code` command when it appears at the beginning of a sentence. Most computer languages are case sensitive. In C, for example, `Printf` is different from the identifier `printf`, and most likely is a misspelling of it. Even in languages which are not case sensitive, it is confusing to a human reader to see identifiers spelled in different ways. Pick one spelling and always use that. If you do not want to start a sentence with a command written all in lower case, you should rearrange the sentence.

Do not use the `@code` command for a string of characters shorter than a syntactic token. If you are writing about `TEXINPU', which is just a part of the name for the `TEXINPUTS` environment variable, you should use `@samp`.

In particular, you should not use the `@code` command when writing about the characters used in a token; do not, for example, use `@code` when you are explaining what letters or printable symbols can be used in the names of functions. (Use `@samp`.) Also, you should not use `@code` to mark text that is considered input to programs unless the input is written in a language that is like a programming language. For example, you should not use `@code` for the keystroke commands of GNU Emacs (use `@kbd` instead) although you may use `@code` for the names of the Emacs Lisp functions that the keystroke commands invoke.

In the printed manual, `@code` causes TeX to typeset the argument in a typewriter face. In the Info file, it causes the Info formatting commands to use single quotation marks around the text.

For example,

```
Use @code{diff} to compare two files.
```

produces this in the printed manual:

> Use `diff` to compare two files.

and this in the Info file:

```
Use `diff' to compare two files.
```

@kbd{keyboard-characters}

Use the `@kbd` command for characters of input to be typed by users. For example, to refer to the characters M-a, write

```
@kbd{M-a}
```

and to refer to the characters M-x shell, write

```
@kbd{M-x shell}
```

The `@kbd` command has the same effect as `@code` in Info, but may produce a different font in a printed manual.

You can embed another @-command inside the braces of an `@kbd` command. Here, for example, is the way to describe a command that would be described more verbosely as "press an `r' and then press the RET key":

```
@kbd{r @key{RET}}
```

This produces: r RET

You also use the `@kbd` command if you are spelling out the letters you type; for example:

```
To give the @code{logout} command,
type the characters @kbd{l o g o u t @key{RET}}.
```

This produces:

> To give the `logout` command, type the characters l o g o u t RET.

(Also, this example shows that you can add spaces for clarity. If you really want to mention a space character as one of the characters of input, write @key{SPC} for it.)

@key{key-name}

Use the `@key` command for the conventional name for a key on a keyboard, as in:

```
@key{RET}
```

You can use the `@key` command within the argument of an `@kbd` command when the sequence of characters to be typed includes one or more keys that are described by name.

For example, to produce C-x ESC you would type:

```
@kbd{C-x @key{ESC}}
```

Here is a list of the recommended names for keys; they are all in upper case:

`SPC`
: Space

`RET`
: Return

`LFD`
: Linefeed

`TAB`
: Tab

`BS`
: Backspace

`ESC`
: Escape

`DEL`
: Delete

`SFT`
: Shift

`CTL`
: Control

`META`
: Meta

There are subtleties to handling words like `meta' or `ctl' that are names of shift keys. When mentioning a character in which the shift key is used, such as Meta-a, use the `@kbd` command alone; do not use the `@key` command; but when you are referring to the shift key in isolation, use the `@key` command. For example, write `@kbd{Meta-a}' to produce Meta-a and `@key{META}' to produce META. This is because Meta-a refers to keys that you press on a keyboard, but META refers to a key without implying that you press it. In short, use `@kbd` for what you do, and use `@key` for what you talk about: "Press

`@kbd{M-a}` to move point to the beginning of the sentence. The `@key{META}` key is often in the lower left of the keyboard."

@samp{text}

Use the `@samp` command to indicate text that is a literal example or `sample' of a sequence of characters in a file, string, pattern, etc. Enclose the text in braces. The argument appears within single quotation marks in both the Info file and the printed manual; in addition, it is printed in a fixed-width font.

```
To match @samp{foo} at the end of the line,
use the regexp @samp{foo$}.
```

produces

> To match `foo' at the end of the line, use the regexp `foo$'.

Any time you are referring to single characters, you should use `@samp` unless `@kbd` is more appropriate. Use `@samp` for the names of command-line options. Also, you may use `@samp` for entire statements in C and for entire shell commands--in this case, `@samp` often looks better than `@code`. Basically, `@samp` is a catchall for whatever is not covered by `@code`, `@kbd`, or `@key`.

Only include punctuation marks within braces if they are part of the string you are specifying. Write punctuation marks outside the braces if those punctuation marks are part of the English text that surrounds the string. In the following sentence, for example, the commas and period are outside of the braces:

```
In English, the vowels are @samp{a}, @samp{e},
@samp{i}, @samp{o}, @samp{u}, and sometimes
@samp{y}.
```

This produces:

> In English, the vowels are `a', `e', `i', `o', `u', and sometimes `y'.

@var{metasyntactic-variable}

Use the `@var` command to indicate metasyntactic variables. A metasyntactic variable is something that stands for another piece of text. For example, you should use a metasyntactic variable in the documentation of a function to describe the arguments that are passed to that function.

Do not use `@var` for the names of particular variables in programming languages. These are specific names from a program, so `@code` is correct for them. For example, the Lisp variable `texinfo-tex-command` is not a metasyntactic variable; it is properly formatted using `@code`.

The effect of `@var` in the Info file is to change the case of the argument to all upper case; in the printed manual, to italicize it.

For example,


```
To delete file @var{filename},
type @code{rm @var{filename}}.
```

produces

> To delete file filename, type `rm filename`.

(Note that `@var` may appear inside `@code`, `@samp`, `@file`, etc.)

Write a metasyntactic variable all in lower case without spaces, and use hyphens to make it more readable. Thus, the Texinfo source for the illustration of how to begin a Texinfo manual looks like this:

```
\input texinfo
@@setfilename @var{info-file-name}
@@settitle @var{name-of-manual}
```

This produces:

```
\input texinfo
@setfilename info-file-name
@settitle name-of-manual
```

In some documentation styles, metasyntactic variables are shown with angle brackets, for example:

```
..., type rm <filename>
```

However, that is not the style that Texinfo uses. (You can, of course, modify the sources to TeX and the Info formatting commands to output the `<...>` format if you wish.)

@file{file-name}

Use the `@file` command to indicate text that is the name of a file, buffer, or directory, or is the name of a node in Info. You can also use the command for file name suffixes. Do not use `@file` for symbols in a programming language; use `@code`.

Currently, `@file` is equivalent to `@samp` in its effects. For example,

```
The @file{.el} files are in
the @file{/usr/local/emacs/lisp} directory.
```

produces

> The `.el' files are in the `/usr/local/emacs/lisp' directory.

@dfn{term}

Use the `@dfn` command to identify the introductory or defining use of a technical term. Use the command only in passages whose purpose is to introduce a term which will be used again or which the reader ought to know. Mere passing mention of a term for the first time does not deserve `@dfn`. The

command generates italics in the printed manual, and double quotation marks in the Info file. For example:

```
Getting rid of a file is called @dfn{deleting} it.
```

produces

> Getting rid of a file is called deleting it.

As a general rule, a sentence containing the defining occurrence of a term should be a definition of the term. The sentence does not need to say explicitly that it is a definition, but it should contain the information of a definition--it should make the meaning clear.

@cite{reference}

Use the `@cite` command for the name of a book that lacks a companion Info file. The command produces italics in the printed manual, and quotation marks in the Info file.

(If a book is written in Texinfo, it is better to use a cross reference command since a reader can easily follow such a reference in Info. See section `@xref`.)

Emphasizing Text

Usually, Texinfo changes the font to mark words in the text according to what category the words belong to; an example is the `@code` command. Most often, this is the best way to mark words. However, sometimes you will want to emphasize text without indicating a category. Texinfo has two commands to do this. Also, Texinfo has several commands that specify the font in which TeX will typeset text. These commands have no affect on Info and only one of them, the `@r` command, has any regular use.

@emph{text} and @strong{text}

The `@emph` and `@strong` commands are for emphasis; `@strong` is stronger. In printed output, `@emph` produces *italics* and `@strong` produces **bold**.

For example,

```
@quotation
@strong{Caution:} @code{rm * .[^.]*} removes @emph{all}
files in the directory.
@end quotation
```

produces the following in printed output:

> **Caution**: `rm * .[^.]*` removes *all* files in the directory.

and the following in Info:


```
*Caution*: `rm * .[^.]*' removes *all*
files in the directory.
```

The `@strong` command is seldom used except to mark what is, in effect, a typographical element, such as the word `Caution' in the preceding example.

In the Info file, both `@emph` and `@strong` put asterisks around the text.

> **Caution:** Do not use `@emph` or `@strong` with the word `Note'; Info will mistake the combination for a cross reference. Use a phrase such as **Please note** or **Caution** instead.

@sc{text}: The Small Caps Font

Use the `@sc' command to set text in the printed output in A SMALL CAPS FONT and set text in the Info file in upper case letters.

Write the text between braces in lower case, like this:

```
The @sc{acm} and @sc{ieee} are technical societies.
```

This produces:

```
The ACM and IEEE are technical societies.
```

TeX typesets the small caps font in a manner that prevents the letters from `jumping out at you on the page'. This makes small caps text easier to read than text in all upper case. The Info formatting commands set all small caps text in upper case.

If the text between the braces of an `@sc` command is upper case, TeX typesets in FULL-SIZE CAPITALS. Use full-size capitals sparingly.

You may also use the small caps font for a jargon word such as ATO (a NASA word meaning `abort to orbit').

There are subtleties to using the small caps font with a jargon word such as CDR, a word used in Lisp programming. In this case, you should use the small caps font when the word refers to the second and subsequent elements of a list (the CDR of the list), but you should use `@code' when the word refers to the Lisp function of the same spelling.

Fonts for Printing, Not Info

Texinfo provides four font commands that specify font changes in the printed manual but have no effect in the Info file. `@i` requests *italic* font (in some versions of TeX, a slanted font is used), `@b` requests **bold** face, `@t` requests the `fixed-width`, typewriter-style font used by `@code`, and `@r` requests a roman font, which is the usual font in which text is printed. All four commands apply to an argument that follows, surrounded by braces.

Only the `@r` command has much use: in example programs, you can use the `@r` command to convert code comments from the fixed-width font to a roman font. This looks better in printed output.

For example,

```
@lisp
(+ 2 2)    ; @r{Add two plus two.}
@end lisp
```

produces

```
(+ 2 2)    ; Add two plus two.
```

If possible, you should avoid using the other three font commands. If you need to use one, it probably indicates a gap in the Texinfo language.

Go to the previous, next section.

Go to the previous, next section.

Quotations and Examples

Quotations and examples are blocks of text consisting of one or more whole paragraphs that are set off from the bulk of the text and treated differently. They are usually indented.

In Texinfo, you always begin a quotation or example by writing an @-command at the beginning of a line by itself, and end it by writing an `@end` command that is also at the beginning of a line by itself. For instance, you begin an example by writing `@example` by itself at the beginning of a line and end the example by writing `@end example` on a line by itself, at the beginning of that line.

The Block Enclosing Commands

Here are commands for quotations and examples:

`@quotation`

Indicate text that is quoted. The text is filled, indented, and printed in a roman font by default.

`@example`

Illustrate code, commands, and the like. The text is printed in a fixed-width font, and indented but not filled.

`@lisp`

Illustrate Lisp code. The text is printed in a fixed-width font, and indented but not filled.

`@smallexample`

Illustrate code, commands, and the like. Similar to `@example`, except that in TeX this command typesets text in a smaller font for the smaller `@smallbook` format than for the 8.5 by 11 inch format.

`@smalllisp`

Illustrate Lisp code. Similar to `@lisp`, except that in TeX this command typesets text in a smaller font for the smaller `@smallbook` format than for the 8.5 by 11 inch format.

`@display`

Display illustrative text. The text is indented but not filled, and no font is specified (so, by default, the font is roman).

`@format`

Print illustrative text. The text is not indented and not filled and no font is specified (so, by default, the font is roman).

The `@exdent` command is used within the above constructs to undo the indentation of a line.

The `@flushleft` and `@flushright` commands are used to line up the left or right margins of unfilled text.

The `@noindent` command may be used after one of the above constructs to prevent the following t from being indented as a new paragraph.

You can use the `@cartouche` command within one of the above constructs to highlight the example or quotation by drawing a box with rounded corners around it. (The `@cartouche` command affects only the printed manual; it has no effect in the Info file; see section Drawing Cartouches Around Examples.)

@quotation

The text of a quotation is processed normally except that:

- the margins are closer to the center of the page, so the whole of the quotation is indented;
- the first lines of paragraphs are indented no more than other lines;
- in the printed output, interparagraph spacing is reduced.

  This is an example of text written between an `@quotation` command and an `@end quotation` command. An `@quotation` command is most often used to indicate text that is excerpted from another (real or hypothetical) printed work.

Write an `@quotation` command as text on a line by itself. This line will disappear from the output. Mark the end of the quotation with a line beginning with and containing only `@end quotation`. The `@end quotation` line will likewise disappear from the output. Thus, the following,

```
@quotation
This is
a foo.
@end quotation
```

produces

> This is a foo.

@example

The `@example` command is used to indicate an example that is not part of the running text, such as computer input or output.

```
This is an example of text written between an
@example command
and an @end example command.
The text is indented but not filled.

In the printed manual, the text is typeset in a
fixed-width font, and extra spaces and blank lines are
significant.  In the Info file, an analogous result is
obtained by indenting each line with five spaces.
```

Write an `@example` command at the beginning of a line by itself. This line will disappear from the output. Mark the end of the example with an `@end example` command, also written at the beginning of a line by itself. The `@end example` will disappear from the output.

For example,

```
@example
mv foo bar
@end example
```

produces

```
mv foo bar
```

Since the lines containing `@example` and `@end example` will disappear, you should put a blank line before the `@example` and another blank line after the `@end example`. (Remember that blank lines between the beginning `@example` and the ending `@end example` will appear in the output.)

> **Caution:** Do not use tabs in the lines of an example (or anywhere else in Texinfo, for that matter)! TeX treats tabs as single spaces, and that is not what they look like. This is a problem with TeX. (If necessary, in Emacs, you can use M-x untabify to convert tabs in a region to multiple spaces.)

Examples are often, logically speaking, "in the middle" of a paragraph, and the text continues after an example should not be indented. The `@noindent` command prevents a piece of text from being indented as if it were a new paragraph.

(The `@code` command is used for examples of code that are embedded within sentences, not set off from preceding and following text. See section `@code{sample-code}`.)

`@noindent`

An example or other inclusion can break a paragraph into segments. Ordinarily, the formatters indent text that follows an example as a new paragraph. However, you can prevent this by writing `@noindent` at the beginning of a line by itself preceding the continuation text.

For example:

```
@example
This is an example
@end example

@noindent
This line is not indented.  As you can see, the
beginning of the line is fully flush left with the line
that follows after it.  (This whole example is between
@code{@@display} and @code{@@end display}.)
```

produces

```
This is an example
```

```
This line is not indented.  As you can see, the
beginning of the line is fully flush left with the line
that follows after it.  (This whole example is between
@display and @end display.)
```

To adjust the number of blank lines properly in the Info file output, remember that the line containing `@noindent` does not generate a blank line, and neither does the `@end example` line.

In the Texinfo source file for this manual, each line that says `produces' is preceded by a line containing `@noindent`.

Do not put braces after an `@noindent` command; they are not necessary, since `@noindent` is a command used outside of paragraphs (see section @-Command Syntax).

@lisp

The `@lisp` command is used for Lisp code. It is synonymous with the `@example` command.

```
This is an example of text written between an
@lisp command and an @end lisp command.
```

Use `@lisp` instead of `@example` so as to preserve information regarding the nature of the example. This is useful, for example, if you write a function that evaluates only and all the Lisp code in a Texinfo file. Then you can use the Texinfo file as a Lisp library.(7)

Mark the end of `@lisp` with `@end lisp` on a line by itself.

@smallexample and @smalllisp

In addition to the regular `@example` and `@lisp` commands, Texinfo has two other "example-style" commands. These are the `@smallexample` and `@smalllisp` commands. Both these commands are designed for use with the `@smallbook` command that causes TeX to produce a printed manual in a 7 by 9.25 inch format rather than the regular 8.5 by 11 inch format.

In TeX, the `@smallexample` and `@smalllisp` commands typeset text in a smaller font for the smaller `@smallbook` format than for the 8.5 by 11 inch format. Consequently, many examples containing long lines fit in a narrower, `@smallbook` page without needing to be shortened. Both commands typeset in the normal font size when you format for the 8.5 by 11 inch size; indeed, in this

situation, the `@smallexample` and `@smalllisp` commands are defined to be the `@example` and `@lisp` commands.

In Info, the `@smallexample` and `@smalllisp` commands are equivalent to the `@example` and `@lisp` commands, and work exactly the same.

Mark the end of `@smallexample` or `@smalllisp` with `@end smallexample` or `@end smalllisp`, respectively.

Here is an example written in the small font used by the `@smallexample` and `@smalllisp` commands:

The `@smallexample` and `@smalllisp` commands make it easier to prepare smaller format manuals without forcing you to edit examples by hand to fit them onto narrower pages.

As a general rule, a printed document looks better if you write all the examples in a chapter consistently in `@example` or in `@smallexample`. Only occasionally should you mix the two formats.

See section Printing "Small" Books, for more information about the `@smallbook` command.

@display

The `@display` command begins a kind of example. It is like the `@example` command except that, in a printed manual, `@display` does not select the fixed-width font. In fact, it does not specify the font at all, so that the text appears in the same font it would have appeared in without the `@display` command.

```
This is an example of text written between an @display command
and an @end display command.  The @display command
indents the text, but does not fill it.
```

@format

The `@format` command is similar to `@example` except that, in the printed manual, `@format` does not select the fixed-width font and does not narrow the margins.

```
This is an example of text written between an @format command
and an @end format command.  As you can see
from this example,
the @format command does not fill the text.
```

@exdent: Undoing a Line's Indentation

The `@exdent` command removes any indentation a line might have. The command is written at the beginning of a line and applies only to the text that follows the command that is on the same line. Do not use braces around the text. In a printed manual, the text on an `@exdent` line is printed in the roman font.

`@exdent` is usually used within examples. Thus,

```
@example
This line follows an @@example command.
@exdent This line is exdented.
This line follows the exdented line.
The @@end example comes on the next line.
@end group
```

produces

```
This line follows an @example command.
This line is exdented.
This line follows the exdented line.
The @end example comes on the next line.
```

In practice, the `@exdent` command is rarely used. Usually, you un-indent text by ending the example and returning the page to its normal width.

@flushleft and @flushright

The `@flushleft` and `@flushright` commands line up the ends of lines on the left and right margins of a page, but do not fill the text. The commands are written on lines of their own, without braces. The `@flushleft` and `@flushright` commands are ended by `@end flushleft` and `@end flushright` commands on lines of their own.

For example,

```
@flushleft
This text is
written flushleft.
@end flushleft
```

produces

```
This text is
written flushleft.
```

Flushright produces the type of indentation often used in the return address of letters.

For example,

```
@flushright
Here is an example of text written
flushright.  The @code{@flushright} command
right justifies every line but leaves the
left end ragged.
@end flushright
```

produces

```
Here is an example of text written
flushright.  The @flushright command
right justifies every line but leaves the
left end ragged.
```

Drawing Cartouches Around Examples

In a printed manual, the `@cartouche` command draws a box with rounded corners around its contents. You can use this command to further highlight an example or quotation. For instance, you could write a manual in which one type of example is surrounded by a cartouche for emphasis.

The `@cartouche` command affects only the printed manual; it has no effect in the Info file.

For example,

```
@example
@cartouche
% pwd
/usr/local/lib/emacs/info
@end cartouche
@end example
```

surrounds the two-line example with a box with rounded corners, in the printed manual.

In a printed manual, the example looks like this:

```
% pwd
/usr/local/lib/emacs/info
```

Go to the previous, next section.

Go to the previous, next section.

Making Lists and Tables

Texinfo has several ways of making lists and two-column tables. Lists can be bulleted or numbered, while two-column tables can highlight the items in the first column.

Texinfo automatically indents the text in lists or tables, and numbers an enumerated list. This last feature is useful if you modify the list, since you do not need to renumber it yourself.

Numbered lists and tables begin with the appropriate @-command at the beginning of a line, and end with the corresponding `@end` command on a line by itself. The table and itemized-list commands also require that you write formatting information on the same line as the beginning @-command.

Begin an enumerated list, for example, with an `@enumerate` command and end the list with an `@end enumerate` command. Begin an itemized list with an `@itemize` command, followed on the same line by a formatting command such as `@bullet`, and end the list with an `@end itemize` command.

Precede each element of a list with an `@item` or `@itemx` command.

Here is an itemized list of the different kinds of table and lists:

- Itemized lists with and without bullets.
- Enumerated lists, using numbers or letters.
- Two-column tables with highlighting.

Here is an enumerated list with the same items:

1. Itemized lists with and without bullets.
2. Enumerated lists, using numbers or letters.
3. Two-column tables with highlighting.

And here is a two-column table with the same items and their @-commands:

`@itemize`
: Itemized lists with and without bullets.

`@enumerate`
: Enumerated lists, using numbers or letters.

`@table`
`@ftable`
`@vtable`
: Two-column tables with highlighting.

Making an Itemized List

The `@itemize` command produces sequences of indented paragraphs, with a bullet or other mark inside the left margin at the beginning of each paragraph for which such a mark is desired.

Begin an itemized list by writing `@itemize` at the beginning of a line. Follow the command, on the same line, with a character or a Texinfo command that generates a mark. Usually, you will write `@bullet` after `@itemize`, but you can use `@minus`, or any character or any special symbol that results in a single character in the Info file. (When you write `@bullet` or `@minus` after an `@itemize` command, you may omit the `{}'.)

Write the text of the indented paragraphs themselves after the `@itemize`, up to another line that says `@end itemize`.

Before each paragraph for which a mark in the margin is desired, write a line that says just `@item`. Do not write any other text on this line.

Usually, you should put a blank line before an `@item`. This puts a blank line in the Info file. (TeX inserts the proper interline whitespace in either case.) Except when the entries are very brief, these blank lines make the list look better.

Here is an example of the use of `@itemize`, followed by the output it produces. Note that `@bullet` produces an `*' in Info and a round dot in TeX.

```
@itemize @bullet
@item
Some text for foo.

@item
Some text
for bar.
@end itemize
```

This produces:

- Some text for foo.
- Some text for bar.

Itemized lists may be embedded within other itemized lists. Here is a list marked with dashes embedded in a list marked with bullets:

```
@itemize @bullet
@item
First item.

@itemize @minus
@item
```


```
Inner item.

@item
Second inner item.
@end itemize

@item
Second outer item.
@end itemize
```

This produces:

- First item.
  - Inner item.
  - Second inner item.
- Second outer item.

Making a Numbered or Lettered List

`@enumerate` is like `@itemize` except that the marks in the left margin contain successive integers or letters. (See section Making an Itemized List.)

Write the `@enumerate` command at the beginning of a line. The command does not require an argument, but accepts either a number or a letter as an option. Without an argument, `@enumerate` starts the list with the number 1. With a numeric argument, such as 3, the command starts the list with that number. With an upper or lower case letter, such as a or A, the command starts the list with that letter.

Write the text of the enumerated list in the same way you write an itemized list: put `@item` on a line of its own before the start of each paragraph that you want enumerated. Do not write any other text on the line beginning with `@item`.

You should put a blank line between entries in the list. This generally makes it easier to read the Info file.

Here is an example of `@enumerate` without an argument:

```
@enumerate
@item
Underlying causes.

@item
Proximate causes.
@end enumerate
```

This produces:

1. Underlying causes.

2. Proximate causes.

Here is an example with an argument of 3:

```
@enumerate 3
@item
Predisposing causes.

@item
Precipitating causes.

@item
Perpetuating causes.
@end enumerate
```

This produces:

1. Predisposing causes.
2. Precipitating causes.
3. Perpetuating causes.

Here is a brief summary of the alternatives. The summary is constructed using `@enumerate` with an argument of a.

1. `@enumerate`

   Without an argument, produce a numbered list, starting with the number 1.
2. `@enumerate positive-integer`

   With a (positive) numeric argument, start a numbered list with that number. You can use this to continue a list that you interrupted with other text.
3. `@enumerate upper-case-letter`

   With an upper case letter as argument, start a list in which each item is marked by a letter, beginning with that upper case letter.
4. `@enumerate lower-case-letter`

   With a lower case letter as argument, start a list in which each item is marked by a letter, beginning with that lower case letter.

You can also nest enumerated lists, as in an outline.

Making a Two-column Table

`@table` is similar to `@itemize`, but the command allows you to specify a name or heading line for each item. (See section Making an Itemized List}.) The `@table` command is used to produce two-column tables, and is especially useful for glossaries and explanatory exhibits.

Write the `@table` command at the beginning of a line and follow it on the same line with an argument that is a Texinfo command such as `@code`, `@samp`, `@var`, or `@kbd`. Although these commands are usually followed by arguments in braces, in this case you use the command name without an argument because `@item` will supply the argument. This command will be applied to the text that goes into the first column of each item and determines how it will be highlighted. For example, `@samp` will cause the text in the first column to be highlighted with an `@samp` command.

You may also choose to use the `@asis` command as an argument to `@table`. `@asis` is a command that does nothing; if you use this command after `@table`, TeX and the Info formatting commands output the first column entries without added highlighting (`as is').

(The `@table` command may work with other commands besides those listed here. However, you can only use commands that normally take arguments in braces.)

Begin each table entry with an `@item` command at the beginning of a line. Write the first column text on the same line as the `@item` command. Write the second column text on the line following the `@item` line and on subsequent lines. (You do not need to type anything for an empty second column entry.) You may write as many lines of supporting text as you wish, even several paragraphs. But only text on the same line as the `@item` will be placed in the first column.

Normally, you should put a blank line before an `@item` line. This puts a blank like in the Info file. Except when the entries are very brief, a blank line looks better.

The following table, for example, highlights the text in the first column with an `@samp` command:

```
@table @samp
@item foo
This is the text for
@samp{foo}.

@item bar
Text for @samp{bar}.
@end table
```

This produces:

`foo'
: This is the text for `foo'.

`bar'
: Text for `bar'.

If you want to list two or more named items with a single block of text, use the `@itemx` command. (See section @itemx.)

@ftable and @vtable

The `@ftable` and `@vtable` commands are the same as the `@table` command except that `@ftable` automatically enters each of the items in the first column of the table into the index of functions and `@vtable` automatically enters each of the items in the first column of the table into the index of variables. This simplifies the task of creating indices. Only the items on the same line as the `@item` commands are indexed, and they are indexed in exactly the form that they appear on that line. See section Creating Indices, for more information about indices.

Begin a two-column table using `@ftable` or `@vtable` by writing the @-command at the beginning of a line, followed on the same line by an argument that is a Texinfo command such as `@code`, exactly as you would for an `@table` command; and end the table with an `@end ftable` or `@end vtable` command on a line by itself.

@itemx

Use the `@itemx` command inside a table when you have two or more first column entries for the same item, each of which should appear on a line of its own. Use `@itemx` for all but the first entry. The `@itemx` command works exactly like `@item` except that it does not generate extra vertical space above the first column text.

For example,

```
@table @code
@item upcase
@itemx downcase
These two functions accept a character or a string as
argument, and return the corresponding upper case (lower
case) character or string.
@end table
```

This produces:

`upcase`

`downcase`

> These two functions accept a character or a string as argument, and return the corresponding upper case (lower case) character or string.

(Note also that this example illustrates multi-line supporting text in a two-column table.)

Go to the previous, next section.

Go to the previous, next section.

Creating Indices

Using Texinfo, you can generate indices without having to sort and collate entries manually. In an index, the entries are listed in alphabetical order, together with information on how to find the discussion of each entry. In a printed manual, this information consists of page numbers. In an Info file, this information is a menu entry leading to the first node referenced.

Texinfo provides several predefined kinds of index: an index for functions, an index for variables, an index for concepts, and so on. You can combine indices or use them for other than their canonical purpose. If you wish, you can define your own indices.

Making Index Entries

When you are making index entries, it is good practice to think of the different ways people may look for something. Different people *do not* think of the same words when they look something up. A helpful index will have items indexed under all the different words that people may use. For example, one reader may think it obvious that the two-letter names for indices should be listed under "Indices, two-letter names", since the word "Index" is the general concept. But another reader may remember the specific concept of two-letter names and search for the entry listed as "Two letter names for indices". A good index will have both entries and will help both readers.

Like typesetting, the construction of an index is a highly skilled, professional art, the subtleties of which are not appreciated until you need to do it yourself.

See section Index Menus and Printing an Index, for information about printing an index at the end of a book or creating an index menu in an Info file.

Predefined Indices

Texinfo provides six predefined indices:

- A concept index listing concepts that are discussed.
- A function index listing functions (such as entry points of libraries).
- A variables index listing variables (such as global variables of libraries).
- A keystroke index listing keyboard commands.
- A program index listing names of programs.
- A data type index listing data types (such as structures defined in header files).

Not every manual needs all of these, and most manuals use two or three of them. This manual has two indices: a concept index and an @-command index (that is actually the function index but is called a command index in the chapter heading). Two or more indices can be combined into one using the

`@synindex` or `@syncodeindex` commands. See section Combining Indices.

Defining the Entries of an Index

The data to make an index come from many individual indexing commands scattered throughout the Texinfo source file. Each command says to add one entry to a particular index; after formatting, the index will give the current page number or node name as the reference.

An index entry consists of an indexing command at the beginning of a line followed, on the rest of the line, by the entry.

For example, this section begins with the following five entries for the concept index:

```
@cindex Defining indexing entries
@cindex Index entries
@cindex Entries for an index
@cindex Specifying index entries
@cindex Creating index entries
```

Each predefined index has its own indexing command---`@cindex` for the concept index, `@findex` for the function index, and so on.

The usual convention is to capitalize the first word of each index entry, unless that word is the name of a function, variable, or other such entity that should not be capitalized. Thus, if you are documenting Emacs Lisp, you should usually capitalize entries in the concept index, but not those in the function index. However, if your concept index entries are consistently short (one or two words each) it may look better for each regular entry to start with a lower case letter. Whichever convention you adapt, please be consistent!

By default, entries for a concept index are printed in a small roman font and entries for the other indices are printed in a small `@code` font. You may change the way part of an entry is printed with the usual Texinfo commands, such as `@file` for file names and `@emph` for emphasis (see section Marking Words and Phrases).

The six indexing commands for predefined indices are:

`@cindex concept`
: Make an entry in the concept index for concept.

`@findex function`
: Make an entry in the function index for function.

`@vindex variable`
: Make an entry in the variable index for variable.

`@kindex keystroke`
: Make an entry in the key index for keystroke.

`@pindex program`

Make an entry in the program index for program.

`@tindex data type`

Make an entry in the data type index for data type.

Caution: Do not use a colon in an index entry. In Info, a colon separates the menu entry name from the node name. An extra colon confuses Info. See section The Parts of a Menu, for more information about the structure of a menu entry.

If you write several identical index entries in different places in a Texinfo file, the index in the printed manual will list all the pages to which those entries refer. However, the index in the Info file will list **only** the node that references the **first** of those index entries. Therefore, it is best to write indices in which each entry refers to only one place in the Texinfo file. Fortunately, this constraint is a feature rather than a loss since it means that the index will be easy to use. Otherwise, you could create an index that lists several pages for one entry and your reader would not know to which page to turn. If you have two identical entries for one topic, change the topics slightly, or qualify them to indicate the difference.

You are not actually required to use the predefined indices for their canonical purposes. For example, suppose you wish to index some C preprocessor macros. You could put them in the function index along with actual functions, just by writing `@findex` commands for them; then, when you print the "Function Index" as an unnumbered chapter, you could give it the title `Function and Macro Index' and all will be consistent for the reader. Or you could put the macros in with the data types by writing `@tindex` commands for them, and give that index a suitable title so the reader will understand. (See section Index Menus and Printing an Index.)

Combining Indices

Sometimes you will want to combine two disparate indices such as functions and concepts, perhaps because you have few enough of one of them that a separate index for them would look silly.

You could put functions into the concept index by writing `@cindex` commands for them instead of `@findex` commands, and produce a consistent manual by printing the concept index with the title `Function and Concept Index' and not printing the `Function Index' at all; but this is not a robust procedure. It works only if your document is never included as part of another document that is designed to have a separate function index; if your document were to be included with such a document, the functions from your document and those from the other would not end up together. Also, to make your function names appear in the right font in the concept index, you would need to enclose every one of them between the braces of `@code`.

`@syncodeindex`

When you want to combine functions and concepts into one index, you should index the functions with `@findex` and index the concepts with `@cindex`, and use the `@syncodeindex` command to redirect the function index entries into the concept index.

The `@syncodeindex` command takes two arguments; they are the name of the index to redirect, and the name of the index to redirect it to. The template looks like this:


```
@syncodeindex from to
```

For this purpose, the indices are given two-letter names:

`cp'
: concept index

`fn'
: function index

`vr'
: variable index

`ky'
: key index

`pg'
: program index

`tp'
: data type index

Write an `@syncodeindex` command before or shortly after the end-of-header line at the beginning of a Texinfo file. For example, to merge a function index with a concept index, write the following:

```
@syncodeindex fn cp
```

This will cause all entries designated for the function index to merge in with the concept index instead.

To merge both a variables index and a function index into a concept index, write the following:

```
@syncodeindex vr cp
@syncodeindex fn cp
```

The `@syncodeindex` command puts all the entries from the `from' index (the redirected index) into the `@code` font, overriding whatever default font is used by the index to which the entries are now directed. This way, if you direct function names from a function index into a concept index, all the function names are printed in the `@code` font as you would expect.

`@synindex`

The `@synindex` command is nearly the same as the `@syncodeindex` command, except that it does not put the `from' index entries into the `@code` font; rather it puts them in the roman font. Thus, you use `@synindex` when you merge a concept index into a function index.

See section Index Menus and Printing an Index, for information about printing an index at the end of a book or creating an index menu in an Info file.

Defining New Indices

In addition to the predefined indices, you may use the `@defindex` and `@defcodeindex` commands to define new indices. These commands create new indexing @-commands with which you mark index entries. The `@defindex` command is used like this:

```
@defindex name
```

The name of an index should be a two letter word, such as `au'. For example:

```
@defindex au
```

This defines a new index, called the `au' index. At the same time, it creates a new indexing command, `@auindex`, that you can use to make index entries. Use the new indexing command just as you would use a predefined indexing command.

For example, here is a section heading followed by a concept index entry and two `au' index entries.

```
@section Cognitive Semantics
@cindex kinesthetic image schemas
@auindex Johnson, Mark
@auindex Lakoff, George
```

(Evidently, `au' serves here as an abbreviation for "author".) Texinfo constructs the new indexing command by concatenating the name of the index with `index'; thus, defining an `au' index leads to the automatic creation of an `@auindex` command.

Use the `@printindex` command to print the index, as you do with the predefined indices. For example:

```
@node Author Index, Subject Index, , Top
@unnumbered Author Index

@printindex au
```

The `@defcodeindex` is like the `@defindex` command, except that, in the printed output, it prints entries in an `@code` font instead of a roman font. Thus, it parallels the `@findex` command rather than the `@cindex` command.

You should define new indices within or right after the end-of-header line of a Texinfo file, before any `@synindex` or `@syncodeindex` commands (see section The Texinfo File Header).

Go to the previous, next section.

Go to the previous, next section.

Special Insertions

Texinfo provides several commands for formatting dimensions, for inserting single characters that have special meaning in Texinfo, such as braces, and for inserting special graphic symbols that do not correspond to characters, such as dots and bullets.

These are:

- Braces, `@' and periods.
- Format a dimension, such as `12pt'.
- Dots and bullets.
- The TeX logo and the copyright symbol.
- A minus sign.

Inserting `@', Braces, and Periods

`@' and curly braces are special characters in Texinfo. To insert these characters so they appear in text, you must put an `@' in front of these characters to prevent Texinfo from misinterpreting them.

Periods are also special. Depending on whether the period is inside or at the end of a sentence, less or more space is inserted after a period in a typeset manual. Since it is not always possible for Texinfo to determine when a period ends a sentence and when it is used in an abbreviation, special commands are needed in some circumstances. (Usually, Texinfo can guess how to handle periods, so you do not need to use the special commands; you just enter a period as you would if you were using a typewriter, which means you put two spaces after the period, question mark, or exclamation mark that ends a sentence.)

Do not put braces after any of these commands; they are not necessary.

Inserting `@' with @@

@@ stands for a single `@' in either printed or Info output.

Do not put braces after an @@ command.

Inserting `{' and `}'with @{ and @}

@{ stands for a single `{' in either printed or Info output.

@} stands for a single `}' in either printed or Info output.

Do not put braces after either an @{ or an @} command.

TOC

Spacing After Colons and Periods

Use the `@:` command after a period, question mark, exclamation mark, or colon that should not be followed by extra space. For example, use `@:` after periods that end abbreviations which are not at the ends of sentences. `@:` has no effect on the Info file output.

For example,

```
The s.o.p.@: has three parts ...
The s.o.p. has three parts ...
```

produces the following. If you look carefully at this printed output, you will see a little more whitespace after `s.o.p.' in the second line.

> The s.o.p. has three parts ...
> The s.o.p. has three parts ...

@: has no effect on the Info output. (`s.o.p' is an acronym for "Standard Operating Procedure".)

Use `@.` instead of a period at the end of a sentence that ends with a single capital letter. Otherwise, TeX will think the letter is an abbreviation and will not insert the correct end-of-sentence spacing. Here is an example:

```
Give it to M.I.B. and to M.E.W@.  Also, give it to R.J.C@.
Give it to M.I.B. and to M.E.W.  Also, give it to R.J.C.
```

produces the following. If you look carefully at this printed output, you will see a little more whitespace after the `W' in the first line.

> Give it to M.I.B. and to M.E.W. Also, give it to R.J.C.
> Give it to M.I.B. and to M.E.W. Also, give it to R.J.C.

In the Info file output, `@.` is equivalent to a simple `.'.

The meanings of `@:` and `@.` in Texinfo are designed to work well with the Emacs sentence motion commands. This made it necessary for them to be incompatible with some other formatting systems that use @-commands.

Do not put braces after either an `@:` or an `@.` command.

@dmn{dimension}: Format a Dimension

At times, you may want to write `12pt' or `8.5in' with little or no space between the number and the abbreviation for the dimension. You can use the `@dmn` command to do this. On seeing the command, TeX inserts just enough space for proper typesetting; the Info formatting commands insert no space at all, since the Info file does not require it.

To use the `@dmn` command, write the number and then follow it immediately, with no intervening space, by `@dmn`, and then by the dimension within braces.

For example,

```
A4 paper is 8.27@dmn{in} wide.
```

produces

> A4 paper is 8.27in wide.

Not everyone uses this style. Instead of writing `8.27@dmn{in}' in the Texinfo file, you may write `8.27 in.' or `8.27 inches'. (In these cases, the formatters may insert a line break between the number and the dimension. Also, if you write a period after an abbreviation within a sentence, you should write `@:' after the period to prevent TeX from inserting extra whitespace. See section Spacing After Colons and Periods.)

Inserting Ellipsis, Dots, and Bullets

An ellipsis (a line of dots) is not typeset as a string of periods, so a special command is used for ellipsis in Texinfo. The `@bullet` command is special, too. Each of these commands is followed by a pair of braces, `{}', without any whitespace between the name of the command and the braces. (You need to use braces with these commands because you can use them next to other text; without the braces, the formatters would be confused. See section @-Command Syntax, for further information.)

@dots{}

Use the `@dots{}` command to generate an ellipsis, which is three dots in a row, appropriately spaced, like this: `...'. Do not simply write three periods in the input file; that would work for the Info file output, but would produce the wrong amount of space between the periods in the printed manual.

Here is an ellipsis: ...

Here are three periods in a row: ...

In printed output, the three periods in a row are closer together than the dots in the ellipsis.

@bullet{}

Use the `@bullet{}` command to generate a large round dot, or the closest possible thing to one. In Info, an asterisk is used.

Here is a bullet: *

When you use `@bullet` in `@itemize`, you do not need to type the braces, because `@itemize` supplies them. See section Making an Itemized List.

Inserting TeX and the Copyright Symbol

The logo `TeX' is typeset in a special fashion and it needs an @-command. The copyright symbol, `(C)', is also special. Each of these commands is followed by a pair of braces, `{}', without any whitespace between the name of the command and the braces.

@TeX{}

Use the `@TeX{}` command to generate `TeX'. In a printed manual, this is a special logo that is different from three ordinary letters. In Info, it just looks like `TeX'. The `@TeX{}` command is unique among Texinfo commands in that the T and the X are in upper case.

@copyright{}

Use the `@copyright{}` command to generate `(C)'. In a printed manual, this is a `c' inside a circle, and in Info, this is `(C)'.

@minus{}: Inserting a Minus Sign

Use the `@minus{}` command to generate a minus sign. In a fixed-width font, this is a single hyphen, but in a proportional font, the symbol is the customary length for a minus sign--a little longer than a hyphen.

You can compare the two forms:

```
`-' is a minus sign generated with `@minus{}',

`-' is a hyphen generated with the character `-'.
```

In the fixed-width font used by Info, `@minus{}` is the same as a hyphen.

You should not use `@minus{}` inside `@code` or `@example` because the width distinction is not made in the fixed-width font they use.

When you use `@minus` to specify the mark beginning each entry in an itemized list, you do not need to type the braces (see section Making an Itemized List).

Go to the previous, next section.

Go to the previous, next section.

Glyphs for Examples

In Texinfo, code is often illustrated in examples that are delimited by `@example` and `@end example`, or by `@lisp` and `@end lisp`. In such examples, you can indicate the results of evaluation or an expansion using `=>' or `==>'. Likewise, there are commands to insert glyphs to indicate printed output, error messages, equivalence of expressions, and the location of point.

The glyph-insertion commands do not need to be used within an example, but most often they are. Every glyph-insertion command is followed by a pair of left- and right-hand braces.

=>
: `@result{}` points to the result of an expression.

==>
: `@expansion{}` shows the results of a macro expansion.

-|
: `@print{}` indicates printed output.

error-->
: `@error{}` indicates that the following text is an error message.

==
: `@equiv{}` indicates the exact equivalence of two forms.

-!-
: `@point{}` shows the location of point.

=>: Indicating Evaluation

Use the `@result{}` command to indicate the result of evaluating an expression.

The `@result{}` command is displayed as `=>' in Info and as `=>' in the printed output.

Thus, the following,

```
(cdr '(1 2 3))
     => (2 3)
```

may be read as "`(cdr '(1 2 3))` evaluates to `(2 3)`".

==>: Indicating an Expansion

When an expression is a macro call, it expands into a new expression. You can indicate the result of the expansion with the `@expansion{}` command.

The `@expansion{}` command is displayed as `==>' in Info and as `==>' in the printed output.

For example, the following

```
@lisp
(third '(a b c))
     @expansion{} (car (cdr (cdr '(a b c))))
     @result{} c
@end lisp
```

produces

```
(third '(a b c))
     ==> (car (cdr (cdr '(a b c))))
     => c
```

which may be read as:

> `(third '(a b c))` expands to `(car (cdr (cdr '(a b c))))`; the result of evaluating the expression is `c`.

Often, as in this case, an example looks better if the `@expansion{}` and `@result{}` commands are indented five spaces.

-|: Indicating Printed Output

Sometimes an expression will print output during its execution. You can indicate the printed output with the `@print{}` command.

The `@print{}` command is displayed as `-|' in Info and as `-|' in the printed output.

In the following example, the printed text is indicated with `-|', and the value of the expression follows on the last line.

```
(progn (print 'foo) (print 'bar))
     -| foo
     -| bar
     => bar
```

In a Texinfo source file, this example is written as follows:

```
@lisp
```


```
(progn (print 'foo) (print 'bar))
     @print{} foo
     @print{} bar
     @result{} bar
@end lisp
```

error-->: Indicating an Error Message

A piece of code may cause an error when you evaluate it. You can designate the error message with the `@error{}` command.

The `@error{}` command is displayed as `error-->' in Info and as `error-->' in the printed output.

Thus,

```
@lisp
(+ 23 'x)
@error{} Wrong type argument: integer-or-marker-p, x
@end lisp
```

produces

```
(+ 23 'x)
error--> Wrong type argument: integer-or-marker-p, x
```

This indicates that the following error message is printed when you evaluate the expression:

```
Wrong type argument: integer-or-marker-p, x
```

Note that `error-->' itself is not part of the error message.

==: Indicating Equivalence

Sometimes two expressions produce identical results. You can indicate the exact equivalence of two forms with the `@equiv{}` command.

The `@equiv{}` command is displayed as `==' in Info and as `==' in the printed output.

Thus,

```
@lisp
(make-sparse-keymap) @equiv{} (list 'keymap)
@end lisp
```

produces


```
(make-sparse-keymap) == (list 'keymap)
```

This indicates that evaluating `(make-sparse-keymap)` produces identical results to evaluating `(list 'keymap)`.

Indicating Point in a Buffer

Sometimes you need to show an example of text in an Emacs buffer. In such examples, the convention is to include the entire contents of the buffer in question between two lines of dashes containing the buffer name.

You can use the `@point{}' command to show the location of point in the text in the buffer. (The symbol for point, of course, is not part of the text in the buffer; it indicates the place *between* two characters where point is located.)

The `@point{}` command is displayed as `-!-' in Info and as `-!-' in the printed output.

The following example shows the contents of buffer `` `foo' `` before and after evaluating a Lisp command to insert the word `changed`.

```
---------- Buffer: foo ----------
This is the -!-contents of foo.
---------- Buffer: foo ----------


(insert "changed ")
     => nil
---------- Buffer: foo ----------
This is the changed -!-contents of foo.
---------- Buffer: foo ----------
```

In a Texinfo source file, the example is written like this:

```
@example
---------- Buffer: foo ----------
This is the @point{}contents of foo.
---------- Buffer: foo ----------

(insert "changed ")
     @result{} nil
---------- Buffer: foo ----------
This is the changed @point{}contents of foo.
---------- Buffer: foo ----------
@end example
```

TOC

Go to the previous, next section.

Go to the previous, next section.

Making and Preventing Breaks

Usually, a Texinfo file is processed both by TeX and by one of the Info formatting commands. Line, paragraph, or page breaks sometimes occur in the `wrong' place in one or other form of output. You must ensure that text looks right both in the printed manual and in the Info file.

For example, in a printed manual, page breaks may occur awkwardly in the middle of an example; to prevent this, you can hold text together using a grouping command that keeps the text from being split across two pages. Conversely, you may want to force a page break where none would occur normally. Fortunately, problems like these do not often arise. When they do, use the break, break prevention, or pagination commands.

The break commands create line and paragraph breaks:

`@*`
: Force a line break.

`@sp n`
: Skip n blank lines.

The line-break-prevention command holds text together all on one line:

`@w{text}`
: Prevent text from being split and hyphenated across two lines.

The pagination commands apply only to printed output, since Info files do not have pages.

`@page`
: Start a new page in the printed manual.

`@group`
: Hold text together that must appear on one printed page.

`@need mils`
: Start a new printed page if not enough space on this one.

@*: Generate Line Breaks

The `@*` command forces a line break in both the printed manual and in Info.

For example,

```
This line @* is broken @*in two places.
```

produces


```
This line
 is broken
in two places.
```

(Note that the space after the first `@*` command is faithfully carried down to the next line.)

The `@*` command is often used in a file's copyright page:

```
This is edition 2.0 of the Texinfo documentation,@*
and is for ...
```

In this case, the `@*` command keeps TeX from stretching the line across the whole page in an ugly manner.

> **Please note:** Do not write braces after an `@*` command; they are not needed.
>
> Do not write an `@refill` command at the end of a paragraph containing an `@*` command; it will cause the paragraph to be refilled after the line break occurs, negating the effect of the line break.

@w{text}: Prevent Line Breaks

`@w{text}` outputs text and prohibits line breaks within text.

You can use the `@w` command to prevent TeX from automatically hyphenating a long name or phrase that accidentally falls near the end of a line.

```
You can copy GNU software from @w{@file{prep.ai.mit.edu}}.
```

produces

> You can copy GNU software from `` `prep.ai.mit.edu' ``.

In the Texinfo file, you must write the `@w` command and its argument (all the affected text) all on one line.

> **Caution:** Do not write an `@refill` command at the end of a paragraph containing an `@w` command; it will cause the paragraph to be refilled and may thereby negate the effect of the `@w` command.

@sp n: Insert Blank Lines

A line beginning with and containing only `@sp n` generates n blank lines of space in both the printed manual and the Info file. `@sp` also forces a paragraph break. For example,

```
@sp 2
```

generates two blank lines.

The `@sp` command is most often used in the title page.

`@page`: Start a New Page

A line containing only `@page` starts a new page in a printed manual. The command has no effect on Info files since they are not paginated. An `@page` command is often used in the `@titlepage` section of a Texinfo file to start the copyright page.

`@group`: Prevent Page Breaks

The `@group` command (on a line by itself) is used inside an `@example` or similar construct to begin an unsplittable vertical group, which will appear entirely on one page in the printed output. The group is terminated by a line containing only `@end group`. These two lines produce no output of their own, and in the Info file output they have no effect at all.

Although `@group` would make sense conceptually in a wide variety of contexts, its current implementation works reliably only within `@example` and variants, and within `@display`, `@format`, `@flushleft` and `@flushright`. See section Quotations and Examples. (What all these commands have in common is that each line of input produces a line of output.) In other contexts, `@group` can cause anomalous vertical spacing.

This formatting requirement means that you should write:

```
@example
@group
...
@end group
@end example
```

with the `@group` and `@end group` commands inside the `@example` and `@end example` commands.

The `@group` command is most often used to hold an example together on one page. In this Texinfo manual, more than 100 examples contain text that is enclosed between `@group` and `@end group`.

If you forget to end a group, you may get strange and unfathomable error messages when you run TeX. This is because TeX keeps trying to put the rest of the Texinfo file onto the one page and does not start to generate error messages until it has processed considerable text. It is a good rule of thumb to look for a missing `@end group` if you get incomprehensible error messages in TeX.

`@need mils`: Prevent Page Breaks

A line containing only `@need n` starts a new page in a printed manual if fewer than n mils (thousandths of an inch) remain on the current page. Do not use braces around the argument n. The `@need` command has no effect on Info files since they are not paginated.

This paragraph is preceded by an `@need` command that tells TeX to start a new page if fewer than 8 mils (eight-tenths inch) remain on the page. It looks like this:

```
@need 800
This paragraph is preceded by ...
```

The `@need` command is useful for preventing orphans (single lines at the bottoms of printed pages).

Go to the previous, next section.

Go to the previous, next section.

Definition Commands

The `@deffn` command and the other definition commands enable you to describe functions, variables, macros, commands, user options, special forms and other such artifacts in a uniform format.

In the Info file, a definition causes the entity category---`Function', `Variable', or whatever--to appear at the beginning of the first line of the definition, followed by the entity's name and arguments. In the printed manual, the command causes TeX to print the entity's name and its arguments on the left margin and print the category next to the right margin. In both output formats, the body of the definition is indented. Also, the name of the entity is entered into the appropriate index: `@deffn` enters the name into the index of functions, `@defvr` enters it into the index of variables, and so on.

A manual need not and should not contain more than one definition for a given name. An appendix containing a summary should use `@table` rather than the definition commands.

The Template for a Definition

The `@deffn` command is used for definitions of entities that resemble functions. To write a definition using the `@deffn` command, write the `@deffn` command at the beginning of a line and follow it on the same line by the category of the entity, the name of the entity itself, and its arguments (if any). Then write the body of the definition on succeeding lines. (You may embed examples in the body.) Finally, end the definition with an `@end deffn` command written on a line of its own. (The other definition commands follow the same format.)

The template for a definition looks like this:

```
@deffn category name arguments...
body-of-definition
@end deffn
```

For example,

```
@deffn Command forward-word count
This command moves point forward @var{count} words
(or backward if @var{count} is negative). ...
@end deffn
```

produces

> Command: **forward-word** *count*
>
> This function moves point forward count words (or backward if count is negative). ...

Capitalize the category name like a title. If the name of the category contains spaces, as in the phrase

`Interactive Command', write braces around it. For example:

```
@deffn {Interactive Command} isearch-forward
...
@end deffn
```

Otherwise, the second word will be mistaken for the name of the entity.

Some of the definition commands are more general than others. The `@deffn` command, for example, is the general definition command for functions and the like--for entities that may take arguments. When you use this command, you specify the category to which the entity belongs. The `@deffn` command possesses three predefined, specialized variations, `@defun`, `@defmac`, and `@defspec`, that specify the category for you: "Function", "Macro", and "Special Form" respectively. The `@defvr` command also is accompanied by several predefined, specialized variations for describing particular kinds of variables.

The template for a specialized definition, such as `@defun`, is similar to the template for a generalized definition, except that you do not need to specify the category:

```
@defun name arguments...
body-of-definition
@end defun
```

Thus,

```
@defun buffer-end flag
This function returns @code{(point-min)} if @var{flag}
is less than 1, @code{(point-max)} otherwise.
...
@end defun
```

produces

> Function: **buffer-end** *flag*
>
> This function returns `(point-min)` if flag is less than 1, `(point-max)` otherwise. ...

See section A Sample Function Definition, for a more detailed example of a function definition, including the use of `@example` inside the definition.

The other specialized commands work like `@defun`.

Optional and Repeated Arguments

Some entities take optional or repeated arguments, which may be specified by a distinctive glyph that uses square brackets and ellipses. For example, a special form often breaks its argument list into separate arguments in more complicated ways than a straightforward function.

An argument enclosed within square brackets is optional. Thus, the phrase `[optional-arg]' means

that optional-arg is optional. An argument followed by an ellipsis is optional and may be repeated more than once. Thus, `repeated-args...' stands for zero or more arguments. Parentheses are used when several arguments are grouped into additional levels of list structure in Lisp.

Here is the `@defspec` line of an example of an imaginary special form:

> Special Form: **foobar** *(var [from to [inc]]) body...*

In this example, the arguments from and to are optional, but must both be present or both absent. If they are present, inc may optionally be specified as well. These arguments are grouped with the argument var into a list, to distinguish them from body, which includes all remaining elements of the form.

In a Texinfo source file, this `@defspec` line is written like this (except it would not be split over two lines, as it is in this example).

```
@defspec foobar (@var{var} [@var{from} @var{to}
     [@var{inc}]]) @var{body}@dots{}
```

The function is listed in the Command and Variable Index under `foobar'.

Two or More `First' Lines

To create two or more `first' or header lines for a definition, follow the first `@deffn` line by a line beginning with `@deffnx`. The `@deffnx` command works exactly like `@deffn` except that it does not generate extra vertical white space between it and the preceding line.

For example,

```
@deffn {Interactive Command} isearch-forward
@deffnx {Interactive Command} isearch-backward
These two search commands are similar except ...
@end deffn
```

produces

Interactive Command: **isearch-forward**

Interactive Command: **isearch-backward**

These two search commands are similar except ...

Each of the other definition commands has an `x' form: `@defunx`, `@defvrx`, `@deftypefunx`, etc.

The `x' forms work just like `@itemx`; see section `@itemx`.

The Definition Commands

Texinfo provides more than a dozen definition commands, all of which are described in this section.

The definition commands automatically enter the name of the entity in the appropriate index: for example, `@deffn`, `@defun`, and `@defmac` enter function names in the index of functions; `@defvr` and `@defvar` enter variable names in the index of variables.

Although the examples that follow mostly illustrate Lisp, the commands can be used for other programming languages.

Functions and Similar Entities

This section describes the commands for describing functions and similar entities:

`@deffn category name arguments...`

The `@deffn` command is the general definition command for functions, interactive commands, and similar entities that may take arguments. You must choose a term to describe the category of entity being defined; for example, "Function" could be used if the entity is a function. The `@deffn` command is written at the beginning of a line and is followed on the same line by the category of entity being described, the name of this particular entity, and its arguments, if any. Terminate the definition with `@end deffn` on a line of its own.

For example, here is a definition:

```
@deffn Command forward-char nchars
Move point forward @var{nchars} characters.
@end deffn
```

This shows a rather terse definition for a "command" named `forward-char` with one argument, nchars.

`@deffn` prints argument names such as nchars in italics or upper case, as if `@var` had been used, because we think of these names as metasyntactic variables--they stand for the actual argument values. Within the text of the description, write an argument name explicitly with `@var` to refer to the value of the argument. In the example above, we used `@var{nchars}' in this way.

The template for `@deffn` is:

```
@deffn category name arguments...
body-of-definition
@end deffn
```

`@defun name arguments...`

The `@defun` command is the definition command for functions. `@defun` is equivalent to `@deffn Function ...'.

For example,


```
@defun set symbol new-value
Change the value of the symbol @var{symbol}
to @var{new-value}.
@end defun
```

shows a rather terse definition for a function `set` whose arguments are symbol and new-value. The argument names on the `@defun` line automatically appear in italics or upper case as if they were enclosed in `@var`. Terminate the definition with `@end defun` on a line of its own.

The template is:

```
@defun function-name arguments...
body-of-definition
@end defun
```

`@defun` creates an entry in the index of functions.

`@defmac name arguments...`

The `@defmac` command is the definition command for macros. `@defmac` is equivalent to `@deffn Macro ...' and works like `@defun`.

`@defspec name arguments...`

The `@defspec` command is the definition command for special forms. (In Lisp, a special form is an entity much like a function.) `@defspec` is equivalent to `@deffn {Special Form} ...' and works like `@defun`.

Variables and Similar Entities

Here are the commands for defining variables and similar entities:

`@defvr category name`

The `@defvr` command is a general definition command for something like a variable--an entity that records a value. You must choose a term to describe the category of entity being defined; for example, "Variable" could be used if the entity is a variable. Write the `@defvr` command at the beginning of a line and followed it on the same line by the category of the entity and the name of the entity.

Capitalize the category name like a title. If the name of the category contains spaces, as in the name `User Option', write braces around it. Otherwise, the second word will be mistaken for the name of the entity, for example:

```
@defvr {User Option} fill-column
This buffer-local variable specifies
the maximum width of filled lines.
...
@end defvr
```

Terminate the definition with `@end defvr` on a line of its own.

The template is:

```
@defvr category name
body-of-definition
@end defvr
```

`@defvr` creates an entry in the index of variables for name.

`@defvar name`

The `@defvar` command is the definition command for variables. `@defvar` is equivalent to `@defvr Variable ...'.

For example:

```
@defvar kill-ring
...
@end defvar
```

The template is:

```
@defvar name
body-of-definition
@end defvar
```

`@defvar` creates an entry in the index of variables for name.

`@defopt name`

The `@defopt` command is the definition command for user options. `@defopt` is equivalent to `@defvr {User Option} ...' and works like `@defvar`.

Functions in Typed Languages

The `@deftypefn` command and its variations are for describing functions in C or any other language in which you must declare types of variables and functions.

`@deftypefn category data-type name arguments...`

The `@deftypefn` command is the general definition command for functions and similar entities that may take arguments and that are typed. The `@deftypefn` command is written at the beginning of a line and is followed on the same line by the category of entity being described, the type of the returned value, the name of this particular entity, and its arguments, if any.

For example,

```
@deftypefn {Library Function} int foobar
   (int @var{foo}, float @var{bar})
...
```


```
@end deftypefn
```

(where the text before the "...", shown above as two lines, would actually be a single line in a real Texinfo file) produces the following in Info:

```
-- Library Function: int foobar (int FOO, float BAR)
...
```

In a printed manual, it produces:

> Library Function: int **foobar** *(int foo, float bar)*
>
> ...

This means that `foobar` is a "library function" that returns an `int`, and its arguments are foo (an `int`) and bar (a `float`).

The argument names that you write in `@deftypefn` are not subject to an implicit `@var`---since the actual names of the arguments in `@deftypefn` are typically scattered among data type names and keywords, Texinfo cannot find them without help. Instead, you must write `@var` explicitly around the argument names. In the example above, the argument names are `foo' and `bar'.

The template for `@deftypefn` is:

```
@deftypefn category data-type name arguments ...
body-of-description
@end deftypefn
```

Note that if the category or data type is more than one word then it must be enclosed in braces to make it a single argument.

If you are describing a procedure in a language that has packages, such as Ada, you might consider using `@deftypefn` in a manner somewhat contrary to the convention described in the preceding paragraphs.

For example:

```
@deftypefn stacks private push
        (@var{s}:in out stack;
        @var{n}:in integer)
...
@end deftypefn
```

(The `@deftypefn` arguments are shown split into three lines, but would be a single line in a real Texinfo file.)

In this instance, the procedure is classified as belonging to the package `stacks` rather than classified as a `procedure' and its data type is described as `private`. (The name of the procedure is `push`, and its arguments are s and n.)

`@deftypefn` creates an entry in the index of functions for name.

`@deftypefun data-type name arguments...`

The `@deftypefun` command is the specialized definition command for functions in typed languages. The command is equivalent to `@deftypefn Function ...'.

Thus,

```
@deftypefun int foobar (int @var{foo}, float @var{bar})
...
@end deftypefun
```

produces the following in Info:

```
-- Function: int foobar (int FOO, float BAR)
...
```

and the following in a printed manual:

Function: int **foobar** *(int foo, float bar)*

...

The template is:

```
@deftypefun type name arguments...
body-of-description
@end deftypefun
```

`@deftypefun` creates an entry in the index of functions for name.

Variables in Typed Languages

Variables in typed languages are handled in a manner similar to functions in typed languages. See section Functions in Typed Languages. The general definition command `@deftypevr` corresponds to `@deftypefn` and the specialized definition command `@deftypevar` corresponds to `@deftypefun`.

`@deftypevr category data-type name`

The `@deftypevr` command is the general definition command for something like a variable in a typed language--an entity that records a value. You must choose a term to describe the category of the entity being defined; for example, "Variable" could be used if the entity is a variable.

The `@deftypevr` command is written at the beginning of a line and is followed on the same line by the category of the entity being described, the data type, and the name of this particular entity.

For example:

```
@deftypevr {Global Flag} int enable
...
```


```
@end deftypevr
```

produces the following in Info:

```
-- Global Flag: int enable
...
```

and the following in a printed manual:

Global Flag: int **enable**

...

The template is:

```
@deftypevr category data-type name
body-of-description
@end deftypevr
```

`@deftypevr` creates an entry in the index of variables for name.

`@deftypevar data-type name`

The `@deftypevar` command is the specialized definition command for variables in typed languages. `@deftypevar` is equivalent to `@deftypevr Variable ...'.

For example:

```
@deftypevar int fubar
...
@end deftypevar
```

produces the following in Info:

```
-- Variable: int fubar
...
```

and the following in a printed manual:

Variable: int **fubar**

...

The template is:

```
@deftypevar data-type name
body-of-description
@end deftypevar
```

`@deftypevar` creates an entry in the index of variables for name.

Object-Oriented Programming

Here are the commands for formatting descriptions about abstract objects, such as are used in object-oriented programming. A class is a defined type of abstract object. An instance of a class is a particular object that has the type of the class. An instance variable is a variable that belongs to the class but for which each instance has its own value.

In a definition, if the name of a class is truly a name defined in the programming system for a class, then you should write an `@code` around it. Otherwise, it is printed in the usual text font.

`@defcv category class name`

The `@defcv` command is the general definition command for variables associated with classes in object-oriented programming. The `@defcv` command is followed by three arguments: the category of thing being defined, the class to which it belongs, and its name. Thus,

```
@defcv {Class Option} Window border-pattern
...
@end defcv
```

illustrates how you would write the first line of a definition of the `border-pattern` class option of the class `Window`.

The template is

```
@defcv category class name
...
@end defcv
```

`@defcv` creates an entry in the index of variables.

`@defivar class name`

The `@defivar` command is the definition command for instance variables in object-oriented programming. `@defivar` is equivalent to `@defcv {Instance Variable} ...'

The template is:

```
@defivar class instance-variable-name
body-of-definition
@end defivar
```

`@defivar` creates an entry in the index of variables.

`@defop category class name arguments...`

The `@defop` command is the general definition command for entities that may resemble methods in object-oriented programming. These entities take arguments, as functions do, but are associated with particular classes of objects.

For example, some systems have constructs called wrappers that are associated with classes as methods are, but that act more like macros than like functions. You could use `@defop Wrapper`

to describe one of these.

Sometimes it is useful to distinguish methods and operations. You can think of an operation as the specification for a method. Thus, a window system might specify that all window classes have a method named `expose`; we would say that this window system defines an `expose` operation on windows in general. Typically, the operation has a name and also specifies the pattern of arguments; all methods that implement the operation must accept the same arguments, since applications that use the operation do so without knowing which method will implement it.

Often it makes more sense to document operations than methods. For example, window application developers need to know about the `expose` operation, but need not be concerned with whether a given class of windows has its own method to implement this operation. To describe this operation, you would write:

```
@defop Operation windows expose
```

The `@defop` command is written at the beginning of a line and is followed on the same line by the overall name of the category of operation, the name of the class of the operation, the name of the operation, and its arguments, if any.

The template is:

```
@defop category class name arguments...
body-of-definition
@end defop
```

`@defop` creates an entry, such as `` `expose on windows' ``, in the index of functions.

`@defmethod class name arguments...`

The `@defmethod` command is the definition command for methods in object-oriented programming. A method is a kind of function that implements an operation for a particular class of objects and its subclasses. In the Lisp Machine, methods actually were functions, but they were usually defined with `defmethod`.

`@defmethod` is equivalent to `` `@defop Method ...' ``. The command is written at the beginning of a line and is followed by the name of the class of the method, the name of the method, and its arguments, if any.

For example,

```
@defmethod bar-class bar-method argument
...
@end defmethod
```

illustrates the definition for a method called `bar-method` of the class `bar-class`. The method takes an argument.

The template is:


```
@defmethod class method-name arguments...
body-of-definition
@end defmethod
```

`@defmethod` creates an entry in the index of functions, such as `` `bar-method on bar-class' ``.

Data Types

Here is the command for data types:

`@deftp category name attributes...`

The `@deftp` command is the generic definition command for data types. The command is written at the beginning of a line and is followed on the same line by the category, by the name of the type (which is a word like `int` or `float`), and then by names of attributes of objects of that type. Thus, you could use this command for describing `int` or `float`, in which case you could use `data type` as the category. (A data type is a category of certain objects for purposes of deciding which operations can be performed on them.)

In Lisp, for example, pair names a particular data type, and an object of that type has two slots called the CAR and the CDR. Here is how you would write the first line of a definition of `pair`.

```
@deftp {Data type} pair car cdr
...
@end deftp
```

The template is:

```
@deftp category name-of-type attributes...
body-of-definition
@end deftp
```

`@deftp` creates an entry in the index of data types.

Conventions for Writing Definitions

When you write a definition using `@deffn`, `@defun`, or one of the other definition commands, please take care to use arguments that indicate the meaning, as with the count argument to the `forward-word` function. Also, if the name of an argument contains the name of a type, such as integer, take care that the argument actually is of that type.

A Sample Function Definition

A function definition uses the `@defun` and `@end defun` commands. The name of the function follows immediately after the `@defun` command and it is followed, on the same line, by the parameter list.

Here is a definition from The GNU Emacs Lisp Reference Manual. (See section 'Calling Functions' in The GNU Emacs Lisp Reference Manual.)

> Function: **apply** *function &rest arguments*
>
> `apply` calls function with arguments, just like `funcall` but with one difference: the last of arguments is a list of arguments to give to function, rather than a single argument. We also say that this list is appended to the other arguments.
>
> `apply` returns the result of calling function. As with `funcall`, function must either be a Lisp function or a primitive function; special forms and macros do not make sense in `apply`.
>
> ```
> (setq f 'list)
>      => list
> (apply f 'x 'y 'z)
> error--> Wrong type argument: listp, z
> (apply '+ 1 2 '(3 4))
>      => 10
> (apply '+ '(1 2 3 4))
>      => 10
>
> (apply 'append '((a b c) nil (x y z) nil))
>      => (a b c x y z)
> ```
>
> An interesting example of using `apply` is found in the description of `mapcar`.

In the Texinfo source file, this example looks like this:

```
@defun apply function &rest arguments

@code{apply} calls @var{function} with
@var{arguments}, just like @code{funcall} but with one
difference: the last of @var{arguments} is a list of
arguments to give to @var{function}, rather than a single
argument.  We also say that this list is @dfn{appended}
to the other arguments.

@code{apply} returns the result of calling
@var{function}.  As with @code{funcall},
@var{function} must either be a Lisp function or a
primitive function; special forms and macros do not make
```


```
sense in @code{apply}.

@example
(setq f 'list)
     @result{} list
(apply f 'x 'y 'z)
@error{} Wrong type argument: listp, z
(apply '+ 1 2 '(3 4))
     @result{} 10
(apply '+ '(1 2 3 4))
     @result{} 10

(apply 'append '((a b c) nil (x y z) nil))
     @result{} (a b c x y z)
@end example

An interesting example of using @code{apply} is found
in the description of @code{mapcar}.@refill
@end defun
```

In this manual, this function is listed in the Command and Variable Index under `apply`.

Ordinary variables and user options are described using a format like that for functions except that variables do not take arguments.

Go to the previous, next section.

Go to the previous, next section.

Footnotes

A footnote is for a reference that documents or elucidates the primary text.(8)

In Texinfo, footnotes are created with the `@footnote` command. This command is followed immediately by a left brace, then by the text of the footnote, and then by a terminating right brace. The template is:

```
@footnote{text}
```

Footnotes may be of any length, but are usually short.

For example, this clause is followed by a sample footnote(9); in the Texinfo source, it looks like this:

```
...a sample footnote @footnote{Here is the sample
footnote.}; in the Texinfo source...
```

In a printed manual or book, the reference mark for a footnote is a small, superscripted number; the text of the footnote is written at the bottom of the page, below a horizontal line.

In Info, the reference mark for a footnote is a pair of parentheses with the footnote number between them, like this: `(1)'.

Info has two footnote styles, which determine where the text of the footnote is located:

- In the `End' node style, all the footnotes for a single node are placed at the end of that node. The footnotes are separated from the rest of the node by a line of dashes with the word `Footnotes' within it. Each footnote begins with an `(n)' reference mark.

  Here is an example of a single footnote in the end of node style:

  ```
  --------- Footnotes ---------

  (1)  Here is a sample footnote.
  ```

- In the `Separate' node style, all the footnotes for a single node are placed in an automatically constructed node of their own. In this style, a "footnote reference" follows each `(n)' reference mark in the body of the node. The footnote reference is actually a cross reference which you use to reach the footnote node.

  The name of the node containing the footnotes is constructed by appending `-Footnotes' to the name of the node that contains the footnotes. (Consequently, the footnotes' node for the `Footnotes' node is `Footnotes-Footnotes'!) The footnotes' node has an `Up' node pointer that leads back to its parent node.

  Here is how the first footnote in this manual looks after being formatted for Info in the separate

node style:

```
File: texinfo.info  Node: Overview-Footnotes, Up: Overview

(1) Note that the first syllable of "Texinfo" is
pronounced like "speck", not "hex". ...
```

A Texinfo file may be formatted into an Info file with either footnote style.

Use the `@footnotestyle` command to specify an Info file's footnote style. Write this command at the beginning of a line followed by an argument, either `end' for the end node style or `separate' for the separate node style.

For example,

```
@footnotestyle end
```

or

```
@footnotestyle separate
```

Write an `@footnotestyle` command before or shortly after the end-of-header line at the beginning of a Texinfo file. (If you include the `@footnotestyle` command between the start-of-header and end-of-header lines, the region formatting commands will format footnotes as specified.)

If you do not specify a footnote style, the formatting commands use their default style. Currently, `makeinfo` uses the `end' style, while `texinfo-format-buffer` and `texinfo-format-region` use the `separate' style.

This chapter contains two footnotes.

Go to the previous, next section.

Go to the previous, next section.

Conditionally Visible Text

Sometimes it is good to use different text for a printed manual and its corresponding Info file. In this case, you can use the conditional commands to specify which text is for the printed manual and which is for the Info file.

`@ifinfo` begins segments of text that should be ignored by TeX when it typesets the printed manual. The segment of text appears only in the Info file. The `@ifinfo` command should appear on a line by itself; end the Info-only text with a line containing `@end ifinfo` by itself. At the beginning of a Texinfo file, the Info permissions are contained within a region marked by `@ifinfo` and `@end ifinfo`. (See section Summary and Copying Permissions for Info.)

The `@iftex` and `@end iftex` commands are similar to the `@ifinfo` and `@end ifinfo` commands, except that they specify text that will appear in the printed manual but not in the Info file.

For example,

```
@iftex
This text will appear only in the printed manual.
@end iftex

@ifinfo
However, this text will appear only in Info.
@end ifinfo
```

The preceding example produces the following line:

This text will appear only in the printed manual.

Note how you only see one of the two lines, depending on whether you are reading the Info version or the printed version of this manual.

The `@titlepage` command is a special variant of `@iftex` that is used for making the title and copyright pages of the printed manual. (See section `@titlepage`.)

Using Ordinary TeX Commands

Inside a region delineated by `@iftex` and `@end iftex`, you can embed some PlainTeX commands. Info will ignore these commands since they are only in that part of the file which is seen by TeX. You can write the TeX commands as you would write them in a normal TeX file, except that you must replace the `\' used by TeX with an `@'. For example, in the `@titlepage` section of a Texinfo file, you can use the TeX command `@vskip` to format the copyright page. (The `@titlepage` command causes Info to ignore the region automatically, as it does with the `@iftex` command.)

However, many features of PlainTeX will not work, as they are overridden by features of Texinfo.

You can enter PlainTeX completely, and use `\' in the TeX commands, by delineating a region with the `@tex` and `@end tex` commands. (The `@tex` command also causes Info to ignore the region, like the `@iftex` command.)

For example, here is a mathematical expression written in PlainTeX:

```
@tex
$$ \chi^2 = \sum_{i=1}^N
          \left (y_i - (a + b x_i)
          \over \sigma_i\right)^2 $$
@end tex
```

The output of this example will appear only in a printed manual. If you are reading this in Info, you will not see anything after this paragraph. In a printed manual, the above expression looks like this:

`@set`, `@clear`, and `@value`

You can direct the Texinfo formatting commands to format or ignore parts of a Texinfo file with the `@set`, `@clear`, `@ifset`, and `@ifclear` commands.

In addition, you can use the `@set flag` command to set the value of flag to a string of characters; and use `@value{flag}` to insert that string. You can use `@set`, for example, to set a date and use `@value` to insert the date in several places in the Texinfo file.

`@ifset` and `@ifclear`

When a flag is set, the Texinfo formatting commands format text between subsequent pairs of `@ifset flag` and `@end ifset` commands. When the flag is cleared, the Texinfo formatting commands do *not* format the text.

Use the `@set flag` command to turn on, or set, a flag; a flag can be any single word. The format for the command looks like this:

```
@set flag
```

Write the conditionally formatted text between `@ifset flag` and `@end ifset` commands, like this:

```
@ifset flag
conditional-text
@end ifset
```

For example, you can create one document that has two variants, such as a manual for a `large' and `small' model:


```
You can use this machine to dig up shrubs
without hurting them.

@set large

@ifset large
It can also dig up fully grown trees.
@end ifset

Remember to replant promptly ...
```

In the example, the formatting commands will format the text between `@ifset large` and `@end ifset` because the `large` flag is set.

Use the `@clear flag` command to turn off, or clear, a flag. Clearing a flag is the opposite of setting a flag. The command looks like this:

```
@clear flag
```

Write the command on a line of its own.

When flag is cleared, the Texinfo formatting commands do *not* format the text between `@ifset flag` and `@end ifset`; that text is ignored and does not appear in either printed or Info output.

For example, if you clear the flag of the preceding example by writing an `@clear large` command after the `@set large` command (but before the conditional text), then the Texinfo formatting commands ignore the text between the `@ifset large` and `@end ifset` commands. In the formatted output, that text does not appear; in both printed and Info output, you see only the lines that say, "You can use this machine to dig up shrubs without hurting them. Remember to replant promptly ...".

If a flag is cleared with an `@clear flag` command, then the formatting commands format text between subsequent pairs of `@ifclear` and `@end ifclear` commands. But if the flag is set with `@set flag`, then the formatting commands do *not* format text between an `@ifclear` and an `@end ifclear` command; rather, they ignore that text. An `@ifclear` command looks like this:

```
@ifclear flag
```

In brief, the commands are:

`@set flag`
: Tell the Texinfo formatting commands that flag is set.

`@clear flag`
: Tell the Texinfo formatting commands that flag is cleared.

`@ifset flag`
: If flag is set, tell the Texinfo formatting commands to format the text up to the following `@end ifset` command.

If flag is cleared, tell the Texinfo formatting commands to ignore text up to the following `@end ifset` command.

`@ifclear flag`

If flag is set, tell the Texinfo formatting commands to ignore the text up to the following `@end ifclear` command.

If flag is cleared, tell the Texinfo formatting commands to format the text up to the following `@end ifclear` command.

@value

You can use the `@set` command to specify a value for a flag, which is expanded by the `@value` command. The value is a string a characters.

Write the `@set` command like this:

```
@set foo This is a string.
```

This sets the value of `foo` to "This is a string."

The Texinfo formatters replace an `@value{flag}` command with the string to which flag is set.

Thus, when `foo` is set as shown above, the Texinfo formatters convert

```
@value{foo}
to
This is a string.
```

You can write an `@value` command within a paragraph; but you must write an `@set` command on a line of its own.

If you write the `@set` command like this:

```
@set foo
```

without specifying a string, the value of `foo` is an empty string.

If you clear a previously set flag with an `@clear flag` command, a subsequent `@value{flag}` command is invalid and the string is replaced with an error message that says `{No value for "flag"}'.

For example, if you set `foo` as follows:

```
@set how-much very, very, very
```

then the formatters transform

```
It is a @value{how-much} wet day.
into
```


```
It is a very, very, very wet day.
```

If you write

```
@clear how-much
```

then the formatters transform

```
It is a @value{how-much} wet day.
into
It is a {No value for "how-much"} wet day.
```

@value Example

You can use the `@value` command to limit the number of places you need to change when you record an update to a manual. Here is how it is done in The GNU Make Manual:

Set the flags:

```
@set EDITION 0.35 Beta
@set VERSION 3.63 Beta
@set UPDATED 14 August 1992
@set UPDATE-MONTH August 1992
```

Write text for the first `@ifinfo` section, for people reading the Texinfo file:

```
This is Edition @value{EDITION},
last updated @value{UPDATED},
of @cite{The GNU Make Manual},
for @code{make}, Version @value{VERSION}.
```

Write text for the title page, for people reading the printed manual:

```
@title GNU Make
@subtitle A Program for Directing Recompilation
@subtitle Edition @value{EDITION}, ...
@subtitle @value{UPDATE-MONTH}
```

(On a printed cover, a date listing the month and the year looks less fussy than a date listing the day as well as the month and year.)

Write text for the Top node, for people reading the Info file:

```
This is Edition @value{EDITION}
of the @cite{GNU Make Manual},
last updated @value{UPDATED}
```


```
for @code{make} Version @value{VERSION}.
```

After you format the manual, the text in the first `@ifinfo` section looks like this:

```
This is Edition 0.35 Beta, last updated 14 August 1992,
of `The GNU Make Manual', for `make', Version 3.63 Beta.
```

When you update the manual, change only the values of the flags; you do not need to rewrite the three sections.

Go to the previous, next section.

Go to the previous, next section.

Format and Print Hardcopy

There are three major shell commands for making a printed manual from a Texinfo file: one for converting the Texinfo file into a file that will be printed, a second for sorting indices, and a third for printing the formatted document. When you use the shell commands, you can either work directly in the operating system shell or work within a shell inside GNU Emacs.

If you are using GNU Emacs, you can use commands provided by Texinfo mode instead of shell commands. In addition to the three commands to format a file, sort the indices, and print the result, Texinfo mode offers key bindings for commands to recenter the output buffer, show the print queue, and delete a job from the print queue.

The typesetting program called TeX is used for formatting a Texinfo file. TeX is a very powerful typesetting program and, if used right, does an exceptionally good job. See section How to Obtain TeX, for information on how to obtain TeX.

The `makeinfo`, `texinfo-format-region`, and `texinfo-format-buffer` commands read the very same @-commands in the Texinfo file as does TeX, but process them differently to make an Info file; see section Creating an Info File.

Format and Print Using Shell Commands

Format the Texinfo file with the shell command `tex` followed by the name of the Texinfo file. This produces a formatted DVI file as well as several auxiliary files containing indices, cross references, etc. The DVI file (for DeVice Independent file) can be printed on a wide variety of printers.

The `tex` formatting command itself does not sort the indices; it writes an output file of unsorted index data. This is a misfeature of TeX. Hence, to generate a printed index, you first need a sorted index to work from. The `texindex` command sorts indices. (The source file `` `texindex.c' `` comes as part of the standard GNU distribution and is usually installed when Emacs is installed.)

The `tex` formatting command outputs unsorted index files under names that obey a standard convention. These names are the name of your main input file to the `tex` formatting command, with everything after the first period thrown away, and the two letter names of indices added at the end. For example, the raw index output files for the input file `` `foo.texinfo' `` would be `` `foo.cp' ``, `` `foo.vr' ``, `` `foo.fn' ``, `` `foo.tp' ``, `` `foo.pg' `` and `` `foo.ky' ``. Those are exactly the arguments to give to `texindex`.

Or else, you can use `??' as "wild-cards" and give the command in this form:

```
texindex foo.??
```

This command will run `texindex` on all the unsorted index files, including any that you have defined yourself using `@defindex` or `@defcodeindex`. (You may execute `texindex foo.??' even if there are

similarly named files with two letter extensions that are not index files, such as `foo.el'. The `texinde` command reports but otherwise ignores such files.)

For each file specified, `texindex` generates a sorted index file whose name is made by appending `s' to the input file name. The `@printindex` command knows to look for a file of that name. `texindex` does not alter the raw index output file.

After you have sorted the indices, you need to rerun the `tex` formatting command on the Texinfo file. This regenerates a formatted DVI file with up-to-date index entries.(10)

To summarize, this is a three step process:

1. Run the `tex` formatting command on the Texinfo file. This generates the formatted DVI file as well as the raw index files with two letter extensions.
2. Run the shell command `texindex` on the raw index files to sort them. This creates the corresponding sorted index files.
3. Rerun the `tex` formatting command on the Texinfo file. This regenerates a formatted DVI file with the index entries in the correct order. This second run also corrects the page numbers for the cross references. (The tables of contents are always correct.)

You need not run `texindex` each time after you run the `tex` formatting. If you do not, on the next run, the `tex` formatting command will use whatever sorted index files happen to exist from the previous use of `texindex`. This is usually OK while you are debugging.

Rather than type the `tex` and `texindex` commands yourself, you can use `texi2dvi`. This shell script is designed to simplify the `tex---texindex---tex` sequence by figuring out whether index files and DVI files are up-to-date. It runs `texindex` and `tex` only when necessary.

The syntax for `texi2dvi` is like this (where `%' is the shell prompt):

```
% texi2dvi filename...
```

Finally, you can print the DVI file with the DVI print command. The precise command to use depends on the system; `lpr -d' is common. The DVI print command may require a file name without any extension or with a `.dvi' extension.

The following commands, for example, sort the indices, format, and print the Bison Manual (where `%' is the shell prompt):

```
% tex bison.texinfo
% texindex bison.??
% tex bison.texinfo
% lpr -d bison.dvi
```

(Remember that the shell commands may be different at your site; but these are commonly used versions.)

From an Emacs Shell ...

You can give formatting and printing commands from a shell within GNU Emacs. To create a shell within Emacs, type M-x shell. In this shell, you can format and print the document. See section Format and Print Using Shell Commands, for details.

You can switch to and from the shell buffer while `tex` is running and do other editing. If you are formatting a long document on a slow machine, this can be very convenient.

You can also use `texi2dvi` from an Emacs shell. For example, here is how to use `texi2dvi` to format and print Using and Porting GNU CC from a shell within Emacs (where `%' is the shell prompt):

```
% texi2dvi gcc.texinfo
% lpr -d gcc.dvi
```

Formatting and Printing in Texinfo Mode

Texinfo mode provides several predefined key commands for TeX formatting and printing. These include commands for sorting indices, looking at the printer queue, killing the formatting job, and recentering the display of the buffer in which the operations occur.

C-c C-t C-r

M-x texinfo-tex-region

Run TeX on the current region.

C-c C-t C-b

M-x texinfo-tex-buffer

Run TeX on the current buffer.

C-c C-t C-i

M-x texinfo-texindex

Sort the indices of a Texinfo file that have been formatted with `texinfo-tex-region` or `texinfo-tex-buffer`.

C-c C-t C-p

M-x texinfo-tex-print

Print a DVI file that was made with `texinfo-tex-region` or `texinfo-tex-buffer`.

C-c C-t C-q

M-x texinfo-show-tex-print-queue

Show the print queue.

C-c C-t C-d

M-x texinfo-delete-from-tex-print-queue

Delete a job from the print queue; you will be prompted for the job number shown by a preceding C-c C-t C-q command (`texinfo-show-tex-print-queue`).

C-c C-t C-k

M-x texinfo-kill-tex-job

Kill either the currently running TeX job that has been started by `texinfo-tex-region` or `texinfo-tex-buffer`, or any other process running in the Texinfo shell buffer.

C-c C-t C-x

M-x texinfo-quit-tex-job

Quit a TeX formatting job that has stopped because of an error by sending an x to it. When you do this, TeX preserves a record of what it did in a `` `.log' `` file.

C-c C-t C-l

M-x texinfo-recenter-tex-output-buffer

Redisplay the shell buffer in which the TeX printing and formatting commands are run to show its most recent output.

Thus, the usual sequence of commands for formatting a buffer is as follows (with comments to the right):

```
C-c C-t C-b             Run TeX on the buffer.
C-c C-t C-i             Sort the indices.
C-c C-t C-b             Rerun TeX to regenerate indices.
C-c C-t C-p             Print the DVI file.
C-c C-t C-q             Display the printer queue.
```

The Texinfo mode TeX formatting commands start a subshell in Emacs called the `` `*texinfo-tex-shell*' ``. The `texinfo-tex-command`, `texinfo-texindex-command`, and `tex-dvi-print-command` commands are all run in this shell.

You can watch the commands operate in the `*texinfo-tex-shell*' buffer, and you can switch to and from and use the `*texinfo-tex-shell*' buffer as you would any other shell buffer.

The formatting and print commands depend on the values of several variables. The default values are:

Variable	Default value
texinfo-tex-command	"tex"
texinfo-texindex-command	"texindex"
texinfo-tex-shell-cd-command	"cd"
texinfo-tex-dvi-print-command	"lpr -d"
texinfo-show-tex-queue-command	"lpq"
texinfo-delete-from-print-queue-command	"lprm"
texinfo-start-of-header	"%**start"
texinfo-end-of-header	"%**end"
texinfo-tex-trailer	"@bye"

The default values of both the `texinfo-tex-command` and the `texinfo-texindex-command` variables are set in the `` `texnfo-tex.el' `` file.

You can change the values of these variables with the M-x edit-options command (see section 'Editing Variable Values' in The GNU Emacs Manual), with the M-x set-variable command (see section 'Examining and Setting Variables' in The GNU Emacs Manual), or with your `.emacs' initialization file (see section 'Init File' in The GNU Emacs Manual).

Using the Local Variables List

Yet another way to apply the TeX formatting command to a Texinfo file is to put that command in a local variables list at the end of the Texinfo file. You can then specify the TeX formatting command as a `compile-command` and have Emacs run the TeX formatting command by typing M-x compile. This creates a special shell called the `*compilation buffer*' in which Emacs runs the compile command. For example, at the end of the `gdb.texinfo' file, after the `@bye`, you would put the following:

```
@c Local Variables:
@c compile-command: "tex gdb.texinfo"
@c End:
```

This technique is most often used by programmers who also compile programs this way; see section 'Compilation' in The GNU Emacs Manual.

TeX Formatting Requirements Summary

Every Texinfo file that is to be input to TeX must begin with a `\input` command and contain an `@settitle` command:

```
\input texinfo
@settitle name-of-manual
```

The first command instructs TeX to load the macros it needs to process a Texinfo file and the second command specifies the title of printed manual.

Every Texinfo file must end with a line that terminates TeX processing and forces out unfinished pages:

```
@bye
```

Strictly speaking, these three lines are all a Texinfo file needs for TeX, besides the body. (The `@setfilename` line is the only line that a Texinfo file needs for Info formatting.)

Usually, the file's first line contains an `@c -*-texinfo-*-' comment that causes Emacs to switch to Texinfo mode when you edit the file. In addition, the beginning usually includes an `@setfilename` for Info formatting, an `@setchapternewpage` command, a title page, a copyright page, and permissions. Besides an `@bye`, the end of a file usually includes indices and a table of contents.

For more information, see section `@setchapternewpage`, section Page Headings, section The Title and Copyright Pages, section Index Menus and Printing an Index, and section Generating a Table of

Contents.

Preparing to Use TeX

TeX needs to know where to find the `texinfo.tex' file that you have told it to input with the \input texinfo' command at the beginning of the first line. The `texinfo.tex' file tells TeX how to handle @-commands. (`texinfo.tex' is included in the standard GNU distributions.)

Usually, the `texinfo.tex' file is put in the default directory that contains TeX macros (the `/usr/lib/tex/macros' directory) when GNU Emacs or other GNU software is installed. In this case, TeX will find the file and you do not need to do anything special. Alternatively, you can put `texinfo.tex' in the directory in which the Texinfo source file is located, and TeX will find it there.

However, you may want to specify the location of the `\input' file yourself. One way to do this is to write the complete path for the file after the `\input' command. Another way is to set the `TEXINPUTS' environment variable in your `.cshrc' or `.profile' file. The `TEXINPUTS' environment variable will tell TeX where to find the `texinfo.tex' file and any other file that you might want TeX to use.

Whether you use a `.cshrc' or `.profile' file depends on whether you use `csh', `sh', or `bash' for your shell command interpreter. When you use `csh', it looks to the `.cshrc' file for initialization information, and when you use `sh' or `bash', it looks to the `.profile' file.

In a `.cshrc' file, you could use the following `csh' command sequence:

```
setenv TEXINPUTS .:/usr/me/mylib:/usr/lib/tex/macros
```

In a `.profile' file, you could use the following `sh' command sequence:

```
TEXINPUTS=.:/usr/me/mylib:/usr/lib/tex/macros
export TEXINPUTS
```

This would cause TeX to look for `\input' file first in the current directory, indicated by the `.', then in a hypothetical user's `me/mylib' directory, and finally in the system library.

Overfull "hboxes"

TeX is sometimes unable to typeset a line without extending it into the right margin. This can occur when TeX comes upon what it interprets as a long word that it cannot hyphenate, such as an electronic mail network address or a very long title. When this happens, TeX prints an error message like this:

```
Overfull \hbox (20.76302pt too wide)
```

(In TeX, lines are in "horizontal boxes", hence the term, "hbox". The backslash, `\', is the TeX equivalent of `@'.)

TeX also provides the line number in the Texinfo source file and the text of the offending line, which is

marked at all the places that TeX knows how to hyphenate words. See section Catching Errors with T Formatting, for more information about typesetting errors.

If the Texinfo file has an overfull hbox, you can rewrite the sentence so the overfull hbox does not occur, or you can decide to leave it. A small excursion into the right margin often does not matter and may not even be noticeable.

However, unless told otherwise, TeX will print a large, ugly, black rectangle beside the line that contains the overful hbox. This is so you will notice the location of the problem if you are correcting a draft.

To prevent such a monstrosity from marring your final printout, write the following in the beginning of the Texinfo file on a line of its own, before the `@titlepage` command:

```
@finalout
```

Printing "Small" Books

By default, TeX typesets pages for printing in an 8.5 by 11 inch format. However, you can direct TeX to typeset a document in a 7 by 9.25 inch format that is suitable for bound books by inserting the following command on a line by itself at the beginning of the Texinfo file, before the title page:

```
@smallbook
```

(Since regular sized books are often about 7 by 9.25 inches, this command might better have been called the `@regularbooksize` command, but it came to be called the `@smallbook` command by comparison to the 8.5 by 11 inch format.)

If you write the `@smallbook` command between the start-of-header and end-of-header lines, the Texinfo mode TeX region formatting command, `texinfo-tex-region`, will format the region in "small" book size (see section Start of Header).

The Free Software Foundation distributes printed copies of The GNU Emacs Manual and other manuals in the "small" book size. See section `@smallexample` and `@smalllisp`}, for information about commands that make it easier to produce examples for a smaller manual.

Printing on A4 Paper

You can tell TeX to typeset a document for printing on European size A4 paper with the `@afourpaper` command. Write the command on a line by itself between `@iftex` and `@end iftex` lines near the beginning of the Texinfo file, before the title page:

For example, this is how you would write the header for this manual:

```
\input texinfo    @c -*-texinfo-*-
@c %**start of header
```


```
@setfilename texinfo
@settitle Texinfo
@syncodeindex vr fn
@iftex
@afourpaper
@end iftex
@c %**end of header
```

Cropmarks and Magnification

You can attempt to direct TeX to print cropmarks at the corners of pages with the `@cropmarks` command. Write the `@cropmarks` command on a line by itself between `@iftex` and `@end iftex` lines near the beginning of the Texinfo file, before the title page, like this:

```
@iftex
@cropmarks
@end iftex
```

This command is mainly for printers that typeset several pages on one sheet of film; but you can attempt to use it to mark the corners of a book set to 7 by 9.25 inches with the `@smallbook` command. (Printers will not produce cropmarks for regular sized output that is printed on regular sized paper.) Since different printing machines work in different ways, you should explore the use of this command with a spirit of adventure. You may have to redefine the command in the `` `texinfo.tex' `` definitions file.

You can attempt to direct TeX to typeset pages larger or smaller than usual with the `\mag` TeX command. Everything that is typeset is scaled proportionally larger or smaller. (`\mag` stands for "magnification".) This is *not* a Texinfo @-command, but is a PlainTeX command that is prefixed with a backslash. You have to write this command between `@tex` and `@end tex` (see section Using Ordinary TeX Commands).

Follow the `\mag` command with an `=' and then a number that is 1000 times the magnification you desire. For example, to print pages at 1.2 normal size, write the following near the beginning of the Texinfo file, before the title page:

```
@tex
\mag=1200
@end tex
```

With some printing technologies, you can print normal-sized copies that look better than usual by using a larger-than-normal master.

Depending on your system, `\mag` may not work or may work only at certain magnifications. Be prepared to experiment.

Go to the previous, next section.

Go to the previous, next section.

Creating an Info File

`makeinfo` is a utility that converts a Texinfo file into an Info file; `texinfo-format-region` and `texinfo-format-buffer` are GNU Emacs functions that do the same.

A Texinfo file must possess an `@setfilename` line near its beginning, otherwise the Info formatting commands will fail.

For information on installing the Info file in the Info system, see section Installing an Info File.

The `makeinfo` utility creates an Info file from a Texinfo source file more quickly than either of the Emacs formatting commands and provides better error messages. We recommend it. `makeinfo` is a C program that is independent of Emacs. You do not need to run Emacs to use `makeinfo`, which means you can use `makeinfo` on machines that are too small to run Emacs. You can run `makeinfo` in any one of three ways: from an operating system shell, from a shell inside Emacs, or by typing a key command in Texinfo mode in Emacs.

The `texinfo-format-region` and the `texinfo-format-buffer` commands are useful if you cannot run `makeinfo`. Also, in some circumstances, they format short regions or buffers more quickly than `makeinfo`.

Invoking `makeinfo` from a Shell

To create an Info file from a Texinfo file, type `makeinfo` followed by the name of the Texinfo file. Thus, to create the Info file for Bison, type the following at the shell prompt (where `%' is the prompt):

```
% makeinfo bison.texinfo
```

(You can run a shell inside Emacs by typing M-x shell.)

Options for `makeinfo`

The `makeinfo` command takes a number of options. Most often, options are used to set the value of the fill column and specify the footnote style. Each command line option is a word preceded by `--'(11) or a letter preceded by `-'. You can use abbreviations for the option names as long as they are unique.

For example, you could use the following command to create an Info file for `bison.texinfo' in which each line is filled to only 68 columns (where `%' is the prompt):

```
% makeinfo --fill-column=68 bison.texinfo
```

You can write two or more options in sequence, like this:


```
% makeinfo --no-split --fill-column=70 ...
```

This would keep the Info file together as one possibly very long file and would also set the fill column to 70.

If you wish to discover which version of `makeinfo` you are using, type:

```
% makeinfo --version
```

The options are:

`-D var`

Cause var to be defined. This is equivalent to `@set var` in the Texinfo file.

`--error-limit limit`

Set the maximum number of errors that `makeinfo` will report before exiting (on the assumption that continuing would be useless). The default number of errors that can be reported before `makeinfo` gives up is 100.

`--fill-column width`

Specify the maximum number of columns in a line; this is the right-hand edge of a line. Paragraphs that are filled will be filled to this width. (Filling is the process of breaking up and connecting lines so that lines are the same length as or shorter than the number specified as the fill column. Lines are broken between words.) The default value for `fill-column` is 72.

`--footnote-style style`

Set the footnote style to style, either `end' for the end node style or `separate' for the separate node style. The value set by this option overrides the value set in a Texinfo file by an `@footnotestyle` command. When the footnote style is `separate', `makeinfo` makes a new node containing the footnotes found in the current node. When the footnote style is `end', `makeinfo` places the footnote references at the end of the current node.

`-I dir`

Add `dir` to the directory search list for finding files that are included using the `@include` command. By default, `makeinfo` searches only the current directory.

`--no-headers`

Do not include menus or node lines in the output. This results in an ASCII file that you cannot read in Info since it does not contain the requisite nodes or menus; but you can print such a file in a single, typewriter-like font and produce acceptable output.

`--no-split`

Suppress the splitting stage of `makeinfo`. Normally, large output files (where the size is greater than 70k bytes) are split into smaller subfiles, each one approximately 50k bytes. If you specify `--no-split', `makeinfo` will not split up the output file.

`--no-pointer-validate`

`--no-validate`

Suppress the pointer-validation phase of `makeinfo`. Normally, after a Texinfo file is processed,

some consistency checks are made to ensure that cross references can be resolved, etc. See sec Pointer Validation.

`--no-warn`

Suppress the output of warning messages. This does *not* suppress the output of error messages, only warnings. You might want this if the file you are creating has examples of Texinfo cross references within it, and the nodes that are referenced do not actually exist.

`--no-number-footnotes`

Supress automatic footnote numbering. By default, `makeinfo` numbers each footnote sequentially in a single node, resetting the current footnote number to 1 at the start of each node.

`--output file`

`-o file`

Specify that the output should be directed to file and not to the file name specified in the `@setfilename` command found in the Texinfo source. file can be the special token `-', which specifies standard output.

`--paragraph-indent indent`

Set the paragraph indentation style to indent. The value set by this option overrides the value set in a Texinfo file by an `@paragraphindent` command. The value of indent is interpreted as follows:

If the value of indent is `asis', do not change the existing indentation at the starts of paragraphs.

If the value of indent is zero, delete any existing indentation.

If the value of indent is greater than zero, indent each paragraph by that number of spaces.

`--reference-limit limit`

Set the value of the number of references to a node that `makeinfo` will make without reporting a warning. If a node has more than this number of references in it, `makeinfo` will make the references but also report a warning.

`-U var`

Cause var to be undefined. This is equivalent to `@clear var` in the Texinfo file.

`--verbose`

Cause `makeinfo` to display messages saying what it is doing. Normally, `makeinfo` only outputs messages if there are errors or warnings.

`--version`

Report the version number of this copy of `makeinfo`.

Pointer Validation

`makeinfo` will check the validity of the final Info file unless you suppress pointer-validation by using the `--no-pointer-validation' option. Mostly, this means ensuring that nodes you have referenced really exist. Here is a complete list of what is checked:

1. If a `Next', `Previous', or `Up' node reference is a reference to a node in the current file and is not an external reference such as to `(dir)', then the referenced node must exist.
2. In every node, if the `Previous' node is different from the `Up' node, then the `Previous' node must also be pointed to by a `Next' node.
3. Every node except the `Top' node must have an `Up' pointer.
4. The node referenced by an `Up' pointer must contain a reference to the current node in some manner other than through a `Next' reference. This includes menu entries and cross references.
5. If the `Next' reference of a node is not the same as the `Next' reference of the `Up' reference, then the node referenced by the `Next' pointer must have a `Previous' pointer that points back to the current node. This rule allows the last node in a section to point to the first node of the next chapter.

Running `makeinfo` inside Emacs

You can run `makeinfo` in GNU Emacs Texinfo mode by using either the `makeinfo-region` or the `makeinfo-buffer` commands. In Texinfo mode, the commands are bound to C-c C-m C-r and C-c C-m C-b by default.

C-c C-m C-r

M-x makeinfo-region

Format the current region for Info.

C-c C-m C-b

M-x makeinfo-buffer

Format the current buffer for Info.

When you invoke either `makeinfo-region` or `makeinfo-buffer`, Emacs prompts for a file name, offering the name of the visited file as the default. You can edit the default file name in the minibuffer if you wish, before typing RET to start the `makeinfo` process.

The Emacs `makeinfo-region` and `makeinfo-buffer` commands run the `makeinfo` program in a temporary shell buffer. If `makeinfo` finds any errors, Emacs displays the error messages in the temporary buffer.

You can parse the error messages by typing C-x ` (`next-error`). This causes Emacs to go to and position the cursor on the line in the Texinfo source that `makeinfo` thinks caused the error. See section 'Running `make` or Compilers Generally' in The GNU Emacs Manual, for more information about using the `next-error` command.

In addition, you can kill the shell in which the `makeinfo` command is running or make the shell buffer display its most recent output.

C-c C-m C-k

M-x makeinfo-kill-job

Kill the currently running job created by `makeinfo-region` or `makeinfo-buffer`.

C-c C-m C-l

M-x makeinfo-recenter-output-buffer

Redisplay the `makeinfo` shell buffer to display its most recent output.

(Note that the parallel commands for killing and recentering a TeX job are C-c C-t C-k and C-c C-t C-l. See section Formatting and Printing in Texinfo Mode.)

You can specify options for `makeinfo` by setting the `makeinfo-options` variable with either the M-x edit-options or the M-x set-variable command, or by setting the variable in your `` `.emacs' `` initialization file.

For example, you could write the following in your `` `.emacs' `` file:

```
(setq makeinfo-options
      "--paragraph-indent=0 --no-split
       --fill-column=70 --verbose")
```

For more information, see section Options for `makeinfo`}, as well as "Editing Variable Values,""Examining and Setting Variables," and "Init File" in the The GNU Emacs Manual.

The `texinfo-format...` Commands

In GNU Emacs in Texinfo mode, you can format part or all of a Texinfo file with the `texinfo-format-region` command. This formats the current region and displays the formatted text in a temporary buffer called `*Info Region*'.

Similarly, you can format a buffer with the `texinfo-format-buffer` command. This command creates a new buffer and generates the Info file in it. Typing C-x C-s will save the Info file under the name specified by the `@setfilename` line which must be near the beginning of the Texinfo file.

C-c C-e C-r

`texinfo-format-region`

Format the current region for Info.

C-c C-e C-b

`texinfo-format-buffer`

Format the current buffer for Info.

The `texinfo-format-region` and `texinfo-format-buffer` commands provide you with some error checking, and other functions can provide you with further help in finding formatting errors. These procedures are described in an appendix; see section Formatting Mistakes. However, the `makeinfo` program is often faster and provides better error checking (see section Running `makeinfo` inside Emacs).

Batch Formatting

You can format Texinfo files for Info using `batch-texinfo-format` and Emacs Batch mode. You can run Emacs in Batch mode from any shell, including a shell inside of Emacs. (See section 'Command Line Switches and Arguments' in The GNU Emacs Manual.)

Here is the command to format all the files that end in `.texinfo' in the current directory (where `%' is the shell prompt):

```
% emacs -batch -funcall batch-texinfo-format *.texinfo
```

Emacs processes all the files listed on the command line, even if an error occurs while attempting to format some of them.

Run `batch-texinfo-format` only with Emacs in Batch mode as shown; it is not interactive. It kills the Batch mode Emacs on completion.

`batch-texinfo-format` is convenient if you lack `makeinfo` and want to format several Texinfo files at once. When you use Batch mode, you create a new Emacs process. This frees your current Emacs, so you can continue working in it. (When you run `texinfo-format-region` or `texinfo-format-buffer`, you cannot use that Emacs for anything else until the command finishes.)

Tag Files and Split Files

If a Texinfo file has more than 30,000 bytes, `texinfo-format-buffer` automatically creates a tag table for its Info file; `makeinfo` always creates a tag table. With a tag table, Info can jump to new nodes more quickly than it can otherwise.

In addition, if the Texinfo file contains more than about 70,000 bytes, `texinfo-format-buffer` and `makeinfo` split the large Info file into shorter indirect subfiles of about 50,000 bytes each. Big files are split into smaller files so that Emacs does not need to make a large buffer to hold the whole of a large Info file; instead, Emacs allocates just enough memory for the small, split off file that is needed at the time. This way, Emacs avoids wasting memory when you run Info. (Before splitting was implemented, Info files were always kept short and include files were designed as a way to create a single, large printed manual out of the smaller Info files. See section Include Files, for more information. Include files are still used for very large documents, such as The Emacs Lisp Reference Manual, in which each chapter is a separate file.)

When a file is split, Info itself makes use of a shortened version of the original file that contains just the tag table and references to the files that were split off. The split off files are called indirect files.

The split off files have names that are created by appending `-1', `-2', `-3' and so on to the file name specified by the `@setfilename` command. The shortened version of the original file continues to have the name specified by `@setfilename`.

At one stage in writing this document, for example, the Info file was saved as `test-texinfo' and

that file looked like this:

```
Info file: test-texinfo,    -*-Text-*-
produced by texinfo-format-buffer
from file: new-texinfo-manual.texinfo

^_
Indirect:
test-texinfo-1: 102
test-texinfo-2: 50422
test-texinfo-3: 101300
^_^L
Tag table:
(Indirect)
Node: overview^?104
Node: info file^?1271
Node: printed manual^?4853
Node: conventions^?6855
...
```

(But `test-texinfo' had far more nodes than are shown here.) Each of the split off, indirect files, `test-texinfo-1', `test-texinfo-2', and `test-texinfo-3', is listed in this file after the line that says `Indirect:'. The tag table is listed after the line that says `Tag table:'.

In the list of indirect files, the number following the file name records the cumulative number of bytes in the preceding indirect files, not counting the file list itself, the tag table, or the permissions text in each file. In the tag table, the number following the node name records the location of the beginning of the node, in bytes from the beginning.

If you are using `texinfo-format-buffer` to create Info files, you may want to run the `Info-validate` command. (The `makeinfo` command does such a good job on its own, you do not need `Info-validate`.) However, you cannot run the M-x Info-validate node-checking command on indirect files. For information on how to prevent files from being split and how to validate the structure of the nodes, see section Running `Info-validate`.

Go to the previous, next section.

Go to the previous, next section.

Installing an Info File

Info files are usually kept in the `info' directory. (You can find the location of this directory within Emacs by typing C-h i to enter Info and then typing C-x C-f to see the full pathname to the `info' directory.)

For Info to work, the `info' directory must contain a file that serves as a top level directory for the Info system. By convention, this file is called `dir'. The `dir' file is itself an Info file. It contains the top level menu for all the Info files in the system. The menu looks like this:

```
* Menu:

* Info:    (info).     Documentation browsing system.
* Emacs:   (emacs).    The extensible, self-documenting
                       text editor.
* Texinfo: (texinfo).  With one source file, make
                       either a printed manual using
                       TeX or an Info file.
...
```

Each of these menu entries points to the `Top' node of the Info file that is named in parentheses. (The menu entry does not need to specify the `Top' node, since Info goes to the `Top' node if no node name is mentioned. See section Referring to Other Info Files.)

Thus, the `Info' entry points to the `Top' node of the `info' file and the `Emacs' entry points to the `Top' node of the `emacs' file.

In each of the Info files, the `Up' pointer of the `Top' node refers back to the `dir' file. For example, the line for the `Top' node of the Emacs manual looks like this in Info:

```
File: emacs  Node: Top, Up: (DIR), Next: Distrib
```

(Note that in this case, the `dir' file name is written in upper case letters--it can be written in either upper or lower case. Info has a feature that it will change the case of the file name to lower case if it cannot find the name as written.)

Listing a New Info File

To add a new Info file to your system, write a menu entry for it in the menu in the `dir' file in the `info' directory. Also, move the new Info file itself to the `info' directory. For example, if you were adding documentation for GDB, you would write the following new entry:


```
* GDB: (gdb).            The source-level C debugger.
```

The first part of the menu entry is the menu entry name, followed by a colon. The second part is the name of the Info file, in parentheses, followed by a period. The third part is the description.

Conventionally, the name of an Info file has a `.info' extension. Thus, you might list the name of the file like this:

```
* GDB: (gdb.info).            The source-level C debugger.
```

However, Info will look for a file with a `.info' extension if it does not find the file under the name given in the menu. This means that you can refer to the file `gdb.info' as `gdb', as shown in the first example. This looks better.

Info Files in Other Directories

If an Info file is not in the `info' directory, there are two ways to specify its location:

- Write the pathname as the menu's second part, or;
- Specify the `info' directory name in an environment variable in your `.profile' or `.cshrc' initialization file. (Only you and others with the same environment variable will be able to find Info files whose location is specified this way.)

For example, to reach a test file in the `~bob/manuals' directory, you could add an entry like this to the menu in the `dir' file:

```
* Test: (~bob/manuals/info-test).  Bob's own test file.
```

In this case, the absolute file name of the `info-test' file is written as the second part of the menu entry.

Alternatively, you can tell Info where to look by setting the `INFOPATH' environment variable in your `.cshrc' or `.profile' file.

If you use `sh' or `bash' for your shell command interpreter, you must set the `INFOPATH' environment variable in the `.profile' initialization file; but if you use `csh', you must set the variable in the `.cshrc' initialization file. The two files require slightly different command formats.

- In a `.cshrc' file, you could set the `INFOPATH' variable as follows:

  ```
  setenv INFOPATH .:~bob/manuals:/usr/local/emacs/info
  ```

- In a `.profile' file, you would achieve the same effect by writing:

  ```
  INFOPATH=.:~bob/manuals:/usr/local/emacs/info
  export INFOPATH
  ```

Either form would cause Info to look first in the current directory, indicated by the `.', then in the `~bob/manuals' directory, and finally in the `/usr/local/emacs/info' directory (which is a

common location for the standard Info directory).

Go to the previous, next section.

Go to the previous, next section.

@-Command List

Here is an alphabetical list of the @-commands in Texinfo. Square brackets, `[ ]`, indicate optional arguments; an ellipsis, `...', indicates repeated text.

`@*`

Force a line break. Do not end a paragraph that uses `@*` with an `@refill` command. See section @*: Generate Line Breaks.

`@.`

Stands for a period that really does end a sentence (usually after an end-of-sentence capital letter). See section Spacing After Colons and Periods.

`@:`

Indicate to TeX that an immediately preceding period, question mark, exclamation mark, or colon does not end a sentence. Prevent TeX from inserting extra whitespace as it does at the end of a sentence. The command has no effect on the Info file output. See section Spacing After Colons and Periods.

`@@`

Stands for `@'. See section Inserting `@', Braces, and Periods}.

`@{`

Stands for a left-hand brace, `{'. See section Inserting `@', Braces, and Periods.

`@}`

Stands for a right-hand brace, `}'. See section Inserting `@', Braces, and Periods.

`@appendix title`

Begin an appendix. The title appears in the table of contents of a printed manual. In Info, the title is underlined with asterisks. See section `@unnumbered, @appendix` and `@appendix` Commands}.

`@appendixsec title`

`@appendixsection title`

Begin an appendix section within an appendix. The section title appears in the table of contents of a printed manual. In Info, the title is underlined with equal signs. `@appendixsection` is a longer spelling of the `@appendixsec` command. See section `@unnumberedsec, @appendixsec, @heading`.

`@appendixsubsec title`

Begin an appendix subsection within an appendix. The title appears in the table of contents of a printed manual. In Info, the title is underlined with hyphens. See section The `@subsection`-like Commands.

`@appendixsubsubsec title`

Begin an appendix subsubsection within a subappendix. The title appears in the table of contents of a printed manual. In Info, the title is underlined with periods. See section The `subsub' Commands.

`@asis`

Used following `@table`, `@ftable`, and `@vtable` to print the table's first column without highlighting ("as is"). See section Making a Two-column Table.

`@author author`

Typeset author flushleft and underline it. See section `@title`, `@subtitle`, and `@author` and `@author` Commands}.

`@b{text}`

Print text in **bold** font. No effect in Info. See section Fonts for Printing, Not Info.

`@bullet{}`

Generate a large round dot, or the closest possible thing to one. See section `@bullet{}`.

`@bye`

Stop formatting a file. The formatters do not see the contents of a file following an `@bye` command. See section Ending a Texinfo File.

`@c comment`

Begin a comment in Texinfo. The rest of the line does not appear in either the Info file or the printed manual. A synonym for `@comment`. See section General Syntactic Conventions.

`@cartouche`

Highlight an example or quotation by drawing a box with rounded corners around it. Pair with `@end cartouche`. No effect in Info. See section Drawing Cartouches Around Examples.)

`@center line-of-text`

Center the line of text following the command. See section `@titlefont`, `@center`, and `@sp`.

`@chapheading title`

Print a chapter-like heading in the text, but not in the table of contents of a printed manual. In Info, the title is underlined with asterisks. See section `@majorheading`, `@chapheading`}.

`@chapter title`

Begin a chapter. The chapter title appears in the table of contents of a printed manual. In Info, the title is underlined with asterisks. See section `@chapter`.

`@cindex entry`

Add entry to the index of concepts. See section Making Index Entries.

`@cite{reference}`

Highlight the name of a book or other reference that lacks a companion Info file. See section `@cite`{reference}.

`@clear flag`

Unset flag, preventing the Texinfo formatting commands from formatting text between subsequent pairs of `@ifset flag` and `@end ifset` commands, and preventing `@value{flag}` from expanding to the value to which flag is set. See section `@set, @clear, and @value`.

`@code{sample-code}`

Highlight text that is an expression, a syntactically complete token of a program, or a program name. See section `@code{sample-code}`.

`@comment comment`

Begin a comment in Texinfo. The rest of the line does not appear in either the Info file or the printed manual. A synonym for `@c`. See section General Syntactic Conventions.

`@contents`

Print a complete table of contents. Has no effect in Info, which uses menus instead. See section Generating a Table of Contents.

`@copyright{}`

Generate a copyright symbol. See section `@copyright{}`.

`@defcodeindex index-name`

Define a new index and its indexing command. Print entries in an `@code` font. See section Defining New Indices.

`@defcv category class name`

Format a description for a variable associated with a class in object-oriented programming. Takes three arguments: the category of thing being defined, the class to which it belongs, and its name. See section Definition Commands.

`@deffn category name arguments...`

Format a description for a function, interactive command, or similar entity that may take arguments. `@deffn` takes as arguments the category of entity being described, the name of this particular entity, and its arguments, if any. See section Definition Commands.

`@defindex index-name`

Define a new index and its indexing command. Print entries in a roman font. See section Defining New Indices.

`@defivar class instance-variable-name`

Format a description for an instance variable in object-oriented programming. The command is equivalent to `@defcv {Instance Variable} ...'. See section Definition Commands.

`@defmac macro-name arguments...`

Format a description for a macro. The command is equivalent to `@deffn Macro ...'. See section Definition Commands.

`@defmethod class method-name arguments...`

Format a description for a method in object-oriented programming. The command is equivalent to `@defop Method ...'. Takes as arguments the name of the class of the method, the name of the method, and its arguments, if any. See section Definition Commands.

`@defop category class name arguments...`

Format a description for an operation in object-oriented programming. `@defop` takes as arguments the overall name of the category of operation, the name of the class of the operation, the name of the operation, and its arguments, if any. See section Definition Commands.

`@defopt option-name`

Format a description for a user option. The command is equivalent to `@defvr {User Option} ...'. See section Definition Commands.

`@defspec special-form-name arguments...`

Format a description for a special form. The command is equivalent to `@deffn {Special Form} ...'. See section Definition Commands.

`@deftp category name-of-type attributes...`

Format a description for a data type. `@deftp` takes as arguments the category, the name of the type (which is a word like `int' or `float'), and then the names of attributes of objects of that type. See section Definition Commands.

`@deftypefn classification data-type name arguments...`

Format a description for a function or similar entity that may take arguments and that is typed. `@deftypefn` takes as arguments the classification of entity being described, the type, the name of the entity, and its arguments, if any. See section Definition Commands.

`@deftypefun data-type function-name arguments...`

Format a description for a function in a typed language. The command is equivalent to `@deftypefn Function ...'. See section Definition Commands.

`@deftypevr classification data-type name`

Format a description for something like a variable in a typed language--an entity that records a value. Takes as arguments the classification of entity being described, the type, and the name of the entity. See section Definition Commands.

`@deftypevar data-type variable-name`

Format a description for a variable in a typed language. The command is equivalent to `@deftypevr Variable ...'. See section Definition Commands.

`@defun function-name arguments...`

Format a description for functions. The command is equivalent to `@deffn Function ...'. See section Definition Commands.

`@defvar variable-name`

Format a description for variables. The command is equivalent to `@defvr Variable ...'. See section Definition Commands.

`@defvr category name`

Format a description for any kind of variable. `@defvr` takes as arguments the category of the entity and the name of the entity. See section Definition Commands.

`@dfn{term}`

Highlight the introductory or defining use of a term. See section @dfn{term}.

`@display`

Begin a kind of example. Indent text, do not fill, do not select a new font. Pair with `@end display`. See section @display}.

`@dmn{dimension}`

Format a dimension. Cause TeX to insert a narrow space before dimension. No effect in Info. Use for writing a number followed by an abbreviation of a dimension name, such as `12pt', written as `12@dmn{pt}', with no space between the number and the `@dmn` command. See section @dmn{dimension}: Format a Dimension}.

`@dots{}`

Insert an ellipsis: `...'. See section @dots{}.

`@emph{text}`

Highlight text; text is displayed in *italics* in printed output, and surrounded by asterisks in Info. See section Emphasizing Text.

`@enumerate [number-or-letter]`

Begin a numbered list, using `@item` for each entry. Optionally, start list with number-or-letter. Pair with `@end enumerate`. See section Making a Numbered or Lettered List}.

`@equiv{}`

Indicate to the reader the exact equivalence of two forms with a glyph: `==='. See section ==: Indicating Equivalence.

`@error{}`

Indicate to the reader with a glyph that the following text is an error message: `error-->'. See section error-->: Indicating an Error Message.

`@evenfooting [left] @| [center] @| [right]`

Specify page footings for even-numbered (left-hand) pages. Not relevant to Info. See section How to Make Your Own Headings.

`@evenheading [left] @| [center] @| [right]`

Specify page headings for even-numbered (left-hand) pages. Not relevant to Info. See section How to Make Your Own Headings.

`@everyfooting [left] @| [center] @| [right]`

Specify page footings for every page. Not relevant to Info. See section How to Make Your Own Headings.

`@everyheading [left] @| [center] @| [right]`

Specify page headings for every page. Not relevant to Info. See section How to Make Your Own Headings.

`@example`

Begin an example. Indent text, do not fill, and select fixed-width font. Pair with `@end example`.

See section @example}.

`@exdent line-of-text`

Remove any indentation a line might have. See section @exdent: Undoing a Line's Indentation.

`@expansion{}`

Indicate the result of a macro expansion to the reader with a special glyph: `==>'. See section ==>: Indicating an Expansion.

`@file{filename}`

Highlight the name of a file, buffer, node, or directory. See section @file{file-name}}.

`@finalout`

Prevent TeX from printing large black warning rectangles beside over-wide lines. See section Overfull "hboxes".

`@findex entry`

Add entry to the index of functions. See section Making Index Entries.

`@flushleft`

Left justify every line but leave the right end ragged. Leave font as is. Pair with `@end flushleft`. See section @flushleft and @flushright}.

`@flushright`

Right justify every line but leave the left end ragged. Leave font as is. Pair with `@end flushright`. See section @flushleft and @flushright}.

`@footnote{text-of-footnote}`

Enter a footnote. Footnote text is printed at the bottom of the page by TeX; Info may format in either `End' node or `Separate' node style. See section Footnotes.

`@footnotestyle style`

Specify an Info file's footnote style, either `end' for the end node style or `separate' for the separate node style. See section Footnotes.

`@format`

Begin a kind of example. Like `@example` or `@display`, but do not narrow the margins and do not select the fixed-width font. Pair with `@end format`. See section @example}.

`@ftable formatting-command`

Begin a two-column table, using `@item` for each entry. Automatically enter each of the items in the first column into the index of functions. Pair with `@end ftable`. The same as `@table`, except for indexing. See section @ftable and @vtable and `@vtable`}.

`@group`

Hold text together that must appear on one printed page. Pair with `@end group`. Not relevant to Info. See section @group: Prevent Page Breaks}.

`@heading title`

Print an unnumbered section-like heading in the text, but not in the table of contents of a printed

manual. In Info, the title is underlined with equal signs. See section @unnumberedsec, @appendixsec, @heading.

`@headings on-off-single-double`

Turn page headings on or off, or specify single-sided or double-sided page headings for printing. `@headings on` is synonymous with `@headings double`. See section The @headings Command Command}.

`@i{text}`

Print text in *italic* font. No effect in Info. See section Fonts for Printing, Not Info.

`@ifclear flag`

If flag is cleared, the Texinfo formatting commands format text between `@ifclear flag` and the following `@end ifclear` command. See section @set, @clear, and @value.

`@ifinfo`

Begin a stretch of text that will be ignored by TeX when it typesets the printed manual. The text appears only in the Info file. Pair with `@end ifinfo`. See section Conditionally Visible Text.

`@ifset flag`

If flag is set, the Texinfo formatting commands format text between `@ifset flag` and the following `@end ifset` command. See section @set, @clear, and @value.

`@iftex`

Begin a stretch of text that will not appear in the Info file, but will be processed only by TeX. Pair with `@end iftex`. See section Conditionally Visible Text.

`@ignore`

Begin a stretch of text that will not appear in either the Info file or the printed output. Pair with `@end ignore`. See section Comments.

`@include filename`

Incorporate the contents of the file filename into the Info file or printed document. See section Include Files.

`@inforef{node-name, [entry-name], info-file-name}`

Make a cross reference to an Info file for which there is no printed manual. See section @inforef}.

`\input macro-definitions-file`

Use the specified macro definitions file. This command is used only in the first line of a Texinfo file to cause TeX to make use of the `` `texinfo' `` macro definitions file. The backslash in `\input` is used instead of an @ because TeX does not properly recognize @ until after it has read the definitions file. See section The Texinfo File Header.

`@item`

Indicate the beginning of a marked paragraph for `@itemize` and `@enumerate`; indicate the beginning of the text of a first column entry for `@table`, `@ftable`, and `@vtable`. See section Making Lists and Tables.

`@itemize mark-generating-character-or-command`

Produce a sequence of indented paragraphs, with a mark inside the left margin at the beginning of each paragraph. Pair with `@end itemize`. See section Making an Itemized List.

`@itemx`

Like `@item` but do not generate extra vertical space above the item text. See section `@itemx`.

`@kbd{keyboard-characters}`

Indicate text that consists of characters of input to be typed by users. See section `@kbd{keyboard-characters}`.

`@key{key-name}`

Highlight key-name, a conventional name for a key on a keyboard. See section `@key{key-name}`.

`@kindex entry`

Add entry to the index of keys. See section Making Index Entries.

`@lisp`

Begin an example of Lisp code. Indent text, do not fill, and select fixed-width font. Pair with `@end lisp`. See section `@lisp`}.

`@majorheading title`

Print a chapter-like heading in the text, but not in the table of contents of a printed manual. Generate more vertical whitespace before the heading than the `@chapheading` command. In Info, the chapter heading line is underlined with asterisks. See section `@majorheading`, `@chapheading` and `@chapheading`}.

`@menu`

Mark the beginning of a menu of nodes in Info. No effect in a printed manual. Pair with `@end menu`. See section Menus.

`@minus{}`

Generate a minus sign. See section `@minus{}`: Inserting a Minus Sign.

`@need n`

Start a new page in a printed manual if fewer than n mils (thousandths of an inch) remain on the current page. See section `@need mils`: Prevent Page Breaks}.

`@node name, next, previous, up`

Define the beginning of a new node in Info, and serve as a locator for references for TeX. See section The `@node` Command.

`@noindent`

Prevent text from being indented as if it were a new paragraph. See section `@noindent`.

`@oddfooting [left] @| [center] @| [right]`

Specify page footings for odd-numbered (right-hand) pages. Not relevant to Info. See section How to Make Your Own Headings.

`@oddheading [left] @| [center] @| [right]`

Specify page headings for odd-numbered (right-hand) pages. Not relevant to Info. See section How to Make Your Own Headings.

`@page`

Start a new page in a printed manual. No effect in Info. See section `@page`: Start a New Page.

`@paragraphindent indent`

Indent paragraphs by indent number of spaces; delete indentation if the value of indent is 0; and do not change indentation if indent is `asis`. See section Paragraph Indenting.

`@pindex entry`

Add entry to the index of programs. See section Making Index Entries.

`@point{}`

Indicate the position of point in a buffer to the reader with a glyph: `-!-'. See section Indicating Point in a Buffer.

`@print{}`

Indicate printed output to the reader with a glyph: `-|'. See section -|: Indicating Printed Output.

`@printindex index-name`

Print an alphabetized two-column index in a printed manual or generate an alphabetized menu of index entries for Info. See section Index Menus and Printing an Index.

`@pxref{node-name, [entry], [topic-or-title], [info-file], [manual]}`

Make a reference that starts with a lower case `see' in a printed manual. Use within parentheses only. Do not follow command with a punctuation mark. The Info formatting commands automatically insert terminating punctuation as needed, which is why you do not need to insert punctuation. Only the first argument is mandatory. See section `@pxref`.

`@quotation`

Narrow the margins to indicate text that is quoted from another real or imaginary work. Write command on a line of its own. Pair with `@end quotation`. See section `@quotation`}.

`@r{text}`

Print text in roman font. No effect in Info. See section Fonts for Printing, Not Info.

`@ref{node-name, [entry], [topic-or-title], [info-file], [manual]}`

Make a reference. In a printed manual, the reference does not start with a `See'. Follow command with a punctuation mark. Only the first argument is mandatory. See section `@ref`.

`@refill`

In Info, refill and indent the paragraph after all the other processing has been done. No effect on TeX, which always refills. This command is no longer needed, since all formatters now automatically refill. See section Refilling Paragraphs.

`@result{}`

Indicate the result of an expression to the reader with a special glyph: `=>'. See section =>: Indicating Evaluation.

`@samp{text}`

Highlight text that is a literal example of a sequence of characters. Used for single characters, for statements, and often for entire shell commands. See section `@samp{text}`.

`@sc{text}`

Set text in a printed output in THE SMALL CAPS FONT and set text in the Info file in uppercase letters. See section `@sc{text}`: The Small Caps Font.

`@section title`

Begin a section within a chapter. In a printed manual, the section title is numbered and appears in the table of contents. In Info, the title is underlined with equal signs. See section `@section`}.

`@set flag [string]`

Make flag active, causing the Texinfo formatting commands to format text between subsequent pairs of `@ifset flag` and `@end ifset` commands. Optionally, set value of flag to string. See section `@set`, `@clear`, and `@value`.

`@setchapternewpage on-off-odd`

Specify whether chapters start on new pages, and if so, whether on odd-numbered (right-hand) new pages. See section `@setchapternewpage`}.

`@setfilename info-file-name`

Provide a name for the Info file. See section General Syntactic Conventions.

`@settitle title`

Provide a title for page headers in a printed manual. See section General Syntactic Conventions.

`@shortcontents`

Print a short table of contents. Not relevant to Info, which uses menus rather than tables of contents. A synonym for `@summarycontents`. See section Generating a Table of Contents.

`@smallbook`

Cause TeX to produce a printed manual in a 7 by 9.25 inch format rather than the regular 8.5 by 11 inch format. See section Printing "Small" Books. Also, see section `@smallexample` and `@smalllisp` and `@smalllisp`}.

`@smallexample`

Indent text to indicate an example. Do not fill, select fixed-width font. In `@smallbook` format, print text in a smaller font than with `@example`. Pair with `@end smallexample`. See section `@smallexample` and `@smalllisp`}.

`@smalllisp`

Begin an example of Lisp code. Indent text, do not fill, select fixed-width font. In `@smallbook` format, print text in a smaller font. Pair with `@end smalllisp`. See section `@smallexample` and `@smalllisp` and `@smalllisp`}.

`@sp n`

Skip n blank lines. See section `@sp` n: Insert Blank Lines.

`@strong text`

Emphasize text by typesetting it in a **bold** font for the printed manual and by surrounding it wi asterisks for Info. See section @emph{text} and @strong{text}.

`@subheading title`

Print an unnumbered subsection-like heading in the text, but not in the table of contents of a printed manual. In Info, the title is underlined with hyphens. See section The @subsection-like Commands @appendixsubsec @subheading}.

`@subsection title`

Begin a subsection within a section. In a printed manual, the subsection title is numbered and appears in the table of contents. In Info, the title is underlined with hyphens. See section The @subsection Command}.

`@subsubheading title`

Print an unnumbered subsubsection-like heading in the text, but not in the table of contents of a printed manual. In Info, the title is underlined with periods. See section The `subsub' Commands.

`@subsubsection title`

Begin a subsubsection within a subsection. In a printed manual, the subsubsection title is numbered and appears in the table of contents. In Info, the title is underlined with periods. See section The `subsub' Commands.

`@subtitle title`

In a printed manual, set a subtitle in a normal sized font flush to the right-hand side of the page. Not relevant to Info, which does not have title pages. See section @title, @subtitle, and @author and @author Commands}.

`@summarycontents`

Print a short table of contents. Not relevant to Info, which uses menus rather than tables of contents. A synonym for @shortcontents. See section Generating a Table of Contents.

`@syncodeindex from-index into-index`

Merge the index named in the first argument into the index named in the second argument, printing the entries from the first index in @code font. See section Combining Indices.

`@synindex from-index into-index`

Merge the index named in the first argument into the index named in the second argument. Do not change the font of from-index entries. See section Combining Indices.

`@t{text}`

Print text in a fixed-width, typewriter-like font. No effect in Info. See section Fonts for Printing, Not Info.

`@table formatting-command`

Begin a two-column table, using @item for each entry. Write each first column entry on the same line as @item. First column entries are printed in the font resulting from formatting-command. Pair with @end table. See section Making a Two-column Table. Also see section @ftable and @vtable, and section @itemx.

`@TeX{}`

Insert the logo TeX. See section Inserting TeX and the Copyright Symbol.

`@tex`

Enter TeX completely. Pair with `@end tex`. See section Using Ordinary TeX Commands.

`@thischapter`

In a heading or footing, stands for the number and name of the current chapter, in the format `Chapter 1: Title'. See section How to Make Your Own Headings.

`@thischaptername`

In a heading or footing, stands for the name of the current chapter. See section How to Make Your Own Headings.

`@thisfile`

In a heading or footing, stands for the name of the current `@include` file. Does not insert anything if not within an `@include` file. See section How to Make Your Own Headings.

`@thispage`

In a heading or footing, stands for the current page number. See section How to Make Your Own Headings.

`@thistitle`

In a heading or footing, stands for the name of the document, as specified by the `@settitle` command. See section How to Make Your Own Headings.

`@tindex entry`

Add entry to the index of data types. See section Making Index Entries.

`@title title`

In a printed manual, set a title flush to the left-hand side of the page in a larger than normal font and underline it with a black rule. Not relevant to Info, which does not have title pages. See section `@title, @subtitle, and @author` `@subtitle` and `@author` Commands}.

`@titlefont{text}`

In a printed manual, print text in a larger than normal font. Not relevant to Info, which does not have title pages. See section `@titlefont, @center, and @sp` Commands}.

`@titlepage`

Indicate to Texinfo the beginning of the title page. Write command on a line of its own. Pair with `@end titlepage`. Nothing between `@titlepage` and `@end titlepage` appears in Info. See section `@titlepage`.

`@today{}`

Insert the current date, in `1 Jan 1900' style. See section How to Make Your Own Headings.

`@top title`

In a Texinfo file to be formatted with `makeinfo`, identify the topmost `@node` line in the file, which must be written on the line immediately preceding the `@top` command. Used for

`makeinfo`'s node pointer insertion feature. The title is underlined with asterisks. Both the `@n` line and the `@top` line normally should be enclosed by `@ifinfo` and `@end ifinfo`. In Te and `texinfo-format-buffer`, the `@top` command is merely a synonym for `@unnumbered`. See section Creating Pointers with `makeinfo`}.

`@unnumbered title`

In a printed manual, begin a chapter that appears without chapter numbers of any kind. The title appears in the table of contents of a printed manual. In Info, the title is underlined with asterisks. See section `@unnumbered`, `@appendix`}.

`@unnumberedsec title`

In a printed manual, begin a section that appears without section numbers of any kind. The title appears in the table of contents of a printed manual. In Info, the title is underlined with equal signs. See section `@unnumberedsec`, `@appendixsec`, `@heading`.

`@unnumberedsubsec title`

In a printed manual, begin an unnumbered subsection within a chapter. The title appears in the table of contents of a printed manual. In Info, the title is underlined with hyphens. See section The `@subsection`-like Commands `@appendixsubsec` `@subheading`}.

`@unnumberedsubsubsec title`

In a printed manual, begin an unnumbered subsubsection within a chapter. The title appears in the table of contents of a printed manual. In Info, the title is underlined with periods. See section The `subsub' Commands.

`@value{flag}`

Replace flag with the value to which it is set by `@set flag`. See section `@set`, `@clear`, and `@value`.

`@var{metasyntactic-variable}`

Highlight a metasyntactic variable, which is something that stands for another piece of text. See section `@var`{metasyntactic-variable}.

`@vindex entry`

Add entry to the index of variables. See section Making Index Entries.

`@vskip amount`

In a printed manual, insert whitespace so as to push text on the remainder of the page towards the bottom of the page. Used in formatting the copyright page with the argument `0pt plus 1filll'. (Note spelling of `filll'.) `@vskip` may be used only in contexts ignored for Info. See section Copyright Page and Permissions.

`@vtable formatting-command`

Begin a two-column table, using `@item` for each entry. Automatically enter each of the items in the first column into the index of variables. Pair with `@end vtable`. The same as `@table`, except for indexing. See section `@ftable` and `@vtable` and `@vtable`}.

`@w{text}`

Prevent text from being split across two lines. Do not end a paragraph that uses `@w` with an

`@refill` command. In the Texinfo file, keep text on one line. See section @w{text}: Prevent Breaks.

`@xref{node-name, [entry], [topic-or-title], [info-file], [manual]}`

Make a reference that starts with `See' in a printed manual. Follow command with a punctuation mark. Only the first argument is mandatory. See section `@xref`.

Go to the previous, next section.

Go to the previous, next section.

TOC

Tips and Hints

Here are some tips for writing Texinfo documentation:

- Write in the present tense, not in the past or the future.
- Write actively! For example, write "We recommend that ..." rather than "It is recommended that ...".
- Use 70 or 72 as your fill column. Longer lines are hard to read.
- Include a copyright notice and copying permissions.

Index, index, index!

Write many index entries, in different ways. Readers like indices; they are helpful and convenient.

Although it is easiest to write index entries as you write the body of the text, some people prefer to write entries afterwards. In either case, write an entry before the paragraph to which it applies. This way, an index entry points to the first page of a paragraph that is split across pages.

Here are more hints we have found valuable:

- Write each index entry differently, so each entry refers to a different place in the document. The index of an Info file lists only one location for each entry.
- Write index entries only where a topic is discussed significantly. For example, it is not useful to index "debugging information" in a chapter on reporting bugs. Someone who wants to know about debugging information will certainly not find it in that chapter.
- Consistently capitalize the first word of every index entry, or else use lower case. According to convention, you should capitalize the first word of an index entry. However, this practice may make an index look crowded. Some writers prefer lower case. Regardless of which you prefer, choose one style and stick to it. Mixing the two styles looks bad.
- Always capitalize or use upper case for those words in an index for which this is proper, such as names of countries or acronyms.
- Write the indexing commands that refer to a whole section immediately after the section command, and write the indexing commands that refer to the paragraph before the paragraph.

  In the example that follows, a blank line comes after the index entry for "Leaping":

```
@section The Dog and the Fox
@cindex Jumping, in general
@cindex Leaping

@cindex Dog, lazy, jumped over
@cindex Lazy dog jumped over
@cindex Fox, jumps over dog
```


```
@cindex Quick fox jumps over dog
The quick brown fox jumps over the lazy dog.
```

(Note that the example shows entries for the same concept that are written in different ways---`Lazy dog', and `Dog, lazy'---so readers can look up the concept in different ways.)

Blank lines

- Insert a blank line between a sectioning command and the first following sentence or paragraph, or between the indexing commands associated with the sectioning command and the first following sentence or paragraph, as shown in the tip on indexing. Otherwise, a formatter may fold title and paragraph together.
- Always insert a blank line before an `@table` command and after an `@end table` command; but never insert a blank line after an `@table` command or before an `@end table` command.

  For example,

```
Types of fox:

@table @samp
@item Quick
Jump over lazy dogs.

@item Brown
Also jump over lazy dogs.
@end table

@noindent
On the other hand, ...
```

  Insert blank lines before and after `@itemize` ... `@end itemize` and `@enumerate` ... `@end enumerate` in the same way.

Complete phrases

Complete phrases are easier to read than ...

- Write entries in an itemized list as complete sentences; or at least, as complete phrases. Incomplete expressions ... awkward ... like this.
- Write the prefatory sentence or phrase for a multi-item list or table as a complete expression. Do not write "You can set:"; instead, write "You can set these variables:". The former expression sounds cut off.

Editions, dates and versions

Write the edition and version numbers and date in three places in every manual:

1. In the first `@ifinfo` section, for people reading the Texinfo file.

2. In the `@titlepage` section, for people reading the printed manual.
3. In the `Top' node, for people reading the Info file.

Also, it helps to write a note before the first `@ifinfo` section to explain what you are doing.

For example:

```
@c ===> NOTE! <==
@c Specify the edition and version numbers and date
@c in *three* places:
@c   1. First ifinfo section  2. title page  3. top node
@c To find the locations, search for !!set

@ifinfo
@c !!set edition, date, version
This is Edition 4.03, January 1992,
of the @cite{GDB Manual} for GDB Version 4.3.
...
```

---or use `@set` and `@value` (see section `@value` Example).

Definition Commands

Definition commands are `@deffn`, `@defun`, `@defmac`, and the like, and enable you to write descriptions in a uniform format.

- Write just one definition command for each entity you define with a definition command. The automatic indexing feature creates an index entry that leads the reader to the definition.
- Use `@table` ... `@end table` in an appendix that contains a summary of functions, not `@deffn` or other definition commands.

Capitalization

- Capitalize `Texinfo'; it is a name. Do not write the `x' or `i' in upper case.
- Capitalize `Info'; it is a name.
- Write TeX using the `@TeX{}` command. Note the uppercase `T' and `X'. This command causes the formatters to typeset the name according to the wishes of Donald Knuth, who wrote TeX.

Spaces

Do not use spaces to format a Texinfo file, except inside of `@example` ... `@end example` and similar commands.

For example, TeX fills the following:

```
@kbd{C-x v}
@kbd{M-x vc-next-action}
```


```
Perform the next logical operation
on the version-controlled file
corresponding to the current buffer.
```

so it looks like this:

> C-x v M-x vc-next-action Perform the next logical operation on the version-controlled file corresponding to the current buffer.

In this case, the text should be formatted with `@table`, `@item`, and `@itemx`, to create a table.

@code, @samp, @var, and `---'

- Use `@code` around Lisp symbols, including command names. For example,

  ```
  The main function is @code{vc-next-action}, ...
  ```

- Avoid putting letters such as `s' immediately after an `@code'. Such letters look bad.
- Use `@var` around meta-variables. Do not write angle brackets around them.
- Use three hyphens in a row, `---', to indicate a long dash. TeX typesets these as a long dash and the Info formatters reduce three hyphens to two.

Periods Outside of Quotes

Place periods and other punctuation marks *outside* of quotations, unless the punctuation is part of the quotation. This practice goes against convention, but enables the reader to distinguish between the contents of the quotation and the whole passage.

For example, you should write the following sentence with the period outside the end quotation marks:

```
Evidently, `au' is an abbreviation for "author".
```

since `au' does *not* serve as an abbreviation for `author.' (with a period following the word).

Introducing New Terms

- Introduce new terms so that a user who does not know them can understand them from context; or write a definition for the term.

  For example, in the following, the terms "check in", "register" and "delta" are all appearing for the first time; the example sentence should be rewritten so they are understandable.

  > The major function assists you in checking in a file to your version control system and registering successive sets of changes to it as deltas.

- Use the `@dfn` command around a word being introduced, to indicate that the user should not expect to know the meaning already, and should expect to learn the meaning from this passage.

@pxref

Absolutely never use `@pxref` except in the special context for which it is designed: inside parenthe with the closing parenthesis following immediately after the closing brace. One formatter automatica inserts closing punctuation and the other does not. This means that the output looks right both in printed output and in an Info file, but only when the command is used inside parentheses.

Invoking from a Shell

You can invoke programs such as Emacs, GCC, and GAWK from a shell. The documentation for each program should contain a section that describes this. Unfortunately, if the node names and titles for these sections are all different, readers find it hard to search for the section.

Name such sections with a phrase beginning with the word `Invoking ...', as in `Invoking Emacs'; this way users can find the section easily.

ANSI C Syntax

When you use `@example` to describe a C function's calling conventions, use the ANSI C syntax, like this:

```
void dld_init (char *@var{path});
```

And in the subsequent discussion, refer to the argument values by writing the same argument names, again highlighted with `@var`.

Avoid the obsolete style that looks like this:

```
#include <dld.h>

dld_init (path)
char *path;
```

Also, it is best to avoid writing `#include` above the declaration just to indicate that the function is declared in a header file. The practice may give the misimpression that the `#include` belongs near the declaration of the function. Either state explicitly which header file holds the declaration or, better yet, name the header file used for a group of functions at the beginning of the section that describes the functions.

Bad Examples

Here are several examples of bad writing to avoid:

In this example, say, " ... you must `@dfn`{check in} the new version." That flows better.

> When you are done editing the file, you must perform a `@dfn`{check in}.

In the following example, say, "... makes a unified interface such as VC mode possible."

> SCCS, RCS and other version-control systems all perform similar functions in broadly similar ways (it is this resemblance which makes a unified control mode like this possible).

And in this example, you should specify what `it' refers to:

> If you are working with other people, it assists in coordinating everyone's changes so they do not step on each other.

And Finally ...

- Pronounce TeX as if the `X' were a Greek `chi', as the last sound in the name `Bach'. But pronounce Texinfo as in `speck': `teckinfo'.
- Write notes for yourself at the very end of a Texinfo file after the `@bye`. None of the formatters process text after the `@bye`; it is as if the text were within `@ignore` ... `@end ignore`.

Go to the previous, next section.

Go to the previous, next section.

A Sample Texinfo File

Here is a complete, short sample Texinfo file, without any commentary. You can see this file, with comments, in the first chapter. See section A Short Sample Texinfo File.

```
\input texinfo   @c -*-texinfo-*-
@c %**start of header
@setfilename sample.info
@settitle Sample Document
@c %**end of header

@setchapternewpage odd

@ifinfo
This is a short example of a complete Texinfo file.

Copyright 1990 Free Software Foundation, Inc.
@end ifinfo

@titlepage
@sp 10
@comment The title is printed in a large font.
@center @titlefont{Sample Title}

@c The following two commands start the copyright page.
@page
@vskip 0pt plus 1filll
Copyright @copyright{} 1990 Free Software Foundation, Inc.
@end titlepage

@node    Top,       First Chapter, (dir),    (dir)
@comment node-name, next,          previous, up

@menu
* First Chapter::    The first chapter is the
                     only chapter in this sample.
* Concept Index::    This index has two entries.
@end menu

@node    First Chapter, Concept Index, Top,      Top
@comment node-name,     next,          previous, up
@chapter First Chapter
```


```
@cindex Sample index entry

This is the contents of the first chapter.
@cindex Another sample index entry

Here is a numbered list.

@enumerate
@item
This is the first item.

@item
This is the second item.
@end enumerate

The @code{makeinfo} and @code{texinfo-format-buffer}
commands transform a Texinfo file such as this into
an Info file; and @TeX{} typesets it for a printed
manual.

@node    Concept Index,    ,  First Chapter, Top
@comment node-name,    next,  previous,      up
@unnumbered Concept Index

@printindex cp

@contents
@bye
```

Go to the previous, next section.

Go to the previous, next section.

Sample Permissions

Texinfo files should contain sections that tell the readers that they have the right to copy and distribute the Texinfo file, the Info file, and the printed manual.

Also, if you are writing a manual about software, you should explain that the software is free and either include the GNU General Public License (GPL) or provide a reference to it. See section 'Distribution' in The GNU Emacs Manual, for an example of the text that could be used in the software "Distribution", "General Public License", and "NO WARRANTY" sections of a document. See section Texinfo Copying Conditions, for an example of a brief explanation of how the copying conditions provide you with rights.

In a Texinfo file, the first `@ifinfo` section usually begins with a line that says what the file documents. This is what a person reading the unprocessed Texinfo file or using the advanced Info command g * sees first. @inforef{Expert, Advanced Info commands, info}, for more information. (A reader using the regular Info commands usually starts reading at the first node and skips this first section, which is not in a node.)

In the `@ifinfo` section, the summary sentence is followed by a copyright notice and then by the copying permission notice. One of the copying permission paragraphs is enclosed in `@ignore` and `@end ignore` commands. This paragraph states that the Texinfo file can be processed through TeX and printed, provided the printed manual carries the proper copying permission notice. This paragraph is not made part of the Info file since it is not relevant to the Info file; but it is a mandatory part of the Texinfo file since it permits people to process the Texinfo file in TeX and print the results.

In the printed manual, the Free Software Foundation copying permission notice follows the copyright notice and publishing information and is located within the region delineated by the `@titlepage` and `@end titlepage` commands. The copying permission notice is exactly the same as the notice in the `@ifinfo` section except that the paragraph enclosed in `@ignore` and `@end ignore` commands is not part of the notice.

To make it simple to insert a permission notice into each section of the Texinfo file, sample permission notices for each section are reproduced in full below.

Note that you may need to specify the correct name of a section mentioned in the permission notice. For example, in The GDB Manual, the name of the section referring to the General Public License is called the "GDB General Public License", but in the sample shown below, that section is referred to generically as the "GNU General Public License". If the Texinfo file does not carry a copy of the General Public License, leave out the reference to it, but be sure to include the rest of the sentence.

`ifinfo' Copying Permissions

In the `@ifinfo` section of a Texinfo file, the standard Free Software Foundation permission notice reads as follows:

```
This file documents ...

Copyright 1992 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim
copies of this manual provided the copyright notice and
this permission notice are preserved on all copies.

@ignore
Permission is granted to process this file through TeX
and print the results, provided the printed document
carries a copying permission notice identical to this
one except for the removal of this paragraph (this
paragraph not being relevant to the printed manual).

@end ignore
Permission is granted to copy and distribute modified
versions of this manual under the conditions for
verbatim copying, provided also that the sections
entitled "Copying" and "GNU General Public License"
are included exactly as in the original, and provided
that the entire resulting derived work is distributed
under the terms of a permission notice identical to this
one.

Permission is granted to copy and distribute
translations of this manual into another language,
under the above conditions for modified versions,
except that this permission notice may be stated in a
translation approved by the Free Software Foundation.
```

Titlepage Copying Permissions

In the `@titlepage` section of a Texinfo file, the standard Free Software Foundation copying permission notice follows the copyright notice and publishing information. The standard phrasing is as follows:

```
Permission is granted to make and distribute verbatim
```

Go to the previous, next section.

Go to the previous, next section.

Include Files

When TeX or an Info formatting command sees an `@include` command in a Texinfo file, it processes the contents of the file named by the command and incorporates them into the DVI or Info file being created. Index entries from the included file are incorporated into the indices of the output file.

Include files let you keep a single large document as a collection of conveniently small parts.

How to Use Include Files

To include another file within a Texinfo file, write the `@include` command at the beginning of a line and follow it on the same line by the name of a file to be included. For example:

```
@include buffers.texi
```

An included file should simply be a segment of text that you expect to be included as is into the overall or outer Texinfo file; it should not contain the standard beginning and end parts of a Texinfo file. In particular, you should not start an included file with a line saying `\input texinfo'; if you do, that phrase is inserted into the output file as is. Likewise, you should not end an included file with an `@bye` command; nothing after `@bye` is formatted.

In the past, you were required to write an `@setfilename` line at the beginning of an included file, but no longer. Now, it does not matter whether you write such a line. If an `@setfilename` line exists in an included file, it is ignored.

Conventionally, an included file begins with an `@node` line that is followed by an `@chapter` line. Each included file is one chapter. This makes it easy to use the regular node and menu creating and updating commands to create the node pointers and menus within the included file. However, the simple Emacs node and menu creating and updating commands do not work with multiple Texinfo files. Thus you cannot use these commands to fill in the `Next', `Previous', and `Up' pointers of the `@node` line that begins the included file. Also, you cannot use the regular commands to create a master menu for the whole file. Either you must insert the menus and the `Next', `Previous', and `Up' pointers by hand, or you must use the GNU Emacs Texinfo mode command, `texinfo-multiple-files-update`, that is designed for `@include` files.

texinfo-multiple-files-update

GNU Emacs Texinfo mode provides a command to handle included files called `texinfo-multiple-files-update`. This command creates or updates `Next', `Previous', and `Up' pointers of included files as well as those in the outer or overall Texinfo file, and it creates or updates a main menu in the outer file. Depending whether you call it with optional arguments, the command updates only the pointers in the first `@node` line of the included files or all of them:

M-x texinfo-multiple-files-update

Called without any arguments:

Create or update the `Next', `Previous', and `Up' pointers of the first `@node` line in each file included in an outer or overall Texinfo file.

Create or update the `Top' level node pointers of the outer or overall file.

Create or update a main menu in the outer file.

C-u M-x texinfo-multiple-files-update

Called with C-u as a prefix argument:

Create or update pointers in the first `@node` line in each included file.

Create or update the `Top' level node pointers of the outer file.

Create and insert a master menu in the outer file. The master menu is made from all the menus in all the included files.

C-u 8 M-x texinfo-multiple-files-update

Called with a numeric prefix argument, such as C-u 8:

Create or update **all** the `Next', `Previous', and `Up' pointers of all the included files.

Create or update **all** the menus of all the included files.

Create or update the `Top' level node pointers of the outer or overall file.

And then create a master menu in the outer file. This is similar to invoking `texinfo-master-menu` with an argument when you are working with just one file.

Note the use of the prefix argument in interactive use: with a regular prefix argument, just C-u, the `texinfo-multiple-files-update` command inserts a master menu; with a numeric prefix argument, such as C-u 8, the command updates **every** pointer and menu in **all** the files and then inserts a master menu.

Include File Requirements

If you plan to use the `texinfo-multiple-files-update` command, the outer Texinfo file that lists included files within it should contain nothing but the beginning and end parts of a Texinfo file, and a number of `@include` commands listing the included files. It should not even include indices, which should be listed in an included file of their own.

Moreover, each of the included files must contain exactly one highest level node (conventionally, `@chapter` or equivalent), and this node must be the first node in the included file. Furthermore, each of these highest level nodes in each included file must be at the same hierarchical level in the file structure. Usually, each is an `@chapter`, an `@appendix`, or an `@unnumbered` node. Thus, normally, each included file contains one, and only one, chapter or equivalent-level node.

The outer file should contain only *one* node, the `Top' node. It should *not* contain any nodes besides the single `Top' node. The `texinfo-multiple-files-update` command will not process them.

Sample File with `@include`

Here is an example of a complete outer Texinfo file with `@include` files within it before running `texinfo-multiple-files-update`, which would insert a main or master menu:

```
\input texinfo @c -*-texinfo-*-
@setfilename  include-example.info
@settitle Include Example

@setchapternewpage odd
@titlepage
@sp 12
@center @titlefont{Include Example}
@sp 2
@center by Whom Ever

@page
@vskip 0pt plus 1filll
Copyright @copyright{} 1990 Free Software Foundation, Inc.
@end titlepage

@ifinfo
@node Top, First, (dir), (dir)
@top Master Menu
@end ifinfo

@include foo.texinfo
@include bar.texinfo
@include concept-index.texinfo

@summarycontents
@contents

@bye
```

An included file, such as `` `foo.texinfo' ``, might look like this:

```
@node First, Second, , Top
@chapter First Chapter

Contents of first chapter ...
```

The full contents of `` `concept-index.texinfo' `` might be as simple as this:

```
@node Concept Index, , Second, Top
```


```
@unnumbered Concept Index

@printindex cp
```

The outer Texinfo source file for The GNU Emacs Lisp Reference Manual is named `elisp.texi'. This outer file contains a master menu with 417 entries and a list of 41 `@include` files.

Evolution of Include Files

When Info was first created, it was customary to create many small Info files on one subject. Each Info file was formatted from its own Texinfo source file. This custom meant that Emacs did not need to make a large buffer to hold the whole of a large Info file when someone wanted information; instead, Emacs allocated just enough memory for the small Info file that contained the particular information sought. This way, Emacs could avoid wasting memory.

References from one file to another were made by referring to the file name as well as the node name. (See section Referring to Other Info Files. Also, see section `@xref` with Four and Five Arguments with Four and Five Arguments}.)

Include files were designed primarily as a way to create a single, large printed manual out of several smaller Info files. In a printed manual, all the references were within the same document, so TeX could automatically determine the references' page numbers. The Info formatting commands used include files only for creating joint indices; each of the individual Texinfo files had to be formatted for Info individually. (Each, therefore, required its own `@setfilename` line.)

However, because large Info files are now split automatically, it is no longer necessary to keep them small.

Nowadays, multiple Texinfo files are used mostly for large documents, such as The GNU Emacs Lisp Reference Manual, and for projects in which several different people write different sections of a document simultaneously.

In addition, the Info formatting commands have been extended to work with the `@include` command so as to create a single large Info file that is split into smaller files if necessary. This means that you can write menus and cross references without naming the different Texinfo files.

Go to the previous, next section.

Go to the previous, next section.

TOC

Page Headings

Most printed manuals contain headings along the top of every page except the title and copyright pages. Some manuals also contain footings. (Headings and footings have no meaning to Info, which is not paginated.)

Texinfo provides standard page heading formats for manuals that are printed on one side of each sheet of paper and for manuals that are printed on both sides of the paper. Usually, you will use one or other of these formats, but you can specify your own format, if you wish.

In addition, you can specify whether chapters should begin on a new page, or merely continue the same page as the previous chapter; and if chapters begin on new pages, you can specify whether they must be odd-numbered pages.

By convention, a book is printed on both sides of each sheet of paper. When you open a book, the right-hand page is odd-numbered, and chapters begin on right-hand pages--a preceding left-hand page is left blank if necessary. Reports, however, are often printed on just one side of paper, and chapters begin on a fresh page immediately following the end of the preceding chapter. In short or informal reports, chapters often do not begin on a new page at all, but are separated from the preceding text by a small amount of whitespace.

The `@setchapternewpage` command controls whether chapters begin on new pages, and whether one of the standard heading formats is used. In addition, Texinfo has several heading and footing commands that you can use to generate your own heading and footing formats.

In Texinfo, headings and footings are single lines at the tops and bottoms of pages; you cannot create multiline headings or footings. Each header or footer line is divided into three parts: a left part, a middle part, and a right part. Any part, or a whole line, may be left blank. Text for the left part of a header or footer line is set flushleft; text for the middle part is centered; and, text for the right part is set flushright.

Standard Heading Formats

Texinfo provides two standard heading formats, one for manuals printed on one side of each sheet of paper, and the other for manuals printed on both sides of the paper.

By default, nothing is specified for the footing of a Texinfo file, so the footing remains blank.

The standard format for single-sided printing consists of a header line in which the left-hand part contains the name of the chapter, the central part is blank, and the right-hand part contains the page number.

A single-sided page looks like this:


```
|                          |
| chapter   page number    |
|                          |
| Start of text ...        |
| ...                      |
|                          |
```

The standard format for two-sided printing depends on whether the page number is even or odd. By convention, even-numbered pages are on the left- and odd-numbered pages are on the right. (TeX will adjust the widths of the left- and right-hand margins. Usually, widths are correct, but during double-sided printing, it is wise to check that pages will bind properly--sometimes a printer will produce output in which the even-numbered pages have a larger right-hand margin than the odd-numbered pages.)

In the standard double-sided format, the left part of the left-hand (even-numbered) page contains the page number, the central part is blank, and the right part contains the title (specified by the `@settitle` command). The left part of the right-hand (odd-numbered) page contains the name of the chapter, the central part is blank, and the right part contains the page number.

Two pages, side by side as in an open book, look like this:

```
 _________________________     _________________________
|                         |   |                         |
| page number     title   |   | chapter   page number   |
|                         |   |                         |
| Start of text ...       |   | More  text ...          |
| ...                     |   | ...                     |
|                         |   |                         |
```

The chapter name is preceded by the word `Chapter', the chapter number and a colon. This makes it easier to keep track of where you are in the manual.

Specifying the Type of Heading

TeX does not begin to generate page headings for a standard Texinfo file until it reaches the `@end titlepage` command. Thus, the title and copyright pages are not numbered. The `@end titlepage` command causes TeX to begin to generate page headings according to a standard format specified by the `@setchapternewpage` command that precedes the `@titlepage` section.

There are four possibilities:

No `@setchapternewpage` command

> Cause TeX to specify the single-sided heading format, with chapters on new pages. This is the same as `@setchapternewpage on`.

`@setchapternewpage on`

Specify the single-sided heading format, with chapters on new pages.

`@setchapternewpage off`

Cause TeX to start a new chapter on the same page as the last page of the preceding chapter, after skipping some vertical whitespace. Also cause TeX to typeset for single-sided printing. (You can override the headers format with the `@headings double` command; see section The `@headings` Command.)

`@setchapternewpage odd`

Specify the double-sided heading format, with chapters on new pages.

Texinfo lacks an `@setchapternewpage even` command.

How to Make Your Own Headings

You can use the standard headings provided with Texinfo or specify your own.

Texinfo provides six commands for specifying headings and footings. The `@everyheading` command and `@everyfooting` command generate page headers and footers that are the same for both even- and odd-numbered pages. The `@evenheading` command and `@evenfooting` command generate headers and footers for even-numbered (left-hand) pages; and the `@oddheading` command and `@oddfooting` command generate headers and footers for odd-numbered (right-hand) pages.

Write custom heading specifications in the Texinfo file immediately after the `@end titlepage` command. Enclose your specifications between `@iftex` and `@end iftex` commands since the `texinfo-format-buffer` command may not recognize them. Also, you must cancel the predefined heading commands with the `@headings off` command before defining your own specifications.

Here is how to tell TeX to place the chapter name at the left, the page number in the center, and the date at the right of every header for both even- and odd-numbered pages:

```
@iftex
@headings off
@everyheading @thischapter @| @thispage @| @today{}
@end iftex
```

You need to divide the left part from the central part and the central part from the right had part by inserting `@|' between parts. Otherwise, the specification command will not be able to tell where the text for one part ends and the next part begins.

Each part can contain text or @-commands. The text is printed as if the part were within an ordinary paragraph in the body of the page. The @-commands replace themselves with the page number, date, chapter name, or whatever.

Here are the six heading and footing commands:

```
@everyheading left @| center @| right
@everyfooting left @| center @| right
```

The `every' commands specify the format for both even- and odd-numbered pages. These commands are for documents that are printed on one side of each sheet of paper, or for documents in which you want symmetrical headers or footers.

```
@evenheading left @| center @| right
@oddheading left @| center @| right
@evenfooting left @| center @| right
@oddfooting left @| center @| right
```

The `even' and `odd' commands specify the format for even-numbered pages and odd-numbered pages. These commands are for books and manuals that are printed on both sides of each sheet of paper.

Use the `@this...' series of @-commands to provide the names of chapters and sections and the page number. You can use the `@this...' commands in the left, center, or right portions of headers and footers, or anywhere else in a Texinfo file so long as they are between `@iftex` and `@end iftex` commands.

Here are the `@this...' commands:

`@thispage`

Expands to the current page number.

`@thischaptername`

Expands to the name of the current chapter.

`@thischapter`

Expands to the number and name of the current chapter, in the format `Chapter 1: Title'.

`@thistitle`

Expands to the name of the document, as specified by the `@settitle` command.

`@thisfile`

For `@include` files only: expands to the name of the current `@include` file. If the current Texinfo source file is not an `@include` file, this command has no effect. This command does *not* provide the name of the current Texinfo source file unless it is an `@include` file. (See section Include Files, for more information about `@include` files.)

You can also use the `@today{}` command, which expands to the current date, in `1 Jan 1900' format.

Other @-commands and text are printed in a header or footer just as if they were in the body of a page. It is useful to incorporate text, particularly when you are writing drafts:

```
@iftex
@headings off
@everyheading @emph{Draft!} @| @thispage @| @thischapter
@everyfooting @| @| Version: 0.27: @today{}
@end iftex
```

Beware of overlong titles: they may overlap another part of the header or footer and blot it out.

Go to the previous, next section.

Go to the previous, next section.

Formatting Mistakes

Besides mistakes in the content of your documentation, there are two kinds of mistake you can make with Texinfo: you can make mistakes with @-commands, and you can make mistakes with the structure of the nodes and chapters.

Emacs has two tools for catching the @-command mistakes and two for catching structuring mistakes.

For finding problems with @-commands, you can run TeX or a region formatting command on the region that has a problem; indeed, you can run these commands on each region as you write it.

For finding problems with the structure of nodes and chapters, you can use C-c C-s (`texinfo-show-structure`) and the related `occur` command and you can use the M-x Info-validate command.

The `makeinfo` program does an excellent job of catching errors and reporting them--far better than either the `texinfo-format-region` or the `texinfo-format-buffer` command. In addition, the various functions for automatically creating and updating node pointers and menus remove many opportunities for human error.

If you can, use the updating commands to create and insert pointers and menus. These prevent many errors. Then use `makeinfo` (or its Texinfo mode manifestations, `makeinfo-region` and `makeinfo-buffer`) to format your file and check for other errors. This is the best way to work with Texinfo. But if you cannot use `makeinfo`, or your problem is very puzzling, then you may want to use the tools described in this appendix.

Catching Errors with Info Formatting

After you have written part of a Texinfo file, you can use the `texinfo-format-region` or the `makeinfo-region` command to see whether the region formats properly.

Most likely, however, you are reading this section because for some reason you cannot use the `makeinfo-region` command; therefore, the rest of this section presumes that you are using `texinfo-format-region`.

If you make a mistake with an @-command, `texinfo-format-region` will stop processing at or after the error and display an error message. To see where in the buffer the error occurred, switch to the `*Info Region*' buffer; the cursor will be in a position that is after the location of the error. Also, the text will not be formatted after the place where the error occurred (or more precisely, where it was detected).

For example, if you accidentally end a menu with the command `@end menus` with an `s' on the end, instead of with `@end menu`, you will see an error message that says:

```
@end menus is not handled by texinfo
```

The cursor will stop at the point in the buffer where the error occurs, or not long after it. The buffer will look like this:

```
---------- Buffer: *Info Region* ----------
* Menu:

* Using texinfo-show-structure::  How to use
                                  `texinfo-show-structure'
                                  to catch mistakes.
* Running Info-Validate::         How to check for
                                  unreferenced nodes.
@end menus
-!-
---------- Buffer: *Info Region* ----------
```

The `texinfo-format-region` command sometimes provides slightly odd error messages. For example, the following cross reference fails to format:

```
(@xref{Catching Mistakes, for more info.)
```

In this case, `texinfo-format-region` detects the missing closing brace but displays a message that says `Unbalanced parentheses' rather than `Unbalanced braces'. This is because the formatting command looks for mismatches between braces as if they were parentheses.

Sometimes `texinfo-format-region` fails to detect mistakes. For example, in the following, the closing brace is swapped with the closing parenthesis:

```
(@xref{Catching Mistakes), for more info.}
```

Formatting produces:

```
(*Note for more info.: Catching Mistakes)
```

The only way for you to detect this error is to realize that the reference should have looked like this:

```
(*Note Catching Mistakes::, for more info.)
```

Incidentally, if you are reading this node in Info and type f RET (`Info-follow-reference`), you will generate an error message that says:

```
No such node: "Catching Mistakes) The only way ...
```

This is because Info perceives the example of the error as the first cross reference in this node and if you type a RET immediately after typing the Info f command, Info will attempt to go to the referenced node. If you type f catch TAB RET, Info will complete the node name of the correctly written example and take you to the `Catching Mistakes' node. (If you try this, you can return from the `Catching Mistakes'

node by typing l (`Info-last`).)

Catching Errors with TeX Formatting

You can also catch mistakes when you format a file with TeX.

Usually, you do this after you have run `texinfo-format-buffer` (or, better, `makeinfo-buffer`) on the same file, because `texinfo-format-buffer` sometimes displays error messages that make more sense than TeX. (See section Catching Errors with Info Formatting, for more information.)

For example, TeX was run on a Texinfo file, part of which is shown here:

```
---------- Buffer: texinfo.texi ----------
name of the texinfo file as an extension.  The
@samp{??} are `wildcards' that cause the shell to
substitute all the raw index files.  (@xref{sorting
indices, for more information about sorting
indices.)@refill
---------- Buffer: texinfo.texi ----------
```

(The cross reference lacks a closing brace.) TeX produced the following output, after which it stopped:

```
---------- Buffer: *texinfo-tex-shell* ----------
Runaway argument?
{sorting indices, for more information about sorting
indices.) @refill @ETC.
! Paragraph ended before @xref was complete.
<to be read again>
                   @par
l.27

?
---------- Buffer: *texinfo-tex-shell* ----------
```

In this case, TeX produced an accurate and understandable error message:

```
Paragraph ended before @xref was complete.
```

`@par' is an internal TeX command of no relevance to Texinfo. `l.27' means that TeX detected the problem on line 27 of the Texinfo file. The `?' is the prompt TeX uses in this circumstance.

Unfortunately, TeX is not always so helpful, and sometimes you must truly be a Sherlock Holmes to discover what went wrong.

In any case, if you run into a problem like this, you can do one of three things.

1. You can tell TeX to continue running and ignore just this error by typing RET at the `?' prompt.
2. You can tell TeX to continue running and to ignore all errors as best it can by typing r RET at the `?' prompt.

   This is often the best thing to do. However, beware: the one error may produce a cascade of additional error messages as its consequences are felt through the rest of the file. (To stop TeX when it is producing such an avalanche of error messages, type C-d (or C-c C-d, if you are running a shell inside Emacs Version 18.))
3. You can tell TeX to stop this run by typing x RET at the `?' prompt.

Please note that if you are running TeX inside Emacs, you need to switch to the shell buffer and line at which TeX offers the `?' prompt.

Sometimes TeX will format a file without producing error messages even though there is a problem. This usually occurs if a command is not ended but TeX is able to continue processing anyhow. For example, if you fail to end an itemized list with the `@end itemize` command, TeX will write a DVI file that you can print out. The only error message that TeX will give you is the somewhat mysterious comment that

```
(@end occurred inside a group at level 1)
```

However, if you print the DVI file, you will find that the text of the file that follows the itemized list is entirely indented as if it were part of the last item in the itemized list. The error message is the way TeX says that it expected to find an `@end` command somewhere in the file; but that it could not determine where it was needed.

Another source of notoriously hard-to-find errors is a missing `@end group` command. If you ever are stumped by incomprehensible errors, look for a missing `@end group` command first.

If the Texinfo file lacks header lines, TeX may stop in the beginning of its run and display output that looks like the following. The `*' indicates that TeX is waiting for input.

```
This is TeX, Version 2.0 for Berkeley UNIX
(preloaded format=plain-cm 87.10.25)
(test.texinfo [1])
*
```

In this case, simply type \end RET after the asterisk. Then write the header lines in the Texinfo file and run the TeX command again. (Note the use of the backslash, `\'. TeX uses `\' instead of `@'; and in this circumstance, you are working directly with TeX, not with Texinfo.)

Using `texinfo-show-structure`

It is not always easy to keep track of the nodes, chapters, sections, and subsections of a Texinfo file. This is especially true if you are revising or adding to a Texinfo file that someone else has written.

In GNU Emacs, in Texinfo mode, the `texinfo-show-structure` command lists all the lines that begin with the @-commands that specify the structure: `@chapter`, `@section`, `@appendix`, and so

on. With an argument (C-u as prefix argument, if interactive), the command also shows the `@node` l
The `texinfo-show-structure` command is bound to C-c C-s in Texinfo mode, by default.

The lines are displayed in a buffer called the `*Occur*' buffer. For example, when `texinfo-show-structure` was run on an earlier version of this appendix, it produced the following:

```
Lines matching "^@\\(chapter \\|sect\\|sub\\|unnum\\|major\\|
heading \\|appendix\\)" in buffer texinfo.texi.
  4:@appendix Formatting Mistakes
 52:@appendixsec Catching Errors with Info Formatting
222:@appendixsec Catching Errors with @TeX{} Formatting
338:@appendixsec Using @code{texinfo-show-structure}
407:@appendixsubsec Using @code{occur}
444:@appendixsec Finding Badly Referenced Nodes
513:@appendixsubsec Running @code{Info-validate}
573:@appendixsubsec Splitting a File Manually
```

This says that lines 4, 52, and 222 of `` `texinfo.texi' `` begin with the `@appendix`, `@appendixsec`, and `@appendixsec` commands respectively. If you move your cursor into the `*Occur*' window, you can position the cursor over one of the lines and use the C-c C-c command (`occur-mode-goto-occurrence`), to jump to the corresponding spot in the Texinfo file. See section 'Using Occur' in The GNU Emacs Manual, for more information about `occur-mode-goto-occurrence`.

The first line in the `*Occur*' window describes the regular expression specified by texinfo-heading-pattern. This regular expression is the pattern that `texinfo-show-structure` looks for. See section 'Using Regular Expressions' in The GNU Emacs Manual, for more information.

When you invoke the `texinfo-show-structure` command, Emacs will display the structure of the whole buffer. If you want to see the structure of just a part of the buffer, of one chapter, for example, use the C-x n (`narrow-to-region`) command to mark the region. (See section 'Narrowing' in The GNU Emacs Manual.) This is how the example used above was generated. (To see the whole buffer again, use C-x w (`widen`).)

If you call `texinfo-show-structure` with a prefix argument by typing C-u C-c C-s, it will list lines beginning with `@node` as well as the lines beginning with the @-sign commands for `@chapter`, `@section`, and the like.

You can remind yourself of the structure of a Texinfo file by looking at the list in the `*Occur*' window; and if you have mis-named a node or left out a section, you can correct the mistake.

Using occur

Sometimes the `texinfo-show-structure` command produces too much information. Perhaps you want to remind yourself of the overall structure of a Texinfo file, and are overwhelmed by the detailed list produced by `texinfo-show-structure`. In this case, you can use the `occur` command

directly. To do this, type

```
M-x occur
```

and then, when prompted, type a regexp, a regular expression for the pattern you want to match. (See section 'Regular Expressions' in The GNU Emacs Manual.) The `occur` command works from the current location of the cursor in the buffer to the end of the buffer. If you want to run `occur` on the whole buffer, place the cursor at the beginning of the buffer.

For example, to see all the lines that contain the word `@chapter' in them, just type `@chapter'. This will produce a list of the chapters. It will also list all the sentences with `@chapter' in the middle of the line.

If you want to see only those lines that start with the word `@chapter', type `^@chapter' when prompted by `occur`. If you want to see all the lines that end with a word or phrase, end the last word with a `$'; for example, `catching mistakes$'. This can be helpful when you want to see all the nodes that are part of the same chapter or section and therefore have the same `Up' pointer.

See section 'Using Occur' in The GNU Emacs Manual, for more information.

Finding Badly Referenced Nodes

You can use the `Info-validate` command to check whether any of the `Next', `Previous', `Up' or other node pointers fail to point to a node. This command checks that every node pointer points to an existing node. The `Info-validate` command works only on Info files, not on Texinfo files.

The `makeinfo` program validates pointers automatically, so you do not need to use the `Info-validate` command if you are using `makeinfo`. You only may need to use `Info-validate` if you are unable to run `makeinfo` and instead must create an Info file using `texinfo-format-region` or `texinfo-format-buffer`, or if you write an Info file from scratch.

Running `Info-validate`

To use `Info-validate`, visit the Info file you wish to check and type:

```
M-x Info-validate
```

(Note that the `Info-validate` command requires an upper case `I'. You may also need to create a tag table before running `Info-validate`. See section Tagifying a File.)

If your file is valid, you will receive a message that says "File appears valid". However, if you have a pointer that does not point to a node, error messages will be displayed in a buffer called `*problems in info file*'.

For example, `Info-validate` was run on a test file that contained only the first node of this manual. One of the messages said:


```
In node "Overview", invalid Next: Texinfo Mode
```

This meant that the node called `Overview' had a `Next' pointer that did not point to anything (which was true in this case, since the test file had only one node in it).

Now suppose we add a node named `Texinfo Mode' to our test case but we do not specify a `Previous' for this node. Then we will get the following error message:

```
In node "Texinfo Mode", should have Previous: Overview
```

This is because every `Next' pointer should be matched by a `Previous' (in the node where the `Next' points) which points back.

`Info-validate` also checks that all menu entries and cross references point to actual nodes.

Note that `Info-validate` requires a tag table and does not work with files that have been split. (The `texinfo-format-buffer` command automatically splits large files.) In order to use `Info-validate` on a large file, you must run `texinfo-format-buffer` with an argument so that it does not split the Info file; and you must create a tag table for the unsplit file.

Creating an Unsplit File

You can run `Info-validate` only on a single Info file that has a tag table. The command will not work on the indirect subfiles that are generated when a master file is split. If you have a large file (longer than 70,000 bytes or so), you need to run the `texinfo-format-buffer` or `makeinfo-buffer` command in such a way that it does not create indirect subfiles. You will also need to create a tag table for the Info file. After you have done this, you can run `Info-validate` and look for badly referenced nodes.

The first step is to create an unsplit Info file.

To prevent `texinfo-format-buffer` from splitting a Texinfo file into smaller Info files, give a prefix to the M-x texinfo-format-buffer command:

```
C-u M-x texinfo-format-buffer
```

or else

```
C-u C-c C-e C-b
```

When you do this, Texinfo will not split the file and will not create a tag table for it.

Tagifying a File

After creating an unsplit Info file, you must create a tag table for it. Visit the Info file you wish to tagify and type:

```
M-x Info-tagify
```

(Note the upper case I in `Info-tagify`.) This creates an Info file with a tag table that you can vali

The third step is to validate the Info file:

```
M-x Info-validate
```

(Note the upper case I in `Info-validate`.) In brief, the steps are:

```
C-u M-x texinfo-format-buffer
M-x Info-tagify
M-x Info-validate
```

After you have validated the node structure, you will be able to rerun `texinfo-format-buffer` in the normal way so it will construct a tag table and split the file automatically, or you can make the tag table and split the file manually.

Splitting a File Manually

You should split a large file or else let the `texinfo-format-buffer` or `makeinfo-buffer` command do it for you automatically. (Generally you will let one of the formatting commands do this job for you. See section Creating an Info File.)

The split-off files are called the indirect subfiles.

Info files are split to save memory. With smaller files, Emacs does not have make such a large buffer to hold the information.

If an Info file has more than 30 nodes, you should also make a tag table for it. See section Running `Info-validate`, for information about creating a tag table. (Again, tag tables are usually created automatically by the formatting command; you only need to create a tag table yourself if you are doing the job manually. Most likely, you will do this for a large, unsplit file on which you have run `Info-validate`.)

Visit the Info file you wish to tagify and split and type the two commands:

```
M-x Info-tagify
M-x Info-split
```

(Note that the `I' in `Info' is upper case.)

When you use the `Info-split` command, the buffer is modified into a (small) Info file which lists the indirect subfiles. This file should be saved in place of the original visited file. The indirect subfiles are written in the same directory the original file is in, with names generated by appending `-' and a number to the original file name.

The primary file still functions as an Info file, but it contains just the tag table and a directory of subfiles.

Go to the previous, next section.

Go to the previous, next section.

Refilling Paragraphs

The `@refill` command refills and, optionally, indents the first line of a paragraph.(12) The `@refill` command is no longer important, but we describe it here because you once needed it. You will see it in many old Texinfo files.

Without refilling, paragraphs containing long @-constructs may look bad after formatting because the formatter removes @-commands and shortens some lines more than others. In the past, neither `texinfo-format-region` nor `texinfo-format-buffer` refilled paragraphs automatically. The `@refill` command had to be written at the end of every paragraph to cause these formatters to fill them. (Both TeX and `makeinfo` have always refilled paragraphs automatically.) Now, all the Info formatters automatically fill and indent those paragraphs that need to be filled and indented.

The `@refill` command causes both the `texinfo-format-region` command and the `texinfo-format-buffer` command to refill a paragraph in the Info file *after* all the other processing has been done. For this reason, you can not use `@refill` with a paragraph containing either `@*` or `@w{ ... }` since the refilling action will override those two commands.

The `texinfo-format-region` and `texinfo-format-buffer` commands now automatically append `@refill` to the end of each paragraph that should be filled. They do not append `@refill` to the ends of paragraphs that contain `@*` or `@w{ ...}` and therefore do not refill or indent them.

Go to the previous, next section.

Go to the previous, next section.

@-Command Syntax

The character `@' is used to start special Texinfo commands. (It has the same meaning that `\' has in PlainTeX.) Texinfo has four types of @-command:

1. Non-alphabetic commands.

   These commands consist of an @ followed by a punctuation mark or other character that is not part of the alphabet. Non-alphabetic commands are almost always part of the text within a paragraph, and never take any argument. The two characters (@ and the other one) are complete in themselves; none is followed by braces. The non-alphabetic commands are: `@.`, `@:`, `@*`, `@@`, `@{`, and `@}`.

2. Alphabetic commands that do not require arguments.

   These commands start with @ followed by a word followed by left- and right-hand braces. These commands insert special symbols in the document; they do not require arguments. For example, `@dots{}` => `...', `@equiv{}` => `==', `@TeX{}` => `TeX', and `@bullet{}` => `*'.

3. Alphabetic commands that require arguments within braces.

   These commands start with @ followed by a letter or a word, followed by an argument within braces. For example, the command `@dfn` indicates the introductory or defining use of a term; it is used as follows: `In Texinfo, @@-commands are @dfn{mark-up} commands.'

4. Alphabetic commands that occupy an entire line.

   These commands occupy an entire line. The line starts with @, followed by the name of the command (a word); for example, `@center` or `@cindex`. If no argument is needed, the word is followed by the end of the line. If there is an argument, it is separated from the command name by a space. Braces are not used.

Thus, the alphabetic commands fall into classes that have different argument syntaxes. You cannot tell to which class a command belongs by the appearance of its name, but you can tell by the command's meaning: if the command stands for a glyph, it is in class 2 and does not require an argument; if it makes sense to use the command together with other text as part of a paragraph, the command is in class 3 and must be followed by an argument in braces; otherwise, it is in class 4 and uses the rest of the line as its argument.

The purpose of having a different syntax for commands of classes 3 and 4 is to make Texinfo files easier to read, and also to help the GNU Emacs paragraph and filling commands work properly. There is only one exception to this rule: the command `@refill`, which is always used at the end of a paragraph immediately following the final period or other punctuation character. `@refill` takes no argument and does *not* require braces. `@refill` never confuses the Emacs paragraph commands because it cannot appear at the beginning of a line.

Go to the previous, next section.

Go to the previous, next section.

How to Obtain TeX

TeX is freely redistributable. You can obtain TeX for Unix systems from the University of Washington for a distribution fee.

To order a full distribution, send $200.00 for a 1/2-inch 9-track 1600 bpi (`tar` or `cpio`) tape reel, or $210.00 for a 1/4-inch 4-track QIC-24 (`tar` or `cpio`) cartridge, to:

```
Northwest Computing Support Center
DR-10, Thomson Hall 35
University of Washington
Seattle, Washington 98195
```

Please make checks payable to the University of Washington.

Prepaid orders are preferred but purchase orders are acceptable; however, purchase orders carry an extra charge of $10.00, to pay for processing.

Overseas sites: please add to the base cost $20.00 for shipment via air parcel post, or $30.00 for shipment via courier.

Please check with the Northwest Computing Support Center at the University of Washington for current prices and formats:

```
telephone:  (206) 543-6259
email:      elisabet@u.washington.edu
```

Go to the previous, next section.

Go to the previous, next section.

Second Edition Features

The second edition of the Texinfo manual describes more than 20 new Texinfo mode commands and more than 50 previously undocumented Texinfo @-commands. This edition is more than twice the length of the first edition.

Here is a brief description of the new commands.

New Texinfo Mode Commands

Texinfo mode provides commands and features especially designed for working with Texinfo files. More than 20 new commands have been added, including commands for automatically creating and updating both nodes and menus. This is a tedious task when done by hand.

The keybindings are intended to be somewhat mnemonic.

Update all nodes and menus

The `texinfo-master-menu` command is the primary command:

C-c C-u m

M-x texinfo-master-menu

> Create or update a master menu. With C-u as a prefix argument, first create or update all nodes and regular menus.

Update Pointers

Create or update `Next', `Previous', and `Up' node pointers.

See section Updating Nodes and Menus.

C-c C-u C-n

M-x texinfo-update-node

> Update a node.

C-c C-u C-e

M-x texinfo-every-node-update

> Update every node in the buffer.

Update Menus

Create or update menus.

See section Updating Nodes and Menus.

C-c C-u C-m

M-x texinfo-make-menu

Make or update a menu.

C-c C-u C-a

M-x texinfo-all-menus-update

Make or update all the menus in a buffer. With C-u as a prefix argument, first update all the nodes.

Insert Title as Description

Insert a node's chapter or section title in the space for the description in a menu entry line; position point so you can edit the insert. (This command works somewhat differently than the other insertion commands, which insert only a predefined string.)

See section Inserting Frequently Used Commands.

C-c C-c C-d

Insert title.

Format for Info

Provide keybindings both for the Info formatting commands that are written in Emacs Lisp and for `makeinfo` that is written in C.

See section Formatting for Info.

Use the Emacs lisp `texinfo-format...` commands:

C-c C-e C-r

Format the region.

C-c C-e C-b

Format the buffer.

Use `makeinfo`:

C-c C-m C-r

Format the region.

C-c C-m C-b

Format the buffer.

C-c C-m C-l

Recenter the `makeinfo` output buffer.

C-c C-m C-k
: Kill the `makeinfo` formatting job.

Typeset and Print

Typeset and print Texinfo documents from within Emacs.

See section Formatting and Printing.

C-c C-t C-r
: Run TeX on the region.

C-c C-t C-b
: Run TeX on the buffer.

C-c C-t C-i
: Run `texindex`.

C-c C-t C-p
: Print the DVI file.

C-c C-t C-q
: Show the print queue.

C-c C-t C-d
: Delete a job from the print queue.

C-c C-t C-k
: Kill the current TeX formatting job.

C-c C-t C-x
: Quit a currently stopped TeX formatting job.

C-c C-t C-l
: Recenter the output buffer.

Other Updating Commands

The "other updating commands" do not have standard keybindings because they are used less frequently.

See section Other Updating Commands.

M-x texinfo-insert-node-lines
: Insert missing `@node` lines using section titles as node names.

M-x texinfo-multiple-files-update
: Update a multi-file document. With a numeric prefix, such as C-u 8, update **every** pointer and menu in **all** the files and then insert a master menu.

M-x texinfo-indent-menu-description
: Indent descriptions in menus.

M-x texinfo-sequential-node-update

Insert node pointers in strict sequence.

New Texinfo @-Commands

The second edition of the Texinfo manual describes more than 50 commands that were not described in the first edition. A third or so of these commands existed in Texinfo but were not documented in the manual; the others are new. Here is a listing, with brief descriptions of them:

Indexing

Create your own index, and merge indices.

See section Creating Indices.

@defindex index-name

Define a new index and its indexing command. See also the `@defcodeindex` command.

@synindex from-index into-index

Merge the from-index index into the into-index index. See also the `@syncodeindex` command.

Definitions

Describe functions, variables, macros, commands, user options, special forms, and other such artifacts in a uniform format.

See section Definition Commands.

@deffn category name arguments...

Format a description for functions, interactive commands, and similar entities.

@defvr, @defop, ...

15 other related commands.

Glyphs

Indicate the results of evaluation, expansion, printed output, an error message, equivalence of expressions, and the location of point.

See section Glyphs for Examples.

@equiv{}

==

Equivalence:

@error{}

error-->

Error message

@expansion{}
==>
: Macro expansion

@point{}
-!-
: Position of point

@print{}
-|
: Printed output

@result{}
=>
: Result of an expression

Page Headings

Customize page headings.

See section Page Headings.

@headings on-off-single-double
: Headings on or off, single, or double-sided.

@evenfooting [left] @| [center] @| [right]
: Footings for even-numbered (left-hand) pages.

@evenheading, @everyheading, @oddheading, ...
: Five other related commands.

@thischapter
: Insert name of chapter and chapter number.

@thischaptername, @thisfile, @thistitle, @thispage
: Related commands.

Formatting

Format blocks of text.

See section Quotations and Examples, and
section Making Lists and Tables.

@cartouche
: Draw rounded box surrounding text (not in Info).

@enumerate optional-arg
: Enumerate a list with letters or numbers.

@exdent line-of-text

Remove indentation.

@flushleft

Left justify.

@flushright

Right justify.

@format

Do not narrow nor change font.

@ftable formatting-command

@vtable formatting-command

Two-column table with indexing.

@lisp

For an example of Lisp code.

@smallexample

@smalllisp

Like @table and @lisp but for @smallbook.

Conditionals

Conditionally format text.

See section `@set`, `@clear`, and `@value`.

@set flag [string]

Set a flag. Optionally, set value of flag to string.

@clear flag

Clear a flag.

@value{flag}

Replace with value to which flag is set.

@ifset flag

Format, if flag is set.

@ifclear flag

Ignore, if flag is set.

@heading series for Titles

Produce unnumbered headings that do not appear in a table of contents.

See section Chapter Structuring.

@heading title

Unnumbered section-like heading not listed in the table of contents of a printed manual.

@chapheading, @majorheading, @subheading, @subsubheading

Related commands.

Font commands

See section @sc{text}: The Small Caps Font, and
section Fonts for Printing, Not Info.

@r{text}

Print in roman font.

@sc{text}

Print in SMALL CAPS font.

Miscellaneous

See section @title, @subtitle, and @author,
see section Overfull "hboxes",
see section Footnotes,
see section @dmn{dimension}: Format a Dimension,
see section @minus{}: Inserting a Minus Sign,
see section Paragraph Indenting,
see section Different Cross Reference Commands,
see section @title, @subtitle, and @author, and
see section How to Make Your Own Headings.

@author author

Typeset author's name.

@finalout

Produce cleaner printed output.

@footnotestyle

Specify footnote style.

@dmn{dimension}

Format a dimension.

@minus{}

Generate a minus sign.

@paragraphindent

Specify paragraph indentation.

@ref{node-name, [entry], [topic-or-title], [info-file], [manual]}

Make a reference. In the printed manual, the reference does not start with the word `see'.

@title title

Typeset title in the alternative title page format.

@subtitle subtitle

Typeset subtitle in the alternative title page format.

@today{}

Insert the current date.

Go to the previous, next section.

Go to the previous, next section.

Command and Variable Index

This is an alphabetical list of all the @-commands and several variables. To make the list easier to use, the commands are listed without their preceding `@'.

b

c

d

e

f

g

h

i

k

l

m

n

o

p

q

r

s

t

u

Go to the previous, next section.

Go to the previous section.

Concept Index

.

@

a

b

C

d

e

f

g

h

i

k

l

m

n

o

p

q

r

S

t

u

V

W

Go to the previous section.

Texinfo

(1)

Note that the first syllable of "Texinfo" is pronounced like "speck", not "hex". This odd pronunciation is derived from, but is not the same as, the pronunciation of TeX. In the word TeX, the `X' is actually the Greek letter "chi" rather than the English letter "ex". Pronounce TeX as if the `X' were the last sound in the name `Bach'; but pronounce Texinfo as if the `x' were a `k'. Spell "Texinfo" with a capital "T" and write the other letters in lower case.

(2)

In some documents, the first child has no `Previous' pointer. Occasionally, the last child has the node name of the next following higher level node as its `Next' pointer.

(3)

You can also use the `texi2roff` program if you do not have TeX; since Texinfo is designed for use with TeX, `texi2roff` is not described here. `texi2roff` is part of the standard GNU distribution.

(4)

The word argument comes from the way it is used in mathematics and does not refer to a disputation between two people; it refers to the information presented to the command. According to the Oxford English Dictionary, the word derives from the Latin for to make clear, prove; thus it came to mean `the evidence offered as proof', which is to say, `the information offered', which led to its mathematical meaning. In its other thread of derivation, the word came to mean `to assert in a manner against which others may make counter assertions', which led to the meaning of `argument' as a disputation.

(5)

We have found that it is helpful to refer to versions of manuals as `editions' and versions of programs as `versions'; otherwise, we find we are liable to confuse each other in conversation by referring to both the documentation and the software with the same words.

(6)

Menus can carry you to any node, regardless of the hierarchical structure; even to nodes in a different Info file. However, the GNU Emacs Texinfo mode updating commands work only to create menus of subordinate nodes. Conventionally, cross references are used to refer to other nodes.

(7)

It would be straightforward to extend Texinfo to work in a similar fashion for C, FORTRAN, or other languages.

(8)

A footnote should complement or expand upon the primary text, but a reader should not need to read a footnote to understand the primary text. For a thorough discussion of footnotes, see The Chicago Manual of Style, which is published by the University of Chicago Press.

(9)

Here is the sample footnote.

(10)

If you use more than one index and have cross references to an index other than the first, you must run `tex` *three times* to get correct output: once to generate raw index data; again (after `texindex`) to output the text of the indices and determine their true page numbers; and a third time to output correct page numbers in cross references to them. However, cross references to indices are rare.

(11)

`--' has replaced `+', the old introductory character, to maintain POSIX.2 compatibility without losing long-named options.

(12)

Perhaps the command should have been called the `@refillandindent` command, but `@refill` is shorter and the name was chosen before indenting was possible.

GNU textutils

A set of text utilities

for version 1.19, 27 June 1996

David MacKenzie et al.

Go to the next section.

@defcodeindex op

Introduction

This manual is incomplete: No attempt is made to explain basic concepts in a way suitable for novices. Thus, if you are interested, please get involved in improving this manual. The entire GNU community will benefit.

The GNU text utilities are mostly compatible with the POSIX.2 standard.

Please report bugs to `bug-gnu-utils@prep.ai.mit.edu'. Remember to include the version number, machine architecture, input files, and any other information needed to reproduce the bug: your input, what you expected, what you got, and why it is wrong. Diffs are welcome, but please include a description of the problem as well, since this is sometimes difficult to infer. See section 'Bugs' in GNU CC.

This manual is based on the Unix man pages in the distribution, which were originally written by David MacKenzie and updated by Jim Meyering. The original `fmt` man page was written by Ross Paterson. Pinard did the initial conversion to Texinfo format. Karl Berry did the indexing, some reorganization, and editing of the results. Richard Stallman contributed his usual invaluable insights to the overall process.

Go to the next section.

Go to the previous, next section.

Common options

Certain options are available in all these programs. Rather than writing identical descriptions for each of the programs, they are described here. (In fact, every GNU program accepts (or should accept) these options.)

A few of these programs take arbitrary strings as arguments. In those cases, `--help' and `--version' are taken as these options only if there is one and exactly one command line argument.

`--help'

@opindex --help Print a usage message listing all available options, then exit successfully.

`--version'

@opindex --version Print the version number, then exit successfully.

Go to the previous, next section.

Go to the previous, next section.

TOC

Output of entire files

These commands read and write entire files, possibly transforming them in some way.

`cat`: Concatenate and write files

`cat` copies each file (`-' means standard input), or standard input if none are given, to standard output. Synopsis:

```
cat [option] [file]...
```

The program accepts the following options. Also see section Common options.

`-A'

`--show-all'

@opindex -A @opindex --show-all Equivalent to `-vET'.

`-b'

`--number-nonblank'

@opindex -b @opindex --number-nonblank Number all nonblank output lines, starting with 1.

`-e'

@opindex -e Equivalent to `-vE'.

`-E'

`--show-ends'

@opindex -E @opindex --show-ends Display a `$' after the end of each line.

`-n'

`--number'

@opindex -n @opindex --number Number all output lines, starting with 1.

`-s'

`--squeeze-blank'

@opindex -s @opindex --squeeze-blank Replace multiple adjacent blank lines with a single blank line.

`-t'

@opindex -t Equivalent to `-vT'.

`-T'

`--show-tabs'

@opindex -T @opindex --show-tabs Display TAB characters as `^I'.

`-u'

@opindex -u Ignored; for Unix compatibility.

`-v'

`--show-nonprinting'

@opindex -v @opindex --show-nonprinting Display control characters except for LFD and TAB using `^' notation and precede characters that have the high bit set with `M-'.

`tac`: Concatenate and write files in reverse

`tac` copies each file (`-' means standard input), or standard input if none are given, to standard output, reversing the records (lines by default) in each separately. Synopsis:

```
tac [option]... [file]...
```

Records are separated by instances of a string (newline by default). By default, this separator string is attached to the end of the record that it follows in the file.

The program accepts the following options. Also see section Common options.

`-b'

`--before'

@opindex -b @opindex --before The separator is attached to the beginning of the record that it precedes in the file.

`-r'

`--regex'

@opindex -r @opindex --regex Treat the separator string as a regular expression.

`-s separator'

`--separator=separator'

@opindex -s @opindex --separator Use separator as the record separator, instead of newline.

`nl`: Number lines and write files

`nl` writes each file (`-' means standard input), or standard input if none are given, to standard output, with line numbers added to some or all of the lines. Synopsis:

```
nl [option]... [file]...
```

`nl` decomposes its input into (logical) pages; by default, the line number is reset to 1 at the top of each logical page. `nl` treats all of the input files as a single document; it does not reset line numbers or logical pages between files.

A logical page consists of three sections: header, body, and footer. Any of the sections can be empty. Each can be numbered in a different style from the others.

The beginnings of the sections of logical pages are indicated in the input file by a line containing exa one of these delimiter strings:

`\:\:\:'

start of header;

`\:\:'

start of body;

`\:'

start of footer.

The two characters from which these strings are made can be changed from `\' and `:' via options (see below), but the pattern and length of each string cannot be changed.

A section delimiter is replaced by an empty line on output. Any text that comes before the first section delimiter string in the input file is considered to be part of a body section, so `nl` treats a file that contains no section delimiters as a single body section.

The program accepts the following options. Also see section Common options.

`-b style'

`--body-numbering=style'

@opindex -b @opindex --body-numbering Select the numbering style for lines in the body section of each logical page. When a line is not numbered, the current line number is not incremented, but the line number separator character is still prepended to the line. The styles are:

`a'

number all lines,

`t'

number only nonempty lines (default for body),

`n'

do not number lines (default for header and footer),

`pregexp'

number only lines that contain a match for regexp.

- -d cd
- --section-delimiter=cd @opindex -d @opindex --section-delimiter Set the section delimiter characters to cd; default is `\:'. If only c is given, the second remains `:'. (Remember to protect `\' or other metacharacters from shell expansion with quotes or extra backslashes.)
- -f style
- --footer-numbering=style @opindex -f @opindex --footer-numbering Analogous to `--body-numbering'.
- -h style
- --header-numbering=style @opindex -h @opindex --header-numbering Analogous to `--body-numbering'.

- -i number
- --page-increment=number @opindex -i @opindex --page-increment Increment line numbers by number (default 1).
- -l number
- --join-blank-lines=number @opindex -l @opindex --join-blank-lines Consider number (default 1) consecutive empty lines to be one logical line for numbering, and only number the last one. Where fewer than number consecutive empty lines occur, do not number them. An empty line is one that contains no characters, not even spaces or tabs.
- -n format
- --number-format=format @opindex -n @opindex --number-format Select the line numbering format (default is `rn`):

`ln'

@opindex ln format for `nl` left justified, no leading zeros;

`rn'

@opindex rn format for `nl` right justified, no leading zeros;

`rz'

@opindex rz format for `nl` right justified, leading zeros.

- -p
- --no-renumber @opindex -p @opindex --no-renumber Do not reset the line number at the start of a logical page.
- -s string
- --number-separator=string @opindex -s @opindex --number-separator Separate the line number from the text line in the output with string (default is TAB).
- -v number
- --starting-line-number=number @opindex -v @opindex --starting-line-number Set the initial line number on each logical page to number (default 1).
- -w number
- --number-width=number @opindex -w @opindex --number-width Use number characters for line numbers (default 6).

od: Write files in octal or other formats

`od` writes an unambiguous representation of each file (`-' means standard input), or standard input if none are given. Synopsis:

```
od [option]... [file]...
od -C [file] [[+]offset [[+]label]]
```

Each line of output consists of the offset in the input, followed by groups of data from the file. By default, `od` prints the offset in octal, and each group of file data is two bytes of input printed as a single

octal number.

The program accepts the following options. Also see section Common options.

`-A radix'

`--address-radix=radix'

@opindex -A @opindex --address-radix Select the base in which file offsets are printed. radix can be one of the following:

`d'

decimal;

`o'

octal;

`x'

hexadecimal;

`n'

none (do not print offsets).

The default is octal.

- -j bytes
- --skip-bytes=bytes @opindex -j @opindex --skip-bytes Skip bytes input bytes before formatting and writing. If bytes begins with `0x' or `0X', it is interpreted in hexadecimal; otherwise, if it begins with `0', in octal; otherwise, in decimal. Appending `b' multiplies bytes by 512, `k' by 1024, and `m' by 1048576.
- -N bytes
- --read-bytes=bytes @opindex -N @opindex --read-bytes Output at most bytes bytes of the input. Prefixes and suffixes on `bytes` are interpreted as for the `-j' option.
- -s [n]
- --strings[=n] @opindex -s @opindex --strings Instead of the normal output, output only string constants: at least n (3 by default) consecutive ASCII graphic characters, followed by a null (zero) byte.
- -t type
- --format=type @opindex -t @opindex --format Select the format in which to output the file data. type is a string of one or more of the below type indicator characters. If you include more than one type indicator character in a single type string, or use this option more than once, `od` writes one copy of each output line using each of the data types that you specified, in the order that you specified.

`a'

named character,

`c'

ASCII character or backslash escape,

`d'

signed decimal,

`f'

floating point,

`o'

octal,

`u'

unsigned decimal,

`x'

hexadecimal.

The type `a` outputs things like `sp' for space, `nl' for newline, and `nul' for a null (zero) byte. Type `c` outputs ` ', `\n', and `\0`, respectively.

Except for types `a' and `c', you can specify the number of bytes to use in interpreting each number in the given data type by following the type indicator character with a decimal integer. Alternately, you can specify the size of one of the C compiler's built-in data types by following the type indicator character with one of the following characters. For integers (`d', `o', `u', `x'):

`C'

char,

`S'

short,

`I'

int,

`L'

long.

For floating point (`f`):

F

float,

D

double,

L

long double.

- -v
- --output-duplicates @opindex -v @opindex --output-duplicates Output consecutive lines that are identical. By default, when two or more consecutive output lines would be identical, `od` outputs only the first line, and puts just an asterisk on the following line to indicate the elision.
- -w[n]
- --width[=n] @opindex -w @opindex --width Dump `n` input bytes per output line. This must be a multiple of the least common multiple of the sizes associated with the specified output types. If n is omitted, the default is 32. If this option is not given at all, the default is 16.

The next several options map the old, pre-POSIX format specification options to the corresponding

POSIX format specs. GNU `od` accepts any combination of old- and new-style options. Format specification options accumulate.

`-a'

@opindex -a Output as named characters. Equivalent to `-ta'.

`-b'

@opindex -b Output as octal bytes. Equivalent to `-toC'.

`-c'

@opindex -c Output as ASCII characters or backslash escapes. Equivalent to `-tc'.

`-d'

@opindex -d Output as unsigned decimal shorts. Equivalent to `-tu2'.

`-f'

@opindex -f Output as floats. Equivalent to `-tfF'.

`-h'

@opindex -h Output as hexadecimal shorts. Equivalent to `-tx2'.

`-i'

@opindex -i Output as decimal shorts. Equivalent to `-td2'.

`-l'

@opindex -l Output as decimal longs. Equivalent to `-td4'.

`-o'

@opindex -o Output as octal shorts. Equivalent to `-to2'.

`-x'

@opindex -x Output as hexadecimal shorts. Equivalent to `-tx2'.

`-C'

`--traditional'

@opindex --traditional Recognize the pre-POSIX non-option arguments that traditional `od` accepted. The following syntax:

```
od --traditional [file] [[+]offset[.][b] [[+]label[.][b]]]
```

can be used to specify at most one file and optional arguments specifying an offset and a pseudo-start address, label. By default, offset is interpreted as an octal number specifying how many input bytes to skip before formatting and writing. The optional trailing decimal point forces the interpretation of offset as a decimal number. If no decimal is specified and the offset begins with `0x' or `0X' it is interpreted as a hexadecimal number. If there is a trailing `b', the number of bytes skipped will be offset multiplied by 512. The label argument is interpreted just like offset, but it specifies an initial pseudo-address. The pseudo-addresses are displayed in parentheses following any normal address.

Go to the previous, next section.

Go to the previous, next section.

Formatting file contents

These commands reformat the contents of files.

`fmt`: Reformat paragraph text

`fmt` fills and joins lines to produce output lines of (at most) a given number of characters (75 by default). Synopsis:

```
fmt [option]... [file]...
```

`fmt` reads from the specified file arguments (or standard input if none are given), and writes to standard output.

By default, blank lines, spaces between words, and indentation are preserved in the output; successive input lines with different indentation are not joined; tabs are expanded on input and introduced on output.

`fmt` prefers breaking lines at the end of a sentence, and tries to avoid line breaks after the first word of a sentence or before the last word of a sentence. A sentence break is defined as either the end of a paragraph or a word ending in any of `.?!', followed by two spaces or end of line, ignoring any intervening parentheses or quotes. Like TeX, `fmt` reads entire "paragraphs" before choosing line breaks; the algorithm is a variant of that in "Breaking Paragraphs Into Lines" (Donald E. Knuth and Michael F. Plass, Software--Practice and Experience, 11 (1981), 1119--1184).

The program accepts the following options. Also see section Common options.

`-c'

`--crown-margin'

@opindex -c @opindex --crown-margin Crown margin mode: preserve the indentation of the first two lines within a paragraph, and align the left margin of each subsequent line with that of the second line.

`-t'

`--tagged-paragraph'

@opindex -t @opindex --tagged-paragraph Tagged paragraph mode: like crown margin mode, except that if indentation of the first line of a paragraph is the same as the indentation of the second, the first line is treated as a one-line paragraph.

`-s'

`--split-only'

@opindex -s @opindex --split-only Split lines only. Do not join short lines to form longer ones. This prevents sample lines of code, and other such "formatted" text from being unduly combined.

`-u'

`--uniform-spacing'

@opindex -u @opindex --uniform-spacing Uniform spacing. Reduce spacing between words to one space, and spacing between sentences to two spaces.

`-width'

`-w width'

`--width=width'

@opindex -width @opindex -w @opindex --width Fill output lines up to width characters (default 75). `fmt` initially tries to make lines about 7% shorter than this, to give it room to balance line lengths.

`-p prefix'

`--prefix=prefix'

Only lines beginning with prefix (possibly preceded by whitespace) are subject to formatting. The prefix and any preceding whitespace are stripped for the formatting and then re-attached to each formatted output line. One use is to format certain kinds of program comments, while leaving the code unchanged.

`pr`: Paginate or columnate files for printing

`pr` writes each file (`-' means standard input), or standard input if none are given, to standard output, paginating and optionally outputting in multicolumn format. Synopsis:

```
pr [option]... [file]...
```

By default, a 5-line header is printed: two blank lines; a line with the date, the file name, and the page count; and two more blank lines. A five line footer (entirely) is also printed.

Form feeds in the input cause page breaks in the output.

The program accepts the following options. Also see section Common options.

`+page'

Begin printing with page page.

`-column'

@opindex -column Produce column-column output and print columns down. The column width is automatically decreased as column increases; unless you use the `-w' option to increase the page width as well, this option might well cause some input to be truncated.

`-a'

@opindex -a Print columns across rather than down.

`-b'

@opindex -b Balance columns on the last page.

`-c'

@opindex -c Print control characters using hat notation (e.g., `^G'); print other unprintable

characters in octal backslash notation. By default, unprintable characters are not changed.

`-d'

@opindex -d Double space the output.

`-e[in-tabchar[in-tabwidth]]'

@opindex -e Expand tabs to spaces on input. Optional argument in-tabchar is the input tab character (default is TAB). Second optional argument in-tabwidth is the input tab character's width (default is 8).

`-f'

`-F'

@opindex -F @opindex -f Use a formfeed instead of newlines to separate output pages.

`-h header'

@opindex -h Replace the file name in the header with the string header.

`-i[out-tabchar[out-tabwidth]]'

@opindex -i Replace spaces with tabs on output. Optional argument out-tabchar is the output tab character (default is TAB). Second optional argument out-tabwidth is the output tab character's width (default is 8).

`-l n'

@opindex -l Set the page length to n (default 66) lines. If n is less than 10, the headers and footers are omitted, as if the `-t' option had been given.

`-m'

@opindex -m Print all files in parallel, one in each column.

`-n[number-separator[digits]]'

@opindex -n Precede each column with a line number; with parallel files (`-m'), precede each line with a line number. Optional argument number-separator is the character to print after each number (default is TAB). Optional argument digits is the number of digits per line number (default is 5).

`-o n'

@opindex -o Indent each line with n (default is zero) spaces wide, i.e., set the left margin. The total page width is `n' plus the width set with the `-w' option.

`-r'

@opindex -r Do not print a warning message when an argument file cannot be opened. (The exit status will still be nonzero, however.)

`-s[c]'

@opindex -s Separate columns by the single character c. If c is omitted, the default is space; if this option is omitted altogether, the default is TAB.

`-t'

@opindex -t Do not print the usual 5-line header and the 5-line footer on each page, and do not fill out the bottoms of pages (with blank lines or formfeeds).

`-v'

@opindex -v Print unprintable characters in octal backslash notation.

`-w n'

@opindex -w Set the page width to n (default is 72) columns.

`fold`: Wrap input lines to fit in specified width

`fold` writes each file (`-' means standard input), or standard input if none are given, to standard output, breaking long lines. Synopsis:

```
fold [option]... [file]...
```

By default, `fold` breaks lines wider than 80 columns. The output is split into as many lines as necessary.

`fold` counts screen columns by default; thus, a tab may count more than one column, backspace decreases the column count, and carriage return sets the column to zero.

The program accepts the following options. Also see section Common options.

`-b'

`--bytes'

@opindex -b @opindex --bytes Count bytes rather than columns, so that tabs, backspaces, and carriage returns are each counted as taking up one column, just like other characters.

`-s'

`--spaces'

@opindex -s @opindex --spaces Break at word boundaries: the line is broken after the last blank before the maximum line length. If the line contains no such blanks, the line is broken at the maximum line length as usual.

`-w width'

`--width=width'

@opindex -w @opindex --width Use a maximum line length of width columns instead of 80.

Go to the previous, next section.

Go to the previous, next section.

Output of parts of files

These commands output pieces of the input.

head: Output the first part of files

`head` prints the first part (10 lines by default) of each file; it reads from standard input if no files are given or when given a file of `-'. Synopses:

```
head [option]... [file]...
head -number [option]... [file]...
```

If more than one file is specified, `head` prints a one-line header consisting of

```
==> file name <==
```

before the output for each file.

`head` accepts two option formats: the new one, in which numbers are arguments to the options (`-q -n 1'), and the old one, in which the number precedes any option letters (`-1q').

The program accepts the following options. Also see section Common options.

`-countoptions'

@opindex -count This option is only recognized if it is specified first. count is a decimal number optionally followed by a size letter (`b', `k', `m') as in `-c`, or `l' to mean count by lines, or other option letters (`cqv').

`-c bytes'

`--bytes=bytes'

@opindex -c @opindex --bytes Print the first bytes bytes, instead of initial lines. Appending `b' multiplies bytes by 512, `k' by 1024, and `m' by 1048576.

`-n n'

`--lines=n'

@opindex -n @opindex --lines Output the first n lines.

`-q'

`--quiet'

`--silent'

@opindex -q @opindex --quiet @opindex --silent Never print file name headers.

`-v'

`--verbose'

@opindex -v @opindex --verbose Always print file name headers.

`tail`: Output the last part of files

`tail` prints the last part (10 lines by default) of each file; it reads from standard input if no files are given or when given a file of `-'. Synopses:

```
tail [option]... [file]...
tail -number [option]... [file]...
tail +number [option]... [file]...
```

If more than one file is specified, `tail` prints a one-line header consisting of

```
==> file name <==
```

before the output for each file.

GNU `tail` can output any amount of data (some other versions of `tail` cannot). It also has no `-r' option (print in reverse), since reversing a file is really a different job from printing the end of a file; BSD `tail` (which is the one with `-r`) can only reverse files that are at most as large as its buffer, which is typically 32k. A more reliable and versatile way to reverse files is the GNU `tac` command.

`tail` accepts two option formats: the new one, in which numbers are arguments to the options (`-n 1'), and the old one, in which the number precedes any option letters (`-1' or `+1').

If any option-argument is a number n starting with a `+', `tail` begins printing with the nth item from the start of each file, instead of from the end.

The program accepts the following options. Also see section Common options.

`-count'

`+count'

@opindex -count @opindex +count This option is only recognized if it is specified first. count is a decimal number optionally followed by a size letter (`b', `k', `m') as in `-c`, or `l' to mean count by lines, or other option letters (`cfqv').

`-c bytes'

`--bytes=bytes'

@opindex -c @opindex --bytes Output the last bytes bytes, instead of final lines. Appending `b' multiplies bytes by 512, `k' by 1024, and `m' by 1048576.

`-f'

`--follow'

@opindex -f @opindex --follow Loop forever trying to read more characters at the end of the file, presumably because the file is growing. Ignored if reading from a pipe. If more than one file is given, `tail` prints a header whenever it gets output from a different file, to indicate which file that output is from.

`-n n'

`--lines=n'

@opindex -n @opindex --lines Output the last n lines.

`-q'

`-quiet'

`--silent'

@opindex -q @opindex --quiet @opindex --silent Never print file name headers.

`-v'

`--verbose'

@opindex -v @opindex --verbose Always print file name headers.

`split`: Split a file into fixed-size pieces

`split` creates output files containing consecutive sections of input (standard input if none is given or input is `-'). Synopsis:

```
split [option] [input [prefix]]
```

By default, `split` puts 1000 lines of input (or whatever is left over for the last section), into each output file.

The output files' names consist of prefix (`x' by default) followed by a group of letters `aa', `ab', and so on, such that concatenating the output files in sorted order by file name produces the original input file. (If more than 676 output files are required, `split` uses `zaa', `zab', etc.)

The program accepts the following options. Also see section Common options.

`-lines'

`-l lines'

`--lines=lines'

@opindex -l @opindex --lines Put lines lines of input into each output file.

`-b bytes'

`--bytes=bytes'

@opindex -b @opindex --bytes Put the first bytes bytes of input into each output file. Appending `b' multiplies bytes by 512, `k' by 1024, and `m' by 1048576.

`-C bytes'

`--line-bytes=bytes'

@opindex -C @opindex --line-bytes Put into each output file as many complete lines of input as possible without exceeding bytes bytes. For lines longer than bytes bytes, put bytes bytes into each output file until less than bytes bytes of the line are left, then continue normally. bytes has the same format as for the `--bytes' option.

`--verbose=bytes'

@opindex --verbose Write a diagnostic to standard error just before each output file is opened.

`csplit`: Split a file into context-determined pieces

`csplit` creates zero or more output files containing sections of input (standard input if input is `-'). Synopsis:

```
csplit [option]... input pattern...
```

The contents of the output files are determined by the pattern arguments, as detailed below. An error occurs if a pattern argument refers to a nonexistent line of the input file (e.g., if no remaining line matches a given regular expression). After every pattern has been matched, any remaining input is copied into one last output file.

By default, `csplit` prints the number of bytes written to each output file after it has been created.

The types of pattern arguments are:

`n'

Create an output file containing the input up to but not including line n (a positive integer). If followed by a repeat count, also create an output file containing the next line lines of the input file once for each repeat.

`/regexp/[offset]'

Create an output file containing the current line up to (but not including) the next line of the input file that contains a match for regexp. The optional offset is a `+' or `-' followed by a positive integer. If it is given, the input up to the matching line plus or minus offset is put into the output file, and the line after that begins the next section of input.

`%regexp%[offset]'

Like the previous type, except that it does not create an output file, so that section of the input file is effectively ignored.

`{repeat-count}'

Repeat the previous pattern repeat-count additional times. repeat-count can either be a positive integer or an asterisk, meaning repeat as many times as necessary until the input is exhausted.

The output files' names consist of a prefix (`xx' by default) followed by a suffix. By default, the suffix is an ascending sequence of two-digit decimal numbers from `00' and up to `99'. In any case, concatenating the output files in sorted order by filename produces the original input file.

By default, if `csplit` encounters an error or receives a hangup, interrupt, quit, or terminate signal, it removes any output files that it has created so far before it exits.

The program accepts the following options. Also see section Common options.

`-f prefix'

`--prefix=prefix'

@opindex -f @opindex --prefix Use prefix as the output file name prefix.

`-b suffix'

`--suffix=suffix'

@opindex -b @opindex --suffix Use suffix as the output file name suffix. When this option is specified, the suffix string must include exactly one `printf(3)`-style conversion specification, possibly including format specification flags, a field width, a precision specifications, or all of these kinds of modifiers. The format letter must convert a binary integer argument to readable form; thus, only `d', `i', `u', `o', `x', and `X' conversions are allowed. The entire suffix is given (with the current output file number) to `sprintf(3)` to form the file name suffixes for each of the individual output files in turn. If this option is used, the `--digits' option is ignored.

`-n digits'

`--digits=digits'

@opindex -n @opindex --digits Use output file names containing numbers that are digits digits long instead of the default 2.

`-k'

`--keep-files'

@opindex -k @opindex --keep-files Do not remove output files when errors are encountered.

`-z'

`--elide-empty-files'

@opindex -z @opindex --elide-empty-files Suppress the generation of zero-length output files. (In cases where the section delimiters of the input file are supposed to mark the first lines of each of the sections, the first output file will generally be a zero-length file unless you use this option.) The output file sequence numbers always run consecutively starting from 0, even when this option is specified.

`-s'

`-q'

`--silent'

`--quiet'

@opindex -s @opindex -q @opindex --silent @opindex --quiet Do not print counts of output file sizes.

Go to the previous, next section.

Go to the previous, next section.

Summarizing files

These commands generate just a few numbers representing entire contents of files.

wc: Print byte, word, and line counts

`wc` counts the number of bytes, whitespace-separated words, and newlines in each given file, or standard input if none are given or for a file of `-'. Synopsis:

```
wc [option]... [file]...
```

`wc` prints one line of counts for each file, and if the file was given as an argument, it prints the file name following the counts. If more than one file is given, `wc` prints a final line containing the cumulative counts, with the file name `total'. The counts are printed in this order: newlines, words, bytes.

By default, `wc` prints all three counts. Options can specify that only certain counts be printed. Options do not undo others previously given, so

```
wc --bytes --words
```

prints both the byte counts and the word counts.

The program accepts the following options. Also see section Common options.

`-c'

`--bytes'

`--chars'

@opindex -c @opindex --bytes @opindex --chars Print only the byte counts.

`-w'

`--words'

@opindex -w @opindex --words Print only the word counts.

`-l'

`--lines'

@opindex -l @opindex --lines Print only the newline counts.

sum: Print checksum and block counts

`sum` computes a 16-bit checksum for each given file, or standard input if none are given or for a file of `-'. Synopsis:


```
sum [option]... [file]...
```

`sum` prints the checksum for each file followed by the number of blocks in the file (rounded up). If more than one file is given, file names are also printed (by default). (With the `--sysv' option, corresponding file name are printed when there is at least one file argument.)

By default, GNU `sum` computes checksums using an algorithm compatible with BSD `sum` and prints file sizes in units of 1024-byte blocks.

The program accepts the following options. Also see section Common options.

`-r'

@opindex -r Use the default (BSD compatible) algorithm. This option is included for compatibility with the System V `sum`. Unless `-s' was also given, it has no effect.

`-s'

`--sysv'

@opindex -s @opindex --sysv Compute checksums using an algorithm compatible with System V `sum`'s default, and print file sizes in units of 512-byte blocks.

`sum` is provided for compatibility; the `cksum` program (see next section) is preferable in new applications.

`cksum`: Print CRC checksum and byte counts

`cksum` computes a cyclic redundancy check (CRC) checksum for each given file, or standard input if none are given or for a file of `-'. Synopsis:

```
cksum [option]... [file]...
```

`cksum` prints the CRC checksum for each file along with the number of bytes in the file, and the filename unless no arguments were given.

`cksum` is typically used to ensure that files transferred by unreliable means (e.g., netnews) have not been corrupted, by comparing the `cksum` output for the received files with the `cksum` output for the original files (typically given in the distribution).

The CRC algorithm is specified by the POSIX.2 standard. It is not compatible with the BSD or System V `sum` algorithms (see the previous section); it is more robust.

The only options are `--help' and `--version'. See section Common options.

`md5sum`: Print or check message-digests

`md5sum` computes a 128-bit checksum (or fingerprint or message-digest) for each specified file. If a file is specified as `-' or if no files are given `md5sum` computes the checksum for the standard input. `md5sum` can also determine whether a file and checksum are consistent. Synopsis:


```
md5sum [option]... [file]...
md5sum [option]... --check [file]
md5sum [option]... --string=string ...
```

For each file, `md5sum' outputs the MD5 checksum, a flag indicating a binary or text input file, and the filename. If file is omitted or specified as `-', standard input is read.

The program accepts the following options. Also see section Common options.

`-b'

`--binary'

@opindex -b @opindex --binary Treat all input files as binary. This option has no effect on Unix systems, since they don't distinguish between binary and text files. This option is useful on systems that have different internal and external character representations.

`-c'

`--check'

Read filenames and checksum information from the single file (or from stdin if no file was specified) and report whether each named file and the corresponding checksum data are consistent. The input to this mode of `md5sum` is usually the output of a prior, checksum-generating run of `md5sum'. Each valid line of input consists of an MD5 checksum, a binary/text flag, and then a filename. Binary files are marked with `*', text with ` '. For each such line, `md5sum` reads the named file and computes its MD5 checksum. Then, if the computed message digest does not match the one on the line with the filename, the file is noted as having failed the test. Otherwise, the file passes the test. By default, for each valid line, one line is written to standard output indicating whether the named file passed the test. After all checks have been performed, if there were any failures, a warning is issued to standard error. Use the `--status' option to inhibit that output. If any listed file cannot be opened or read, if any valid line has an MD5 checksum inconsistent with the associated file, or if no valid line is found, `md5sum` exits with nonzero status. Otherwise, it exits successfully.

`--status'

@opindex --status This option is useful only when verifying checksums. When verifying checksums, don't generate the default one-line-per-file diagnostic and don't output the warning summarizing any failures. Failures to open or read a file still evoke individual diagnostics to standard error. If all listed files are readable and are consistent with the associated MD5 checksums, exit successfully. Otherwise exit with a status code indicating there was a failure.

`--string=string'

@opindex --string Compute the message digest for string, instead of for a file. The result is the same as for a file that contains exactly string.

`-t'

`--text'

@opindex -t @opindex --text Treat all input files as text files. This is the reverse of `--binary'.

`-w'

`--warn'

@opindex -w @opindex --warn When verifying checksums, warn about improperly formated MD5 checksum lines. This option is useful only if all but a few lines in the checked input are valid.

Go to the previous, next section.

Go to the previous, next section.

Operating on sorted files

These commands work with (or produce) sorted files.

`sort`: Sort text files

`sort` sorts, merges, or compares all the lines from the given files, or standard input if none are given or for a file of `-'. By default, `sort` writes the results to standard output. Synopsis:

```
sort [option]... [file]...
```

`sort` has three modes of operation: sort (the default), merge, and check for sortedness. The following options change the operation mode:

`-c'

@opindex -c Check whether the given files are already sorted: if they are not all sorted, print an error message and exit with a status of 1. Otherwise, exit successfully.

`-m'

@opindex -m Merge the given files by sorting them as a group. Each input file must always be individually sorted. It always works to sort instead of merge; merging is provided because it is faster, in the case where it works.

A pair of lines is compared as follows: if any key fields have been specified, `sort` compares each pair of fields, in the order specified on the command line, according to the associated ordering options, until a difference is found or no fields are left.

If any of the global options `Mbdfinr' are given but no key fields are specified, `sort` compares the entire lines according to the global options.

Finally, as a last resort when all keys compare equal (or if no ordering options were specified at all), `sort` compares the lines byte by byte in machine collating sequence. The last resort comparison honors the `-r' global option. The `-s' (stable) option disables this last-resort comparison so that lines in which all fields compare equal are left in their original relative order. If no fields or global options are specified, `-s' has no effect.

GNU `sort` (as specified for all GNU utilities) has no limits on input line length or restrictions on bytes allowed within lines. In addition, if the final byte of an input file is not a newline, GNU `sort` silently supplies one.

Upon any error, `sort` exits with a status of `2'.

If the environment variable `TMPDIR` is set, `sort` uses its value as the directory for temporary files instead of `/tmp'. The `-T tempdir' option in turn overrides the environment variable.

The following options affect the ordering of output lines. They may be specified globally or as part of a

specific key field. If no key fields are specified, global options apply to comparison of entire lines; otherwise the global options are inherited by key fields that do not specify any special options of their c

`-b'

@opindex -b Ignore leading blanks when finding sort keys in each line.

`-d'

@opindex -d Sort in phone directory order: ignore all characters except letters, digits and blanks when sorting.

`-f'

@opindex -f Fold lowercase characters into the equivalent uppercase characters when sorting so that, for example, `b' and `B' sort as equal.

`-g'

@opindex -g Sort numerically, but use strtod(3) to arrive at the numeric values. This allows floating point numbers to be specified in scientific notation, like `1.0e-34` and `10e100`. Use this option only if there is no alternative; it is much slower than `-n' and numbers with too many significant digits will be compared as if they had been truncated. In addition, numbers outside the range of representable double precision floating point numbers are treated as if they were zeroes; overflow and underflow are not reported.

`-i'

@opindex -i Ignore characters outside the printable ASCII range 040-0176 octal (inclusive) when sorting.

`-M'

@opindex -M An initial string, consisting of any amount of whitespace, followed by three letters abbreviating a month name, is folded to UPPER case and compared in the order `JAN' < `FEB' < ... < `DEC'. Invalid names compare low to valid names.

`-n'

@opindex -n Sort numerically: the number begins each line; specifically, it consists of optional whitespace, an optional `-' sign, and zero or more digits, optionally followed by a decimal point and zero or more digits.

`sort -n` uses what might be considered an unconventional method to compare strings representing floating point numbers. Rather than first converting each string to the C `double` type and then comparing those values, sort aligns the decimal points in the two strings and compares the strings a character at a time. One benefit of using this approach is its speed. In practice this is much more efficient than performing the two corresponding string-to-double (or even string-to-integer) conversions and then comparing doubles. In addition, there is no corresponding loss of precision. Converting each string to `double` before comparison would limit precision to about 16 digits on most systems.

Neither a leading `+' nor exponential notation is recognized. To compare such strings numerically, use the `-g' option.

`-r'

@opindex -r Reverse the result of comparison, so that lines with greater key values appear earlier in the output instead of later.

Other options are:

`-o output-file'

@opindex -o Write output to output-file instead of standard output. If output-file is one of the input files, `sort` copies it to a temporary file before sorting and writing the output to output-file.

`-t separator'

@opindex -t Use character separator as the field separator when finding the sort keys in each line. By default, fields are separated by the empty string between a non-whitespace character and a whitespace character. That is, given the input line ` foo bar', `sort` breaks it into fields ` foo' and ` bar'. The field separator is not considered to be part of either the field preceding or the field following.

`-u'

@opindex -u For the default case or the `-m' option, only output the first of a sequence of lines that compare equal. For the `-c' option, check that no pair of consecutive lines compares equal.

`-k pos1[,pos2]'

@opindex -k The recommended, POSIX, option for specifying a sort field. The field consists of the line between pos1 and pos2 (or the end of the line, if pos2 is omitted), inclusive. Fields and character positions are numbered starting with 1. See below.

`-z'

@opindex -z Treat the input as a set of lines, each terminated by a zero byte (ASCII NUL (Null) character) instead of a ASCII LF (Line Feed.) This option can be useful in conjunction with `perl -0' or `find -print0' and `xargs -0' which do the same in order to reliably handle arbitrary pathnames (even those which contain Line Feed characters.)

`+pos1[-pos2]'

The obsolete, traditional option for specifying a sort field. The field consists of the line between pos1 and up to but *not including* pos2 (or the end of the line if pos2 is omitted). Fields and character positions are numbered starting with 0. See below.

In addition, when GNU `sort` is invoked with exactly one argument, options `--help' and `--version' are recognized. See section Common options.

Historical (BSD and System V) implementations of `sort` have differed in their interpretation of some options, particularly `-b', `-f', and `-n'. GNU sort follows the POSIX behavior, which is usually (but not always!) like the System V behavior. According to POSIX, `-n' no longer implies `-b'. For consistency, `-M' has been changed in the same way. This may affect the meaning of character positions in field specifications in obscure cases. The only fix is to add an explicit `-b'.

A position in a sort field specified with the `-k' or `+' option has the form `f.c', where f is the number of the field to use and c is the number of the first character from the beginning of the field (for `+pos') or from the end of the previous field (for `-pos'). If the `.c' is omitted, it is taken to be the first character in the field. If the `-b' option was specified, the `.c' part of a field specification is counted from the first nonblank character of the field (for `+pos') or from the first nonblank character following the previous field (for `-pos').

A sort key option may also have any of the option letters `Mbdfinr' appended to it, in which case the global ordering options are not used for that particular field. The `-b' option may be independently attached to either or both of the `+pos' and `-pos' parts of a field specification, and if it is inherited from the global options it will be attached to both. If a `-n' or `-M' option is used, thus implying a `-b' option, the `-b' option

is taken to apply to both the `+pos' and the `-pos' parts of a key specification. Keys may span multiple fields.

Here are some examples to illustrate various combinations of options. In them, the POSIX `-k' option is used to specify sort keys rather than the obsolete `+pos1-pos2' syntax.

- Sort in descending (reverse) numeric order.

  ```
  sort -nr
  ```

  Sort alphabetically, omitting the first and second fields. This uses a single key composed of the characters beginning at the start of field three and extending to the end of each line.

  ```
  sort -k3
  ```

- Sort numerically on the second field and resolve ties by sorting alphabetically on the third and fourth characters of field five. Use `:' as the field delimiter.

  ```
  sort -t : -k 2,2n -k 5.3,5.4
  ```

  Note that if you had written `-k 2' instead of `-k 2,2' `sort' would have used all characters beginning in the second field and extending to the end of the line as the primary *numeric* key. For the large majority of applications, treating keys spanning more than one field as numeric will not do what you expect.

  Also note that the `n' modifier was applied to the field-end specifier for the first key. It would have been equivalent to specify `-k 2n,2' or `-k 2n,2n'. All modifiers except `b' apply to the associated *field*, regardless of whether the modifier character is attached to the field-start and/or the field-end part of the key specifier.

- Sort the password file on the fifth field and ignore any leading white space. Sort lines with equal values in field five on the numeric user ID in field three.

  ```
  sort -t : -k 5b,5 -k 3,3n /etc/passwd
  ```

  An alternative is to use the global numeric modifier `-n'.

  ```
  sort -t : -n -k 5b,5 -k 3,3 /etc/passwd
  ```

- Generate a tags file in case insensitive sorted order.

  ```
  find src -type f -print0 | sort -t / -z -f | xargs -0 etags --append
  ```

  The use of `-print0', `-z', and `-0' in this case mean that pathnames that contain Line Feed characters will not get broken up by the sort operation.

  Finally, to ignore both leading and trailing white space, you could have applied the `b' modifier to the field-end specifier for the first key,

  ```
  sort -t : -n -k 5b,5b -k 3,3 /etc/passwd
  ```

or by using the global `-b' modifier instead of `-n' and an explicit `n' with the second key specifie

```
sort -t : -b -k 5,5 -k 3,3n /etc/passwd
```

uniq: Uniqify files

`uniq` writes the unique lines in the given `input', or standard input if nothing is given or for an input name of `-'. Synopsis:

```
uniq [option]... [input [output]]
```

By default, `uniq` prints the unique lines in a sorted file, i.e., discards all but one of identical successive lines. Optionally, it can instead show only lines that appear exactly once, or lines that appear more than once.

The input must be sorted. If your input is not sorted, perhaps you want to use `sort -u`.

If no output file is specified, `uniq` writes to standard output.

The program accepts the following options. Also see section Common options.

`-n'

`-f n'

`--skip-fields=n'

@opindex -n @opindex -f @opindex --skip-fields Skip n fields on each line before checking for uniqueness. Fields are sequences of non-space non-tab characters that are separated from each other by at least one spaces or tabs.

`+n'

`-s n'

`--skip-chars=n'

@opindex +n @opindex -s @opindex --skip-chars Skip n characters before checking for uniqueness. If you use both the field and character skipping options, fields are skipped over first.

`-c'

`--count'

@opindex -c @opindex --count Print the number of times each line occurred along with the line.

`-i'

`--ignore-case'

@opindex -i @opindex --ignore-case Ignore differences in case when comparing lines.

`-d'

`--repeated'

@opindex -d @opindex --repeated Print only duplicate lines.

`-u'

`--unique'

@opindex -u @opindex --unique Print only unique lines.

`-w n'

`--check-chars=n'

@opindex -w @opindex --check-chars Compare n characters on each line (after skipping any specified fields and characters). By default the entire rest of the lines are compared.

comm: Compare two sorted files line by line

`comm` writes to standard output lines that are common, and lines that are unique, to two input files; a file name of `-' means standard input. Synopsis:

```
comm [option]... file1 file2
```

The input files must be sorted before `comm` can be used.

With no options, `comm` produces three column output. Column one contains lines unique to file1, column two contains lines unique to file2, and column three contains lines common to both files. Columns are separated by TAB.

@opindex -1 @opindex -2 @opindex -3 The options `-1', `-2', and `-3' suppress printing of the corresponding columns. Also see section Common options.

Go to the previous, next section.

Go to the previous, next section.

Operating on fields within a line

`cut`: Print selected parts of lines

`cut` writes to standard output selected parts of each line of each input file, or standard input if no files are given or for a file name of `-'. Synopsis:

```
cut [option]... [file]...
```

In the table which follows, the byte-list, character-list, and field-list are one or more numbers or ranges (two numbers separated by a dash) separated by commas. Bytes, characters, and fields are numbered from starting at 1. Incomplete ranges may be given: `-m' means `1-m'; `n-' means `n' through end of line or last field.

The program accepts the following options. Also see section Common options.

`-b byte-list'

`--bytes=byte-list'

@opindex -b @opindex --bytes Print only the bytes in positions listed in byte-list. Tabs and backspaces are treated like any other character; they take up 1 byte.

`-c character-list'

`--characters=character-list'

@opindex -c @opindex --characters Print only characters in positions listed in character-list. The same as `-b' for now, but internationalization will change that. Tabs and backspaces are treated like any other character; they take up 1 character.

`-f field-list'

`--fields=field-list'

@opindex -f @opindex --fields Print only the fields listed in field-list. Fields are separated by a TAB by default.

`-d delim'

`--delimiter=delim'

@opindex -d @opindex --delimiter For `-f', fields are separated by the first character in delim (default is TAB).

`-n'

@opindex -n Do not split multi-byte characters (no-op for now).

`-s'

`--only-delimited'

@opindex -s @opindex --only-delimited For `-f', do not print lines that do not contain the field

separator character.

`paste`: Merge lines of files

`paste` writes to standard output lines consisting of sequentially corresponding lines of each given file, separated by TAB. Standard input is used for a file name of `-' or if no input files are given.

Synopsis:

```
paste [option]... [file]...
```

The program accepts the following options. Also see section Common options.

`-s'

`--serial'

@opindex -s @opindex --serial Paste the lines of one file at a time rather than one line from each file.

`-d delim-list'

`--delimiters delim-list'

@opindex -d @opindex --delimiters Consecutively use the characters in delim-list instead of TAB to separate merged lines. When delim-list is exhausted, start again at its beginning.

`join`: Join lines on a common field

`join` writes to standard output a line for each pair of input lines that have identical join fields. Synopsis:

```
join [option]... file1 file2
```

Either file1 or file2 (but not both) can be `-', meaning standard input. file1 and file2 should be already sorted in increasing order (not numerically) on the join fields; unless the `-t' option is given, they should be sorted ignoring blanks at the start of the join field, as in `sort -b`. If the `--ignore-case' option is given, lines should be sorted without regard to the case of characters in the join field, as in `sort -f`.

The defaults are: the join field is the first field in each line; fields in the input are separated by one or more blanks, with leading blanks on the line ignored; fields in the output are separated by a space; each output line consists of the join field, the remaining fields from file1, then the remaining fields from file2.

The program accepts the following options. Also see section Common options.

`-a file-number'

@opindex -a Print a line for each unpairable line in file file-number (either `1' or `2'), in addition to the normal output.

`-e string'

@opindex -e Replace those output fields that are missing in the input with string.

`-i'

`--ignore-case'

@opindex -i @opindex --ignore-case Ignore differences in case when comparing keys. With this option, the lines of the input files must be ordered in the same way. Use `sort -f' to produce this ordering.

`-1 field'

`-j1 field'

@opindex -1 @opindex -j1 Join on field field (a positive integer) of file 1.

`-2 field'

`-j2 field'

@opindex -2 @opindex -j2 Join on field field (a positive integer) of file 2.

`-j field'

Equivalent to `-1 field -2 field'.

`-o field-list...'

Construct each output line according to the format in field-list. Each element in field-list is either the single character `0' or has the form m.n where the file number, m, is `1' or `2' and n is a positive field number.

A field specification of `0' denotes the join field. In most cases, the functionality of the `0' field spec may be reproduced using the explicit m.n that corresponds to the join field. However, when printing unpairable lines (using either of the `-a' or `-v' options), there is no way to specify the join field using m.n in field-list if there are unpairable lines in both files. To give `join` that functionality, POSIX invented the `0' field specification notation.

The elements in field-list are separated by commas or blanks. Multiple field-list arguments can be given after a single `-o' option; the values of all lists given with `-o' are concatenated together. All output lines -- including those printed because of any -a or -v option -- are subject to the specified field-list.

`-t char'

Use character char as the input and output field separator.

`-v file-number'

Print a line for each unpairable line in file file-number (either `1' or `2'), instead of the normal output.

In addition, when GNU `join` is invoked with exactly one argument, options `--help' and `--version' are recognized. See section Common options.

Go to the previous, next section.

Go to the previous, next section.

Operating on characters

This commands operate on individual characters.

`tr`: Translate, squeeze, and/or delete characters

Synopsis:

```
tr [option]... set1 [set2]
```

`tr` copies standard input to standard output, performing one of the following operations:

- translate, and optionally squeeze repeated characters in the result,
- squeeze repeated characters,
- delete characters,
- delete characters, then squeeze repeated characters from the result.

The set1 and (if given) set2 arguments define ordered sets of characters, referred to below as set1 and set2. These sets are the characters of the input that `tr` operates on. The `--complement' (`-c') option replaces set1 with its complement (all of the characters that are not in set1).

Specifying sets of characters

The format of the set1 and set2 arguments resembles the format of regular expressions; however, they are not regular expressions, only lists of characters. Most characters simply represent themselves in these strings, but the strings can contain the shorthands listed below, for convenience. Some of them can be used only in set1 or set2, as noted below.

Backslash escapes.

A backslash followed by a character not listed below causes an error message.

`\a'
: Control-G,

`\b'
: Control-H,

`\f'
: Control-L,

`\n'
: Control-J,

`\r'

Control-M,

`\t'

Control-I,

`\v'

Control-K,

`\ooo'

The character with the value given by ooo, which is 1 to 3 octal digits,

`\\'

A backslash.

- Ranges.

The notation `m-n' expands to all of the characters from m through n, in ascending order. m should collate before n; if it doesn't, an error results. As an example, `0-9' is the same as `0123456789'. Although GNU `tr` does not support the System V syntax that uses square brackets to enclose ranges, translations specified in that format will still work as long as the brackets in string1 correspond to identical brackets in string2.

- Repeated characters.

The notation `[c*n]' in set2 expands to n copies of character c. Thus, `[y*6]' is the same as `yyyyyy'. The notation `[c*]' in string2 expands to as many copies of c as are needed to make set2 as long as set1. If n begins with `0', it is interpreted in octal, otherwise in decimal.

- Character classes.

The notation `[:class:]' expands to all of the characters in the (predefined) class class. The characters expand in no particular order, except for the `upper` and `lower` classes, which expand in ascending order. When the `--delete' (`-d') and `--squeeze-repeats' (`-s') options are both given, any character class can be used in set2. Otherwise, only the character classes `lower` and `upper` are accepted in set2, and then only if the corresponding character class (`upper` and `lower`, respectively) is specified in the same relative position in set1. Doing this specifies case conversion. The class names are given below; an error results when an invalid class name is given.

`alnum`

@opindex alnum Letters and digits.

`alpha`

@opindex alpha Letters.

`blank`

@opindex blank Horizontal whitespace.

`cntrl`

@opindex cntrl Control characters.

`digit`

@opindex digit Digits.

`graph`

@opindex graph Printable characters, not including space.

`lower`

@opindex lower Lowercase letters.

`print`

@opindex print Printable characters, including space.

`punct`

@opindex punct Punctuation characters.

`space`

@opindex space Horizontal or vertical whitespace.

`upper`

@opindex upper Uppercase letters.

`xdigit`

@opindex xdigit Hexadecimal digits.

- Equivalence classes.

The syntax `[=c=]' expands to all of the characters that are equivalent to c, in no particular order. Equivalence classes are a relatively recent invention intended to support non-English alphabets. But there seems to be no standard way to define them or determine their contents. Therefore, they are not fully implemented in GNU `tr`; each character's equivalence class consists only of that character, which is of no particular use.

Translating

`tr` performs translation when set1 and set2 are both given and the `--delete' (`-d') option is not given. `tr` translates each character of its input that is in set1 to the corresponding character in set2. Characters not in set1 are passed through unchanged. When a character appears more than once in set1 and the corresponding characters in set2 are not all the same, only the final one is used. For example, these two commands are equivalent:

```
tr aaa xyz
tr a z
```

A common use of `tr` is to convert lowercase characters to uppercase. This can be done in many ways. Here are three of them:

```
tr abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ
tr a-z A-Z
tr '[:lower:]' '[:upper:]'
```

When `tr` is performing translation, set1 and set2 typically have the same length. If set1 is shorter than set2, the extra characters at the end of set2 are ignored.

On the other hand, making set1 longer than set2 is not portable; POSIX.2 says that the result is

undefined. In this situation, BSD `tr` pads set2 to the length of set1 by repeating the last character of as many times as necessary. System V `tr` truncates set1 to the length of set2.

By default, GNU `tr` handles this case like BSD `tr`. When the `--truncate-set1' (`-t') option is given, GNU `tr` handles this case like the System V `tr` instead. This option is ignored for operations other than translation.

Acting like System V `tr` in this case breaks the relatively common BSD idiom:

```
tr -cs A-Za-z0-9 '\012'
```

because it converts only zero bytes (the first element in the complement of set1), rather than all non-alphanumerics, to newlines.

Squeezing repeats and deleting

When given just the `--delete' (`-d') option, `tr` removes any input characters that are in set1.

When given just the `--squeeze-repeats' (`-s') option, `tr` replaces each input sequence of a repeated character that is in set1 with a single occurrence of that character.

When given both `--delete' and `--squeeze-repeats', `tr` first performs any deletions using set1, then squeezes repeats from any remaining characters using set2.

The `--squeeze-repeats' option may also be used when translating, in which case `tr` first performs translation, then squeezes repeats from any remaining characters using set2.

Here are some examples to illustrate various combinations of options:

- Remove all zero bytes:

  ```
  tr -d '\000'
  ```

- Put all words on lines by themselves. This converts all non-alphanumeric characters to newlines, then squeezes each string of repeated newlines into a single newline:

  ```
  tr -cs '[a-zA-Z0-9]' '[\n*]'
  ```

- Convert each sequence of repeated newlines to a single newline:

  ```
  tr -s '\n'
  ```

Warning messages

Setting the environment variable `POSIXLY_CORRECT` turns off the following warning and error messages, for strict compliance with POSIX.2. Otherwise, the following diagnostics are issued:

1. When the `--delete' option is given but `--squeeze-repeats' is not, and set2 is given, GNU `tr` by default prints a usage message and exits, because set2 would not be used. The POSIX specification says that set2 must be ignored in this case. Silently ignoring arguments is a bad idea.

2. When an ambiguous octal escape is given. For example, `\400' is actually `\40' followed by the digit `0', because the value 400 octal does not fit into a single byte.

GNU `tr` does not provide complete BSD or System V compatibility. For example, it is impossible to disable interpretation of the POSIX constructs `[:alpha:]', `[=c=]', and `[c*10]'. Also, GNU `tr` does not delete zero bytes automatically, unlike traditional Unix versions, which provide no way to preserve zero bytes.

expand: Convert tabs to spaces

`expand` writes the contents of each given file, or standard input if none are given or for a file of `-', to standard output, with tab characters converted to the appropriate number of spaces. Synopsis:

```
expand [option]... [file]...
```

By default, `expand` converts all tabs to spaces. It preserves backspace characters in the output; they decrement the column count for tab calculations. The default action is equivalent to `-8' (set tabs every 8 columns).

The program accepts the following options. Also see section Common options.

`-tab1[,tab2]...'

`-t tab1[,tab2]...'

`--tabs=tab1[,tab2]...'

@opindex -tab @opindex -t @opindex --tabs If only one tab stop is given, set the tabs tab1 spaces apart (default is 8). Otherwise, set the tabs at columns tab1, tab2, ... (numbered from 0), and replace any tabs beyond the last tabstop given with single spaces. If the tabstops are specified with the `-t' or `--tabs' option, they can be separated by blanks as well as by commas.

`-i'

`--initial'

@opindex -i @opindex --initial Only convert initial tabs (those that precede all non-space or non-tab characters) on each line to spaces.

unexpand: Convert spaces to tabs

`unexpand` writes the contents of each given file, or standard input if none are given or for a file of `-', to standard output, with strings of two or more space or tab characters converted to as many tabs as possible followed by as many spaces as are needed. Synopsis:

```
unexpand [option]... [file]...
```

By default, `unexpand` converts only initial spaces and tabs (those that precede all non space or tab characters) on each line. It preserves backspace characters in the output; they decrement the column count for tab calculations. By default, tabs are set at every 8th column.

The program accepts the following options. Also see section Common options.

`-tab1[,tab2]...'

`-t tab1[,tab2]...'

`--tabs=tab1[,tab2]...'

@opindex -tab @opindex -t @opindex --tabs If only one tab stop is given, set the tabs tab1 spaces apart instead of the default 8. Otherwise, set the tabs at columns tab1, tab2, ... (numbered from 0), and leave spaces and tabs beyond the tabstops given unchanged. If the tabstops are specified with the `-t' or `--tabs' option, they can be separated by blanks as well as by commas. This option implies the `-a' option.

`-a'

`--all'

@opindex -a @opindex --all Convert all strings of two or more spaces or tabs, not just initial ones, to tabs.

Go to the previous, next section.

Go to the previous, next section.

Opening the software toolbox

This chapter originally appeared in Linux Journal, volume 1, number 2, in the What's GNU? column. It was written by Arnold Robbins.

Toolbox introduction

This month's column is only peripherally related to the GNU Project, in that it describes a number of the GNU tools on your Linux system and how they might be used. What it's really about is the "Software Tools" philosophy of program development and usage.

The software tools philosophy was an important and integral concept in the initial design and development of Unix (of which Linux and GNU are essentially clones). Unfortunately, in the modern day press of Internetworking and flashy GUIs, it seems to have fallen by the wayside. This is a shame, since it provides a powerful mental model for solving many kinds of problems.

Many people carry a Swiss Army knife around in their pants pockets (or purse). A Swiss Army knife is a handy tool to have: it has several knife blades, a screwdriver, tweezers, toothpick, nail file, corkscrew, and perhaps a number of other things on it. For the everyday, small miscellaneous jobs where you need a simple, general purpose tool, it's just the thing.

On the other hand, an experienced carpenter doesn't build a house using a Swiss Army knife. Instead, he has a toolbox chock full of specialized tools--a saw, a hammer, a screwdriver, a plane, and so on. And he knows exactly when and where to use each tool; you won't catch him hammering nails with the handle of his screwdriver.

The Unix developers at Bell Labs were all professional programmers and trained computer scientists. They had found that while a one-size-fits-all program might appeal to a user because there's only one program to use, in practice such programs are

1. difficult to write,
2. difficult to maintain and debug, and
3. difficult to extend to meet new situations.

Instead, they felt that programs should be specialized tools. In short, each program "should do one thing well." No more and no less. Such programs are simpler to design, write, and get right--they only do one thing.

Furthermore, they found that with the right machinery for hooking programs together, that the whole was greater than the sum of the parts. By combining several special purpose programs, you could accomplish a specific task that none of the programs was designed for, and accomplish it much more quickly and easily than if you had to write a special purpose program. We will see some (classic) examples of this further on in the column. (An important additional point was that, if necessary, take a detour and build any software tools you may need first, if you don't already have something appropriate in the toolbox.)

I/O redirection

Hopefully, you are familiar with the basics of I/O redirection in the shell, in particular the concepts of "standard input," "standard output," and "standard error". Briefly, "standard input" is a data source, where data comes from. A program should not need to either know or care if the data source is a disk file, a keyboard, a magnetic tape, or even a punched card reader. Similarly, "standard output" is a data sink, where data goes to. The program should neither know nor care where this might be. Programs that only read their standard input, do something to the data, and then send it on, are called "filters", by analogy to filters in a water pipeline.

With the Unix shell, it's very easy to set up data pipelines:

```
program_to_create_data | filter1 | .... | filterN > final.pretty.data
```

We start out by creating the raw data; each filter applies some successive transformation to the data, until by the time it comes out of the pipeline, it is in the desired form.

This is fine and good for standard input and standard output. Where does the standard error come in to play? Well, think about `filter1` in the pipeline above. What happens if it encounters an error in the data it sees? If it writes an error message to standard output, it will just disappear down the pipeline into `filter2`'s input, and the user will probably never see it. So programs need a place where they can send error messages so that the user will notice them. This is standard error, and it is usually connected to your console or window, even if you have redirected standard output of your program away from your screen.

For filter programs to work together, the format of the data has to be agreed upon. The most straightforward and easiest format to use is simply lines of text. Unix data files are generally just streams of bytes, with lines delimited by the ASCII LF (Line Feed) character, conventionally called a "newline" in the Unix literature. (This is `'\n'` if you're a C programmer.) This is the format used by all the traditional filtering programs. (Many earlier operating systems had elaborate facilities and special purpose programs for managing binary data. Unix has always shied away from such things, under the philosophy that it's easiest to simply be able to view and edit your data with a text editor.)

OK, enough introduction. Let's take a look at some of the tools, and then we'll see how to hook them together in interesting ways. In the following discussion, we will only present those command line options that interest us. As you should always do, double check your system documentation for the full story.

The `who` command

The first program is the `who` command. By itself, it generates a list of the users who are currently logged in. Although I'm writing this on a single-user system, we'll pretend that several people are logged in:

```
$ who
arnold   console Jan 22 19:57
```


```
miriam   ttyp0   Jan 23 14:19(:0.0)
bill     ttyp1   Jan 21 09:32(:0.0)
arnold   ttyp2   Jan 23 20:48(:0.0)
```

Here, the `$' is the usual shell prompt, at which I typed `who`. There are three people logged in, and I am logged in twice. On traditional Unix systems, user names are never more than eight characters long. This little bit of trivia will be useful later. The output of `who` is nice, but the data is not all that exciting.

The `cut` command

The next program we'll look at is the `cut` command. This program cuts out columns or fields of input data. For example, we can tell it to print just the login name and full name from the `` `/etc/passwd file' ``. The `` `/etc/passwd' `` file has seven fields, separated by colons:

```
arnold:xyzzy:2076:10:Arnold D. Robbins:/home/arnold:/bin/ksh
```

To get the first and fifth fields, we would use cut like this:

```
$ cut -d: -f1,5 /etc/passwd
root:Operator
...
arnold:Arnold D. Robbins
miriam:Miriam A. Robbins
...
```

With the `-c' option, `cut` will cut out specific characters (i.e., columns) in the input lines. This command looks like it might be useful for data filtering.

The `sort` command

Next we'll look at the `sort` command. This is one of the most powerful commands on a Unix-style system; one that you will often find yourself using when setting up fancy data plumbing. The `sort` command reads and sorts each file named on the command line. It then merges the sorted data and writes it to standard output. It will read standard input if no files are given on the command line (thus making it into a filter). The sort is based on the machine collating sequence (ASCII) or based on user-supplied ordering criteria.

The `uniq` command

Finally (at least for now), we'll look at the `uniq` program. When sorting data, you will often end up with duplicate lines, lines that are identical. Usually, all you need is one instance of each line. This is where `uniq` comes in. The `uniq` program reads its standard input, which it expects to be sorted. It only prints out one copy of each duplicated line. It does have several options. Later on, we'll use the `-c' option, which prints each unique line, preceded by a count of the number of times that line occurred in the input.

Putting the tools together

Now, let's suppose this is a large BBS system with dozens of users logged in. The management wants the SysOp to write a program that will generate a sorted list of logged in users. Furthermore, even if a user is logged in multiple times, his or her name should only show up in the output once.

The SysOp could sit down with the system documentation and write a C program that did this. It would take perhaps a couple of hundred lines of code and about two hours to write it, test it, and debug it. However, knowing the software toolbox, the SysOp can instead start out by generating just a list of logged on users:

```
$ who | cut -c1-8
arnold
miriam
bill
arnold
```

Next, sort the list:

```
$ who | cut -c1-8 | sort
arnold
arnold
bill
miriam
```

Finally, run the sorted list through `uniq`, to weed out duplicates:

```
$ who | cut -c1-8 | sort | uniq
arnold
bill
miriam
```

The `sort` command actually has a `-u' option that does what `uniq` does. However, `uniq` has other uses for which one cannot substitute `sort -u'.

The SysOp puts this pipeline into a shell script, and makes it available for all the users on the system:

```
# cat > /usr/local/bin/listusers
who | cut -c1-8 | sort | uniq
^D
# chmod +x /usr/local/bin/listusers
```

There are four major points to note here. First, with just four programs, on one command line, the SysOp was able to save about two hours worth of work. Furthermore, the shell pipeline is just about as efficient as the C program would be, and it is much more efficient in terms of programmer time. People time is

much more expensive than computer time, and in our modern "there's never enough time to do everything" society, saving two hours of programmer time is no mean feat.

Second, it is also important to emphasize that with the *combination* of the tools, it is possible to do a special purpose job never imagined by the authors of the individual programs.

Third, it is also valuable to build up your pipeline in stages, as we did here. This allows you to view the data at each stage in the pipeline, which helps you acquire the confidence that you are indeed using these tools correctly.

Finally, by bundling the pipeline in a shell script, other users can use your command, without having to remember the fancy plumbing you set up for them. In terms of how you run them, shell scripts and compiled programs are indistinguishable.

After the previous warm-up exercise, we'll look at two additional, more complicated pipelines. For them, we need to introduce two more tools.

The first is the `tr` command, which stands for "transliterate." The `tr` command works on a character-by-character basis, changing characters. Normally it is used for things like mapping upper case to lower case:

```
$ echo ThIs ExAmPlE HaS MIXED case! | tr '[A-Z]' '[a-z]'
this example has mixed case!
```

There are several options of interest:

`-c'
: work on the complement of the listed characters, i.e., operations apply to characters not in the given set

`-d'
: delete characters in the first set from the output

`-s'
: squeeze repeated characters in the output into just one character.

We will be using all three options in a moment.

The other command we'll look at is `comm`. The `comm` command takes two sorted input files as input data, and prints out the files' lines in three columns. The output columns are the data lines unique to the first file, the data lines unique to the second file, and the data lines that are common to both. The `-1', `-2', and `-3' command line options omit the respective columns. (This is non-intuitive and takes a little getting used to.) For example:

```
$ cat f1
11111
22222
33333
44444
```


```
$ cat f2
00000
22222
33333
55555
$ comm f1 f2
        00000
11111
                22222
                33333
44444
        55555
```

The single dash as a filename tells `comm` to read standard input instead of a regular file.

Now we're ready to build a fancy pipeline. The first application is a word frequency counter. This helps an author determine if he or she is over-using certain words.

The first step is to change the case of all the letters in our input file to one case. "The" and "the" are the same word when doing counting.

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | ...
```

The next step is to get rid of punctuation. Quoted words and unquoted words should be treated identically; it's easiest to just get the punctuation out of the way.

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' | ...
```

The second `tr` command operates on the complement of the listed characters, which are all the letters, the digits, the underscore, and the blank. The `\012' represents the newline character; it has to be left alone. (The ASCII TAB character should also be included for good measure in a production script.)

At this point, we have data consisting of words separated by blank space. The words only contain alphanumeric characters (and the underscore). The next step is break the data apart so that we have one word per line. This makes the counting operation much easier, as we will see shortly.

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' |
> tr -s '[ ]' '\012' | ...
```

This command turns blanks into newlines. The `-s' option squeezes multiple newline characters in the output into just one. This helps us avoid blank lines. (The `>' is the shell's "secondary prompt." This is what the shell prints when it notices you haven't finished typing in all of a command.)

We now have data consisting of one word per line, no punctuation, all one case. We're ready to count each word:

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' |
```


```
> tr -s '[ ]' '\012' | sort | uniq -c | ...
```

At this point, the data might look something like this:

```
  60 a
   2 able
   6 about
   1 above
   2 accomplish
   1 acquire
   1 actually
   2 additional
```

The output is sorted by word, not by count! What we want is the most frequently used words first. Fortunately, this is easy to accomplish, with the help of two more `sort` options:

`-n'
: do a numeric sort, not an ASCII one

`-r'
: reverse the order of the sort

The final pipeline looks like this:

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' |
> tr -s '[ ]' '\012' | sort | uniq -c | sort -nr
 156 the
  60 a
  58 to
  51 of
  51 and
 ...
```

Whew! That's a lot to digest. Yet, the same principles apply. With six commands, on two lines (really one long one split for convenience), we've created a program that does something interesting and useful, in much less time than we could have written a C program to do the same thing.

A minor modification to the above pipeline can give us a simple spelling checker! To determine if you've spelled a word correctly, all you have to do is look it up in a dictionary. If it is not there, then chances are that your spelling is incorrect. So, we need a dictionary. If you have the Slackware Linux distribution, you have the file `/usr/lib/ispell/ispell.words', which is a sorted, 38,400 word dictionary.

Now, how to compare our file with the dictionary? As before, we generate a sorted list of words, one per line:

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' |
> tr -s '[ ]' '\012' | sort -u | ...
```

Now, all we need is a list of words that are *not* in the dictionary. Here is where the `comm` command comes in.

```
$ tr '[A-Z]' '[a-z]' < whats.gnu | tr -cd '[A-Za-z0-9_ \012]' |
> tr -s '[ ]' '\012' | sort -u |
> comm -23 - /usr/lib/ispell/ispell.words
```

The `-2' and `-3' options eliminate lines that are only in the dictionary (the second file), and lines that are in both files. Lines only in the first file (standard input, our stream of words), are words that are not in the dictionary. These are likely candidates for spelling errors. This pipeline was the first cut at a production spelling checker on Unix.

There are some other tools that deserve brief mention.

`grep`
: search files for text that matches a regular expression

`egrep`
: like `grep`, but with more powerful regular expressions

`wc`
: count lines, words, characters

`tee`
: a T-fitting for data pipes, copies data to files and to standard output

`sed`
: the stream editor, an advanced tool

`awk`
: a data manipulation language, another advanced tool

The software tools philosophy also espoused the following bit of advice: "Let someone else do the hard part." This means, take something that gives you most of what you need, and then massage it the rest of the way until it's in the form that you want.

To summarize:

1. Each program should do one thing well. No more, no less.
2. Combining programs with appropriate plumbing leads to results where the whole is greater than the sum of the parts. It also leads to novel uses of programs that the authors might never have imagined.
3. Programs should never print extraneous header or trailer data, since these could get sent on down a pipeline. (A point we didn't mention earlier.)
4. Let someone else do the hard part.
5. Know your toolbox! Use each program appropriately. If you don't have an appropriate tool, build one.

As of this writing, all the programs we've discussed are available via anonymous `ftp` from

`prep.ai.mit.edu` as `` `/pub/gnu/textutils-1.9.tar.gz' `` directory.(1)

None of what I have presented in this column is new. The Software Tools philosophy was first introduced in the book Software Tools, by Brian Kernighan and P.J. Plauger (Addison-Wesley, ISBN 0-201-03669-X). This book showed how to write and use software tools. It was written in 1976, using a preprocessor for FORTRAN named `ratfor` (RATional FORtran). At the time, C was not as ubiquitous as it is now; FORTRAN was. The last chapter presented a `ratfor` to FORTRAN processor, written in `ratfor`. `ratfor` looks an awful lot like C; if you know C, you won't have any problem following the code.

In 1981, the book was updated and made available as Software Tools in Pascal (Addison-Wesley, ISBN 0-201-10342-7). Both books remain in print, and are well worth reading if you're a programmer. They certainly made a major change in how I view programming.

Initially, the programs in both books were available (on 9-track tape) from Addison-Wesley. Unfortunately, this is no longer the case, although you might be able to find copies floating around the Internet. For a number of years, there was an active Software Tools Users Group, whose members had ported the original `ratfor` programs to essentially every computer system with a FORTRAN compiler. The popularity of the group waned in the middle '80s as Unix began to spread beyond universities.

With the current proliferation of GNU code and other clones of Unix programs, these programs now receive little attention; modern C versions are much more efficient and do more than these programs do. Nevertheless, as exposition of good programming style, and evangelism for a still-valuable philosophy, these books are unparalleled, and I recommend them highly.

Acknowledgment: I would like to express my gratitude to Brian Kernighan of Bell Labs, the original Software Toolsmith, for reviewing this column.

Go to the previous, next section.

Go to the previous section.

TOC

Index

1

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

r

s

t

u

V

W

Go to the previous section.

GNU @code{textutils}

(1)

Version 1.9 was current when this column was written. Check the nearest GNU archive for the current version.

V.E.R.A.

Virtual Entity of Relevant Acronyms

An overview of

common and not so common acronyms

in the field of computing

Edition 1.3 (Texinfo)

Released June 1998

Go to the next section.

-- 0 ---

100VG
: 100 Voice Grade [technology]

2S2D
: Double Sided - Double Density (FDD)

3DDDI
: 3D Device Dependent Interface (MS), "3D DDI"

3DRAM
: 3 Dimensional Random Access Memory (RAM)

3GL
: 3rd Generation Language

4GL
: 4th Generation Language

5GL
: 5th Generation Language

Go to the next section.

Go to the previous, next section.

-- A ---

AA
: Auto Answer (MODEM)

AAAAA
: Anonym Association Against Acronym Abuse slang

AAAI
: American Association for Artificial Intelligence org., AI, USA

AAAS
: American Association for the Advancement of Science org., USA

AAC
: Authorization and Access Control (IETF)

AAD
: Authorized AutoCAD Dealer (AutoCAD, CAD)

AADN
: American Association of DOMAIN Names org., USA, Internet

AAE
: Allgemeine AnschalteErlaubnis (Telekom)

AAF
: Advanced Authoring Format (MS)

AAIM
: Association for Applied Interactive Multimedia org.

AAIS
: Allied command europe ACCIS Implementation Strategy ACCIS, NATO, mil.

AAL
: ATM Adaption Layer (ATM)

AAMSI
: American Association for Medical Systems Informatics org., USA

AAP
: Applications Access Point

AAP
: Association of American Publishers org., USA

AARP
: Appletalk Address Resolution Protocol (Apple, AppleTalk)

AASP
: ASCII Asynchronous Support Package (ASCII)

AAT
: Average Access Time

AATP
: Authorized Academic Training Program (MS, MCSD)

AAUI
: Apple Attachment Unit Interface (Apple, AppleTalk)

ABA
: American Bankers Association Org., USA, banking

ABAP4
: Advanced Business Application Programming/4 (SAP, R/3, 4GL), "ABAP/4"

ABATS
: Automatic Bit Access Test System

ABBS
: Apple Bulletin Board System (Apple)

ABC
: Atanasoff-Berry-Computer

ABCD
: Altavista Business Card Directory (WWW)

ABEL
: Advanced Boolean Expression Language

ABEND
: ABnormal END (Netware)

ABI
: Application Binary Interface (POE)

ABIOS
: Advanced Basic Input Output System (IBM)

ABIST
: Automatic Built In Self Test (IBM)

ABM
: Asynchronous Balanced Mode

ABR
: Available Bit Rate (ATM, CBR, VBR, UBR, QOS)

ABS
: Apple Business Systems (Apple)

ABUI

Association of Banyan Users International org., Banyan, User group, Vorlaeufer, ENA

AC

Access Control (Token Ring, ...)

AC

Area Code

AC

Automatic Computer

AC3

[digital] Audio Compression - 3 Dolby, DVD, digital audio, "AC-3"

ACA

Asynchronous Communications Adapter

ACAS

Application Control Architecture Services (DEC)

ACC

Area Communication Controller (MODACOM)

ACC

Atari Competence Center

ACCIS

Automated Command and Control Information System mil., USA

ACCS

Army Command and Control System mil., USA

ACCU

Association of C and C++ Users org., UK

ACD

Automated Call Distribution (CTI)

ACDC

Alternating Current/Direct Current, "AC/DC"

ACDI

Asynchronous Communications Device Interface (IBM, CM/2)

ACE

Advanced Computing Environment manufacturer

ACF

Access Control Field

ACF

Advanced Communications Function (IBM)

ACF

Apple Communications Framework (Apple)

TOC

ACFC

Address and Control Field Compression (PPP)

ACFNCP

Advanced Communication Function / Network Control Program (IBM, ACF), "ACF/NCP"

ACFTCAM

Advanced Communication Function / Telecommunications Access Method (IBM, ACF), "ACF/TCAM"

ACFVTAM

Advanced Communication Function / Virtual Telecommunications Access Method (IBM, ACF), "ACF/VTAM"

ACH

Association for Computers and the Humanities org., USA

ACIA

Associacio Catalana d'Intelligencia Artificial org., Spanien, AI

ACIA

Asynchronous Communications Interface Adapter

ACID

Atomicity - Consistency - Isolation - Durability (DB, TP)

ACIS

American Commitee for Interoperable Systems org., USA

ACL

[MS] Access Compatibility Layer (MS, DB)

ACL

Access Control List (DCE, DFS, NDS)

ACL

Advanced CMOS Logic

ACL

Agent Control Language (Agents)

ACL

Association for Computational Linguistics org., USA

ACLF

Access Control List Facility (DCE)

ACM

Address Complete Message (ATM)

ACM

Association for Computing Machinery org., USA

ACME

A Company that Makes Everything slang

ACMS

Application Control Management System

ACO

AMP Communications Outlet

ACOPS

Automatic CPU Overheating Prevention System (GigaByte), "A-COPS"

ACOT

Apple Classrooms Of Tomorrow (Apple)

ACP

Active Configuration Profile (MODEM)

ACP

Auxillary Control Process

ACPA

Audio Capture and Playback Adapter (IBM)

ACPC

Apple Communications Protocol Card (Apple)

ACPI

Advanced Configuration and Power Interface (Intel, MS, Toshiba)

ACPT

Automated Corporate Planning Tool

ACR

Attenuation to Crosstalk Radio cable

ACR

Automatic Call Recording

ACS

Access Control System (DISA)

ACS

Advanced Communications System

ACS

Architekten, Computer, Systeme fair

ACS

Asynchronous Communication Server

ACS

Australian Computer Science

ACSE

Association Control Service Element (OSI, ISO, DIS 8649)

ACSI

Advanced/Atari Computer System Interface (Atari)

ACSMIB

ATM Circuit Steering Management Information Base (ATM, MIB), "ACS-MIB"

ACSNET

Academic Computing Services NETwork

ACSS

Aural Cascading Style Sheets (CSS, HTML, WWW)

ACT

Architecture Characterization Template (DISA)

ACTA

America's Carriers Telecommunications Association (USA)

ACTPU

ACTivate Physical Unit (SNA)

ACTS

Advanced Communications Technologies and Services Europe

ACU

Automatic Calling Unit

ACU

Automatic Client Update (Novell, Netware)

ACUTA

Association of College and University Telecommunication Administrators org., USA

AD

Authorisierter Distributor (DEC)

AD

Autonomous DOMAIN

ADA

Automatic Data Acquisitions

ADAPSO

Association of Data Processing Service Organisation org., USA

ADAPT

Architecture Design, Analysis, and Planning Tool

ADATP

Allied DATa processing Publication mil., USA

ADB

A DeBugger (Unix)

ADB

TOC

Apple Desktop Bus (Apple)

ADBS

Advanced Data Broadcasting System (DirecPC)

ADC

Adaptive Data Compression (MODEM)

ADC

Analog to Digital Converter

ADC

Apple Distribution Center (Apple)

ADC

Automatic Data Capture

ADCCP

Advanced Data Communications Control Procedure (ANSI)

ADD

Adapter Device Driver (OS/2)

ADDDC

Automatic Direct Distance Dialing System

ADE

Application Development Environment

ADE

Aufforderung zur DatenEingabe (BTX)

ADEPT

Administrative Data Entry for Processing Transmission mil., USA

ADES

Automatic Digital Encoding System

ADEW

Andrew Development Environment Workbench (ATK, Unix)

ADF

Access control Decision Function mil., USA

ADF

Automatic Document Feeder

ADI

Autocad Device Interface (CAD, AutoCAD)

ADI

AUTODIN-to-DISN Interface AUTODIN, DISN, mil., USA

ADL

[international conference on the] Advances in Digital Libraries (IEEE)

ADL
: Adventure Description Language

ADLC
: Asynchronous Data Link Control

ADM
: Add-Drop-Multiplexer

ADMA
: Area Division Multiple Access (Mobile Systems)

ADMD
: ADministration Management DOMAIN (X.400)

ADMS
: Access Device Messaging Specification banking, Visa

ADO
: Active Data Objects (ASP, ODBC, MS, IIS)

ADP
: Administrative Data Processing

ADP
: Advanced Data Processing

ADP
: Automatic Data Processing (ADP)

ADPCM
: Adaptive Delta Pulse Code Modulation

ADPE
: Automatic Data Processing Equipment (ADP)

ADPLO
: Automated Data Processing Liaison Office (ADP)

ADPS
: Automatic Data Processing System (ADP)

ADPT
: Automated Data Processing/Telecommunications (ADP), "ADP/T"

ADS
: Advanced Digital System

ADS
: Application Development System

ADS
: Automatic Distribution System

ADS

Automatic Documenting System (LAN)

ADS

Auxilary Data System

ADSIA

Allied Data Systems Interoperability Agency Org., NATO, mil.

ADSL

Asymmetric Digital Subscriber Line [technology] (BELLCORE, AT&T, DSL)

ADSL

Asymmetric Digital Subscriber Loop [modulation]

ADSP

Advanced Digital Signal Processor (DSP)

ADSP

Appletalk Data Stream Protocol (Apple, AppleTalk)

ADSR

Attack, Decay, Sustain, Release [generator] (VCA)

ADSU

ATM Data Service Unit (ATM)

ADT

Abstract Data Type

ADT

Access Developer's Toolkit (MS, DB, Windows)

ADT

Application Data Type

ADT

Application Development Tools (IBM, AS/400)

ADU

Automatic Dialing Unit

ADUA

Administrative Directory User Agent

ADV

Automatisierte DatenVerarbeitung

ADV

staatliche Akademie fuer DatenVerarbeitung Org., Uni Boeblingen, Germany

ADVS

Automatisiertes DatenVerarbeitungsSystem

ADX

Automatic Data eXchange

TOC

AE
: Apple Events (Apple)

AE
: Application Entity / Environment / Execution / Engineering (APE)

AEA
: American Electronics Association org., USA

AEB
: Analog Expansion Bus

AEC
: Advanced Error Correction (CD)

AEF
: Access control Enforcement Function

AEGIS
: Advanced Electronic Guidance and Instrumentation System

AEI
: Application Enabling Interface (IBM)

AEI
: Automatic Equipment Identification

AEIMP
: Apple Event Interprocess Messaging Protocol (Apple)

AEM
: Automatic Emulation Management (Brother)

AEOM
: Apple Events Object Model (Apple)

AEPIA
: Asociacion Espanola Para la Inteligencia Artificial org., Spanien, AI

AES
: Application Environment Service / Specification (OSF)

AES
: Automatic Emulation Switching (Lexmark)

AESEBU
: Auto Engineering Society/European Broadcasting Union digital audio, "AES/EBU"

AETE
: Apple Event Terminology Extension (Apple)

AEUT
: Apple Event User Terminology (Apple)

AF

Auxiliary carry Flag assembler

AFAIC

As Far As I'm Concerned slang, Usenet, IRC

AFAIK

As Far As I Know slang, Usenet, IRC

AFAMPE

[u.s.] Air Force Automated Message Processing Exchange Org., USA. mil.

AFC

Automatic Font Change

AFCET

Association Francaise pour la Cybernetique Economique et Technique Org., France

AFD

Automatic File Distribution

AFE

Apple File Exchange (Apple)

AFEB

AFrican EDIFACT Board org., EDIFACT, "AF/EB"

AFI

Authority and Format Indicator (NSAP, IDP)

AFI

Authority Frame Identifier

AFII

Association for Font Information Interchange org., USA

AFIN

Air Force Information Network Netzwerk, USA, mil.

AFIS

AutomatisiertesFingerabdruck-InformationsSystem (INPOL)

AFK

Away from Keyboard slang, Usenet, IRC

AFNET

Air Force Network Netzwerk, USA, mil.

AFP

Advanced Function Presentation (IBM)

AFP

Appletalk Filing Protocol (Apple, AppleTalk)

AFPA

Advanced Function Printing Architecture (IBM)

AFPDS
: Advanced Function Printing Data Stream (IBM)

AFT
: Authenticated Firewall Traversal (IETF)

AFTP
: Anonymous File Transfer Protocol (FTP)

AFUU
: Association Francaise des Utilisateurs d'Unix Org., France, Unix

AGA
: Advanced Graphics Adapter

AGC
: Automatic Gain Control (Audio)

AGCH
: Access Grant CHannel (GSM, CCCH)

AGEP
: ArbeitsGemeinschaft Elektronisches Publizieren [e.v.] org., DTP, "AgEP"

AGF
: Arbeitsgemeinschaft der GrossForschungseinrichtungen [deutschlands] org.

AGFMB
: ArbeitsGemeinschaft Freier MailBoxen org.

AGP
: Advanced / Accelerated Graphics Port (Intel, MMX, AGP)

AH
: [IP] Authentication Header (IPSEC, IPV6, RFC 1826)

AHIMA
: American Health Information Management Association org., USA

AHP
: Analytical Hierachy Process

AHPCRC
: Army High Performance Computing Research Center org., USA, HPC

AI
: Adobe Illustrator (Adobe)

AI
: Artificial Intelligence

AIA
: Application Integration Architecture

AIC

[SRI] Artificial Intelligence Center (SRI, AI)

AIC
: AIX windows Interface Composer (AIX, IBM)

AICA
: Associazione Italiana per l'Informatica ed il Calcolo Automatico Org., Italy

AID
: Attention Interrupt ??? [key] (IBM)

AID
: AUTODIN Interface Device AUTODIN, mil., USA

AIDAT
: African Internet Development Action Team org., Internet

AIDS
: Automatic Installation and Diagnostic Service

AIFF
: Audio Interchange File Format

AII
: Active Input Interface (UNI, PMD, FDDI)

AIIA
: Associazione Italiana per l'Intelligenza Artificiale Org., Italy, AI

AIIM
: Association for Information and Image Management org., USA

AIM
: Advanced Invar Mask (Display, ViewSonic)

AIM
: Alternate Input Method (OS/2)

AIM
: Apple, IBM, Motorola [consortium] Apple, IBM, Motorola, org.

AIM
: Association of Imaging Manufacturers org.

AIM
: ATM / Ascend Inverse Multiplexing [protocol] (ATM)

AIM
: Automatic Interface Management (Brother)

AIMS
: Apple Internet Mail Server (Apple)

AIMSRCF
: Association of Imaging Manufacturers ??? /Shared Resource Control Facility (Fujitsu),

"AIM/SRCF"

AIN

Advanced Intelligent Network (IN)

AIN

Auto Insert Notification (CD-R)

AIO

Asynchronous Input/Output

AIP

ATM Interface Processor (ATM)

AIR

Automatic Image Refinement (Canon), "A.I.R."

AIRTC

[international symposium on] Artificial Intelligence in Real-Time Control IFAC, IFIP, IMACS, conference

AIS

Alarm Indication Signal (UNI, ATM, OAM, DS3/E3)

AIS

Applied Information Sciences manufacturer

AIS

Applied Information Systems manufacturer

AIS

Arbeitgeber-Informations-Service (WWW)

AIS

Automated Information System

AISB

Association of Imaging Service Bureaus org.

AISE

Alarm Indication Signal-External (UNI, ATM), "AIS-E"

AISIG

[south australian] Artificial Intelligence Special Interest Group Org., Australia, AI

AISP

Association for Internet Service Providers org., USA, ISP

AISP

Association for Internet Service Providers org.

AISP

Association of Information Systems Professionals org., USA

AISSO

Automated Information Systems Security Officer mil., USA

AIST

Agency of Industrial Science and Technology org., Japan

AIT

Advanced Intelligent Tape (Sony)

AIT

Angewandte InformationsTechnik [verlags gmbh] manufacturer

AITEC

[research institute for] Advanced Information TEChnology org., Japan

AITO

Association Internationale pour les Technologies Objets org.

AIW

APPN Implementers Workshop org., APPN

AIX

Advanced Interactive eXecutive (IBM, Unix, OS)

AIXESA

Advanced Interactive eXecutive/Enterprise Systems Architecture (IBM, AIX), "AIX/ESA"

AJBS

Association of Japanese Business Studies org., Japan

AKI

Arbeitsgemeinschaft der deutschen KI Institute org., KI

AL

Artificial Life

AL

Assembly Language

ALAP

Appletalk Link Access Protocol (Apple, AppleTalk)

ALAP

As Late As Possible slang, Usenet

ALC

Adaption Layer Controller (ATM)

ALC

Arithmetic and Logic Circuits (IC)

ALC

Automatic Level Control

ALDC

Adaptive Lossless Data Compression (IBM)

ALE

Address Lifetime Expectation org., IETF, IP

ALE

Application Link Enabling (R/3, SAP)

ALE

Atlanta Linux Enthusiasts Linux, user group

ALGOL

ALGOrithmic Language

ALIWEB

Archie Like Indexing in the WEB (WWW)

ALK

Automatisierte LiegenschaftsKarte (ALTIS)

ALM

Application / Appware Loadable Module (Novell, Netware)

ALM

Asychronous Line Module

ALM

Asynchronous Line Multiplexer

ALOE

Apple Library for Object Embedding (Apple, OpenDoc)

ALR

Advanced Logic Research, inc manufacturer

ALS

Adjacent Link Station (IBM)

ALTEL

Association of Long-distance TELephone companies org., USA

ALTERNIC

ALTERnative Network Information Center (Internet, USA)

ALTS

Association for Local Telecommunications Services org., USA

ALU

Arithmetic and Logic Unit (CPU)

AM

Active Matrix (LCD)

AM

Amplitude Modulation

TOC

AM
: Asynchronous Mode

AMA
: Automatic Message Accounting

AMACS
: AMA Collection System (AMA)

AMARC
: AMA Recording Center (AMA)

AMASE
: AMA Standard Entry (AMA)

AMAT
: AMA Transmitter (AMA)

AMATPS
: AMA TeleProcessing System (AMA)

AMCD
: Active Matrix Color Display (AMD, LCD)

AMD
: Active Matrix Display (LCD)

AMD
: Advanced MicroDevices [inc.] manufacturer

AME
: Advanced Metal Evaporated [tape] (Seagate)

AME
: Advanced Modeling Extension (AutoCAD)

AMEL
: Active Matrix Electro Luminescent (AMD)

AMH
: Automated Material Handling

AMHS
: Automated Message Handling System (MHS)

AMI
: Alternate Mark Inversion (ISDN, T1)

AMI
: American Megatrends Incorporation manufacturer

AMI
: ATM Management Interface (ForeRunner)

AMIA

American Medical Informatics Association org., USA

AMIA

Australian Medical Informatics Association Org., Australia

AMIC

Apple Memory-mapped I/O Controller (Apple)

AMIGOS

Advanced Mobile Integration in General Operating Systems

AMIS

Audio Message Interface Standard

AMLCD

Active Matrix Liquid Crystal Display (AMD, LCD)

AMMA

Advanced Memory Management Architecture

AMME

Automated Multi-Media Exchange

AMNET

Allgemeines Mailbox NETzwerk BBS, network

AMPE

Automated Message Processing Exchange

AMPS

Advanced Mobile Phone Service (Mobile Systems, Motorola)

AMS

American Mathematical Society org., USA

AMS

Andrew Mail / Message System (Unix)

AMS

Asymetric Multiprocessing System

AMS

AUTODIN Mail Server AUTODIN, mil., USA

AMSDOS

AMStrad Disk Operating System (Amstrad, OS), "AMS-DOS"

AMT

Active Matrix Technology (AMD, LDC)

AMT

Apple Media Tool (Apple)

AMTF

Average Modulation Transfer Function

AMTFT
: Active Matrix Thin Film Transistor (AMD, LCD)

AMTPE
: Apple Media Tool Programming Environment (Apple), "AMT PE"

AMWG
: Architecture Methodology Working Group org., DISA

AN
: Access Node

AN
: Alternating Network

ANBU
: ANlagenBUchhaltung

ANCS
: AT&T Netware Connect Services (AT&T, Netware)

ANDF
: Architecture Neutral Distributation Format (OSF)

ANFSCD
: And Now For Something Completely Different slang, Usenet, IRC

ANHR
: Access Node Hub Router (AN)

ANI
: Automatic Number Identification

ANMA
: Apple Network Managers Association org., Apple

ANMP
: Advanced ??? Network Management Protocol

ANR
: Access Node Router

ANS
: Advanced Network & Services Inc. (IBM, MCI. Merit, NFSnet)

ANS
: American National Standard (ANSI, ISO, USA)

ANSA
: Advanced Networked Systems Architecture (ISA)

ANSI
: American National Standard Institute org., USA

ANSPAG

TOC

Advanced Network System Performance and Application Group (RDN-CRC)

ANTC

Advanced Networking Test Center org., USA, AMD, FDDI

ANTS

AT&T Novell Telephone Services (AT&T, Novell), "A/NTS"

ANUML

Australian National University Meta Language, "ANU ML"

AOA

American Optometric Association org., USA

AOCE

Apple Open Collaboration Environment (Apple)

AOE

Application Operating Environment (AT&T)

AOI

Active Output Interface (UNI, PMD, FDDI)

AOL

America OnLine network

AOS

Algebraic Operating System (IBM)

AOTC

Australian Overseas Telecommunications Corporation Org., Australia

AOU

Arithmetic Output Unit

AOW

Asia and Oceania Workshop (OSI)

AP

Access Point (Wireless LAN)

AP

Application Processor

AP

Automatic Pagination

APA

Adaptive Packet Assembly

APA

All Points Addressable

APA

Arithmetic Processing Accelerator

APAR
: Authorized Program Analysis Report (IBM)

APC
: American Power Conversion manufacturer, UPS

APC
: ArbeitsPlatz Computer (SNI)

APC
: Association for Progressive Communications Org., network

APCUG
: Association of Personal Computer User Groups org.

APD
: Additional Product Documentation

APDA
: Apple Programmers and Developers Association org., Apple

APDU
: Application Protocol Data Unit (OSI, PDU, OSI/RM)

APE
: APplication Engineering

APEL
: A Portable EMACS Library (EMACS, GNU)

APEX
: Advanced Packet EXchange

APF
: Advanced Printer Function (IBM, ADT)

API
: Application Program Interface (API)

APIC
: Advanced Programmable Interrupt Controller (Intel, PIC)

APIS
: Arbeitsdatei PIOS Innere Sicherheit (INPOL, PIOS)

APL
: A Perspicious Language / Alles Parallel Loesbar APL, slang

APL
: A Programming Language (IBM)

APLSF
: A Programming Language with Shared Files (CMU, DEC)

APLSF

A Programming Language with Shared Variables (IBM)

APLV

Arbeitsdatei PIOS LandesVerrat (INPOL, PIOS)

APM

Advanced Power Management

APMS

Automated Program Management information System

APNIC

Asian Pacific Network Information Center org., Internet

APOK

Arbeitsdatei PIOS Organisierte Kriminalitaet (INPOL, PIOS)

APP

Application Portability Profile

APPC

Advanced Peer-to-Peer Communications (IBM, SNA, LU 6.2)

APPC

Advanced Program to Program Communication (IBM)

APPC

Application Program to Program Converter (IBM)

APPCM

Access Protection and Priority Control Mechanism (DQDB)

APPCP

Advanced Program to Program Communications Protocol (APPC, IBM)

APPI

Advanced Peer to Peer Internetworking (Cisco)

APPN

Advanced Peer to Peer Networking (IBM)

APR

Arbeitsdatei PIOS Rauschgift (INPOL, PIOS)

APS

Asynchronous Protocol Specification

APSE

Ada Programming Support Environment

APT

Address Pass Through

APT

Advanced Photoscale Technology (Brother)

APT
: Automatically Programmed Tools

APTC
: Advanced Processor Temperature Control (Chaintech)

APTM
: Application Program to Transaction Manager (TP)

APW
: Arbeitsdatei PIOS Waffen (INPOL, PIOS)

AR
: Administrative Request [message] (LFAP)

ARA
: Administrative Request Acknowledge [message] (LFAP)

ARA
: Apple[talk] Remote Access (Apple, AppleTalk)

ARAG
: AntiReflection AntiGlare (ViewSonic)

ARAP
: Apple Remote Access Protocol (Apple, AppleTalk)

ARB
: Architecture Review Board OpenGL, manufacturer, Org.

ARC
: Advanced RISC Computing [architecture] (ACE, RISC)

ARCA
: Advanced RISC Computing Architecture (ACE, RISC)

ARCNET
: Attached Resource Computer NETwork (Datapoint), "ARCnet"

ARE
: All Routes Explorer (ATM, ???)

AREXX
: Amiga Restructured EXtended eXecutor [language] (Amiga, Commodore)

ARIN
: American Registry for Internet Numbers org., Internet, USA

ARKD
: Abuse Resistant Key Distribution (Arcade)

ARL
: Access Rights List (Banyan, VINES)

ARLL

Advanced Run-Length Limited [encoding]

ARM

Advanced RISC Machines manufacturer, Acorn, Apple, VLSI, RISC

ARM

Annotated [c++] Reference Manual

ARM

Asnchronous Response Mode

ARMA

Association of Records Managers and Administrators, inc. org.

ARMS

Architecture for Reliable Managed Storage (Cheyenne)

ARMS

Automation Resources Management System

ARNS

Appletalk Remote Network Server (Apple, AppleTalk)

AROM

Alterable Read Only Memory

ARP

Address Resolution Protocol (Internet, RFC 826)

ARPA

Advanced Research Projects Agency org., USA

ARPANET

Advanced Research Projects Agency NETwork USA, network

ARPDP

Association of Rehabilitation Programs in Data Processing org., USA

ARQ

Automatic Re-transmission reQuest (MODEM)

ARRA

Announced Retransmission Random Access (MAC)

ART

Adaptive Resonance Theory (NN)

ART

Advanced Resolution Technology (Minolta)

ARTS

Accelerated Ray-Tracing System raytracing

ARTS

Asynchronous Remote Takeover Server

TOC

ARTT
: Asynchronous Remote Takeover Terminal

AS
: Advanced Server Windows NT

AS
: Authentication Service (DCE)

AS
: Autonomous System (IP, Internet, RFC 1930)

AS400
: Application System/400 (IBM), "AS/400"

ASA
: Advanced SCSI Architecture (SCSI)

ASA
: American Standards Association org., USA, ANSI, Vorlaeufer

ASAP
: As Soon As Possible slang, Usenet, IRC

ASAP
: Automatic Switching And Processing

ASBR
: Autonomous System Border / Boundary Router (AS)

ASC
: Accredited Standards Committee org., ANSI

ASC
: Additional Sense Code

ASC
: American Society for Cybernetics org., USA

ASC
: Authorized Support Center

ASC
: Automatic Contrast Selection (FAX)

ASCII
: American Standard Code of Information Interchange

ASCSI
: Advanced Small Computer Systems Interface (SCSI)

ASD
: Architecture Summary Design

ASDSP

Application-Specific Digital Signal Processor

ASE

Active Storage Element GigaB, IP-router

ASE

Aladdin Smartcard Environment (Fast, Aladdin)

ASE

Application Service Elemant (ATM)

ASEB

ASian EDIFACT Board org., EDIFACT, "AS/EB"

ASEMH

Army Standards Electronic Mail Host mil., USA

ASF

Advanced Streaming Format (MS)

ASH

Almquist SHell (BSD, Unix, Shell), "ash"

ASI

Adapter Support Interface (IBM, LAN)

ASI

Amorphous SIlicon drum (Kyocera), "aSI"

ASI

Aquarius Systems International manufacturer

ASI

Asynchronous SCSI Interface

ASI

Automatic System Installation

ASIC

Application Specific Integrated Circuit (IC)

ASIM

Arbeitskreis Simulation und Kuenstliche Intelligenz org., GI

ASIS

American Society for Information Science org.

ASK

Akademische Software Kooperation Org., Karlsruhe, Germany

ASK

Amplitude Shift Keying

ASKSAM

Access Stored Knowledge via Symbolic Access Method (DB), "askSam"

ASL
: Adaptive Speed Levelling (US Robotics)

ASLT
: Advanced Solid Logic Technology (IC)

ASM
: Advanced Server Management (Acer)

ASM
: Association of Systems Management org.

ASMP
: ASymetric MultiProcessing [system]

ASMP
: ASymmetrisches MultiProzessor [system]

ASN1
: Abstract Syntax Notation One (OSI, ISO, IS 8824), "ASN.1"

ASNM
: AdvanceStack Network Management (HP)

ASP
: Abstract Service Primitive (OSI)

ASP
: Active Server Pages (HTTP, MS)

ASP
: Advanced Signal Processing / Processor

ASP
: Appletalk Session Protocol (Apple, AppleTalk)

ASP
: Association of Shareware Professionals org., USA

ASP
: Authorized Service Provider (Sun)

ASPEC
: Advanced SPectre Entropy Coding MPEG, digital audio

ASPEN
: Automatic System for Performance Evaluation of the Network

ASPI
: Advanced SCSI Programming Interface (Adaptec, API, SCSI)

ASPIC
: Advanced ??? Programmable Interrupt Controller (PIC)

ASPIK

Algebraic Specifications and Implementations ??? (ISDV)

ASPS

Advanced Signal Processing System

ASQ

Automated Software Quality

ASQC

American Society of Quality Control org., USA

ASR

Automatic Server Restart (HP)

ASR

Automatic Speech Recognition

ASSERT

ADEPT Subsystem for Scanning of Electronic Received Traffic ADEPT, mil., USA

ASSIST

Automated Special Security Information System Terminal mil., USA

ASSPA

ASsociation Suisse pour l'Automatique org., Schweiz

ASSR

Agreed Set of Security Rules mil., USA

AST

Atlantic Standard Time (TZ)

ASTA

Advanced Software Technology and Agorithms (HPCC)

ASTRAL

Alliance fuer Strategic Token Ring Advancement and Leadership manufacturer

ASU

Asynchron-Synchron Umsetzer

AT

Advanced Technology (IBM, PC)

ATA

Advanced Technology Attachment

ATAPI

Advanced Technology Attachment Packet Interface (ATA)

ATASPI

Advanced Technology Attachment Software Programming Interface (ATA)

ATC

Address Translation Cache (CPU)

ATC
: Address Translation Controller (ATM)

ATC
: Automatic Transmission Control

ATCA
: Allied Tactical Communications Agency Org., NATO, mil.

ATCP
: [PPP] AppleTalk Control Protocol (Apple, AppleTalk, PPP, NCP, RFC 1378)

ATD
: Asynchronous Time Division (ATM)

ATDM
: Asynchronous Time Division Multiplexer (ATD)

ATDMA
: Asynchronous Time Division Multiple Access (ATD)

ATDP
: ATtention Dial Pulse (MODEM)

ATDT
: ATtention Dial Tone (MODEM)

ATE
: Asynchronous Terminal Emulation (Banyan, VINES)

ATE
: ATM Terminating Equipment (SONET, ATM)

ATEC
: Authorized Training and Education Center (MS, MOCS, MCSD)

ATES
: Advanced Techniques integration into Efficient scientific Software (ESPRIT, CASE)

ATF
: Automatic Track Finding (DDS)

ATH
: ATtention Hang up (MODEM)

ATI
: Advanced Telecomunications Institute org., USA

ATI
: Asociacion de Tecnicos de Informatica org., Spanien

ATIP
: Absolute Time In Pregroove (CD)

ATK

Andrew ToolKit (Unix)

ATKIS

Amtliches Topographisch-Kartographisches InformationsSystem

ATL

Activex Template Library (MS, ActiceX, MSVC)

ATL

Adaptive Threshold Learning neural nets

ATM

Abstract Test Method (ISO 9646-1)

ATM

Adobe Type Manager

ATM

Asynchronous Transfer Mode

ATM

Automatic Teller Machine

ATMARP

ATM Address Resolution Protocol (ATM, ARP)

ATMP

Ascend Tunnel Management Protocol (Ascend, RFC 2107)

ATMS

Assumption based Truth Maintenance System (AI)

ATN

Augmented Transition Network

ATOB

ASCII TO Binary (ASCII)

ATOMM

Advanced super Thinlayer and high-Output Metal Media (Fuji)

ATP

Advanced Technology Partner

ATP

Appletalk Transaction Protocol (Apple)

ATP

Application Transaction Program (IBM, APPC)

ATP2

AppleTalk Phase 2 (Apple, AppleTalk)

ATPS

AppleTalk Printing Services (AppleTalk, Apple)

ATR
: Advanced Telecommunications Research laboratory org., Japan

ATR
: Answer To Reset

ATR
: Automatic Terminal Recognition

ATRAC
: Adaptive TRansform Acoustic Coding Sony, digital audio

ATS
: Abstract Test Suite

ATS
: Administrative Terminal System

ATS
: Apple Terminal Services (Apple, AppleTalk)

ATT
: American Telephone and Telegraph, "AT&T"

ATX
: Advanced Technology eXtended [format]

AU
: Access Unit

AUAI
: Association for Uncertainty in Artificial Intelligence org., AI

AUC
: AUthentication Center (GSM), "AuC"

AUDIT
: AUtomated Data Input Terminal

AUDIUS
: AUssenDIenstUnterstuetzungsSystem (CAS)

AUE
: Andrew User Environment (Unix)

AUGE
: Apple User Group Europe org., Apple, User group

AUI
: Access Unit Interface

AUI
: Attachment Unit Interface ethernet

AUI

Audible User Interface (UI)

AUIS

Andrew User Interface System (Unix, UI)

AUP

Acceptable Use Policy (NFSNet)

AURP

Appletalk Update-based Routing Protocol (Apple, Appletalk, RFC 1504)

AUT

Application Under Test

AUTODIN

AUTOmatic DIgital Network (DMS, DISA)

AUTOSEVOCOM

AUTOmated SEcure VOice COMmunications Vorlaeufer, SVIP, mil., USA

AUTOVON

AUTOmatic VOice Network predecessor, DSN, DISA

AUU

ATM User-to-User (ATM)

AUX

Apple UniX (Apple, Unix), "A/UX"

AV

AudioVisual [macintosh] (Apple)

AVA

Audio-Visual Authoring

AVAS

teile-Auftragserfassungs-, Verwaltungs- und AbrufSystem (MBAG)

AVATAR

Advanced Video Attribute Terminal Assembler and Recreator (BBS)

AVC

Audio-Visual Connection

AVCD

Audio-Visual Computer Display

AVE

AutoCAD Visualization Extension (AutoCAD)

AVI

Audio Video Interleaved (MS)

AVIS

AuftragsVerwaltungs- und InformationsSYstem (MBAG)

AVL
: Adelson, Veslkij and Laudis [tree]

AVM
: AudioVisuelles Marketing und computersysteme [gmbh] manufacturer

AVPD
: Anchor Volume Descriptor Pointer (CD-R, UDF, ISO 9660)

AVR
: Automatic Voice Recognition

AVR
: Automatic Voltage Regulation (USV)

AVS
: Adult Verification System (WWW)

AWAC
: Audio Waveform Amplifier and Converter (Apple)

AWADO
: Automatischer Wechselschalter in der AnschlussDose

AWD
: Automatische Waehleinrichtung fuer Datenverbindungen

AWE
: Autocad Windows Extension (AutoCAD)

AWG
: American Wire Gauge cable

AWK
: al Aho, peter Weinberger, brian Kernighan (Unix)

AWL
: AnWeisungsListe (DIN 19239)

AWS
: Apple Workgroup Server (Appletalk, Apple)

AWT
: Abstract Windows Toolkit (Java, Sun)

AWTAPI
: Abstract Windows Toolkit-Application Programmer Interface (AWT, API, Java)

AXE
: Application eXecution Environment

AZEB
: Australian/new Zealand EDIFACT Board org., EDIFACT, "AZ/EB"

TOC

Go to the previous, next section.

Go to the previous, next section.

TOC

-- B ---

B8ZS
: Binary 8 Zero Suppression [encoding] (ISDN, T1)

BACP
: Bandwidth Allocation Control Protocol

BAD
: Broken As Designed slang

BAGEL
: Bay Area GNU Enthusiasts League GNU, org., User Group

BAI
: Basic Access Interface (ISDN)

BALUN
: BALanced-UNbalanced [adapter] cable, "Balun"

BAM
: Bidirectional Associative Memory neural nets

BAM
: Block-Availibility-Map

BAM
: Bundesanstalt fuer Materialpruefung Org., Berlin, Germany

BAP
: [PPP] Bandwidth Allocation Protocol (PPP, RFC 2125)

BAPC
: [PPP] Bandwidth Allocation Control Protocol (PPP, BAP, RFC 2125)

BAPCO
: Business Application Performance COrporation org., Compaq, Dell, HP, IBM, MS, Lotus, Intel, ..., "BAPCo"

BAPI
: Business Application Programmer's Interface (SAP, R/3, API)

BAPT
: BundesAmt fuer Post und Telefon org.

BARRNET
: Bay Area Regional Research NETwork network, "BARRNet"

BART

Basic Application RunTime (OS/2, IBM)

BARWAN

Bay Area Research Wireless Access Network network, USA

BASH

Bourne-Again SHell (Unix, Shell)

BASIC

Beginner's All-purpose Symbolic Instruction Code

BASIN

Bundesweites Alternatives Studentisches InformationsNetzwerk WWW, org.

BAT

Baby Advanced Technology [board] (AT)

BAYSIS

BAY networks' Switched Internetworking Services, "BaySIS"

BB

BridgeBoard (Amiga, Commodore)

BBC

Broadband Bearer Capability (B-ISDN)

BBL

Be Back Later slang, Usenet, IRC

BBN

Bolt, Beranek and Newman manufacturer

BBR

Back Bone Ring

BBS

Bulletin Board System

BCBDS

Broadband Connectionless Data Bearer Service (B-ISDN)

BCC

Base Communications-computer Center mil., USA

BCC

Blind Carbon Copy (DFUE)

BCC

Block Check Character

BCCH

Broadcast Control CHannel (GSM)

BCD

Binary Coded Decimal

BCDBS
: Broadband Connectionless Data Bearer Service (ATM)

BCDIC
: Binary Coded Decimal Interchange Code

BCDMA
: Broadband Code Division Multiple Access (Interdigital, SNI, Samsung), "B-CDMA"

BCF
: Base station Control Function (BS, BTS, GSM)

BCI
: Batibus Club International org.

BCI
: Brain Computer Interface

BCN
: Backbone Concentrator Node (Wellfleet)

BCOB
: Broadband Class of Bearer (B-ISDN)

BCP
: [PPP] Bridging Control Protocol (PPP, RFC 1638)

BCP
: Basic Call Process (IN)

BCP
: Binary Communications Protocol

BCP
: Binary Control Protocol (Adobe, PS)

BCPL
: Basic / BBN Combined Programming Language (BBN)

BCRFS
: Bell System Reference Frequency Standard

BCS
: Banking Communication Standard banking

BCS
: Binary Compatibility Standard (Motorola)

BCS
: British Computer Society org., UK

BCSM
: Basic Call Sate Modell (IN)

BCU

Bus Controller Unit

BDA

Borland Database ??? (Borland, DB)

BDA

BundesDatenAutobahn [e.v] org., ISP

BDE

BetriebsDatenErfassung

BDE

Borland Database Engine (Borland, Delphi, DB)

BDF

Bitmap Description / Display Format (Adobe, Fonts)

BDK

[java]Beans Development Kit (Java)

BDOS

Basic Disk Operating System (CP/M)

BDR

Bus Device Request (SCSI)

BDSG

BundesDatenSchutzGesetz

BEAT

Best Enhanced Advanced Technology (Trident, AT)

BEC

Back-End-Chip (DVR)

BECEEP

BErlin Continuing Engineering Education Program, "BeCEEP"

BECN

Backward Explicit Congestion Notification (ATM)

BED

Bookmark Exploring Dabbler (VRML)

BEDO

Burst EDO [DRAM] (EDO, RAM, DRAM, IC, Micron)

BEDODRAM

Burst Extended Data Out DRAM (RAM, DRAM, IC), "BEDO-DRAM"

BEEV

Bundesverband der Elektronik- und ElektroschrottVerwerter

BELLCORE

BELL COmmunications REsearch org., USA, "Bellcore"

TOC

BEOS
: Be Operating System (OS), "BeOS"

BER
: Basic Encoding Rules [for ASN.1] (ASN.1, OSI, ISO, IS 8825)

BER
: Bit Error Rate

BERKOM
: BERliner KOMmunikationssystem network

BERT
: Bit Error Rate Test

BES
: Block Ended by Symbol IBM, assembler

BES
: Bursty Errored Seconds (DS1/E1)

BEST
: Borland Enhanced Support and Training (Borland)

BEVU
: Bundesvereinigung mittelstaendischer Elektro- und elektronikgeraete entsorgungs- und VerwertungsUnternehmen org.

BEW
: Business Engineering Workbench (R/3, SAP)

BF
: Bus Fraction [pin] (Intel, Pentium, CPU)

BFD
: Binary File Descriptor (Unix)

BFT
: Binary File Tranfer (DFUE)

BGA
: Ball Grid Array (CPU, IC)

BGI
: Borland Graphics Interface (Borland)

BGP
: Border Gateway Protocol (RFC 1267/1771, IP)

BGT
: Broadcast and Group Translators

BHCA
: Busy Hour Call Attemps

BHLI
: Broadband High Layer Information, "B-HLI"

BHN
: Bayerisches Hochschulnetz network

BHT
: Branch History Table (CPU)

BI
: Breidbart-Index (Usenet, ECP, EMP)

BIB
: Bus Interface Board

BIC
: Bit Independence Criterion cryptography

BIC
: Bus Interface Chip (DVR)

BICI
: Broadband InterCarrier Interface (B-ISDN), "B-ICI"

BIDS
: Borland International Data Structures (Borland)

BIF
: Benchmark Interchange Format (PLB)

BIG
: Bionet Intelligent Gateway (BioData)

BIGFON
: Breitbandiges Integriertes Glasfaser-Fernmelde-OrtsNetz

BIKOS
: BueroInformations- und KOmmunikationsSysteme org., GI

BIM
: Broadband Interface Module

BIMA
: British Interactive Multimedia Association org., UK

BIND
: Berkely Internet Name DOMAIN [software] (Unix)

BIOS
: Basic Input Output System / Support (PC)

BIP
: Bit Interleaved Parity (SONET, ...)

BIPV
: Bit Interleaved Parity Violation

BIRA
: Belgian Institute for Automatic Control Org., Belgium

BIS
: Business Information System

BISDN
: Broadband Integrated Services Digital Network (ATM), "B-ISDN"

BISP
: Business Information System Program

BISSI
: Broadband Inter-Switching System Interface (B-ISDN), "B-ISSI"

BISUP
: Broadband ISDN User's Part (B-ISDN), "B-ISUP"

BISYNC
: Binary SYNchronous Communications (IBM)

BIT
: Basic Interconnection Test (ISO 9646-1)

BIT
: Binary digIT

BITNET
: Because It's Time NETwork network

BIU
: Bus Interface Unit

BIX
: Byte Information eXchange

BK
: Buero Kommunikation

BKS
: BenutzerKoordinatenSystem (CAD)

BL
: Blue Lightning [processor family] (Intel)

BLADE
: Basic Linear Algebra for Distributed Environments

BLAST
: BLocked ASynchronous Transmission

BLER

BLock Error Rate (CD)

BLLI

Broadband Low Layer Information (BISDN), "B-LLI"

BLM

Bayerische Landeszentrale fuer neue Medien org.

BLOB

Binary Large OBject

BLOROB

BLOck based ROBot

BLSM

Base Level System Modernization

BMBF

BundesMinisterium fuer Bildung und Forschung org.

BMBW

BundesMinisterium fuer Bildung und Wissenschaft org.

BMDP

Bio-Medical Data Package

BMFT

BundesMinisterium fuer Forschung und Technik org.

BML

Business Management Layerr (TMN)

BMOS

Bipolar Metal Oxyd Semiconductor (IC)

BMS

Basic Mapping Support (CICS)

BMS

Broadcast Message Server (PCTE)

BMTA

Backbone Message Transfer Agent (MTA)

BMUG

Berkeley Macintosh User Group org., Apple, USA, User group

BN

Bridge Number

BNC

Baby N Connector slang

BNC

Bayonet Neill Concelman [connector]

TOC

BNC
: Bayonet Nipple Connector slang

BNC
: Bayonet Nut Coupling ???

BND
: BundesNachrichtenDienst

BNF
: Backus-Naur Form (TTCN, ...)

BNP
: Broadband Network Premises

BNT
: Broadband Network Termination (B-ISDN), "B-NT"

BNU
: Basic Networking Utilities (AT&T)

BOA
: Basic Object Adapter

BOCA
: Borland Object Component Architecture (Borland)

BOD
: Business Object Documents (OAG)

BOM
: Begining of Message

BOM
: Byte Order Mark (Unicode)

BONDING
: Bandwidth ON Demand INteroperability Group Org., manufacturer, AIM

BOPS
: Billion Operations Per Second

BORSCHT
: Battery, Overvoltage, Ringing, Signalling, Coding, Hybrid, Testing [functions] (PBX)

BOS
: Basic Operating System (AIX, IBM)

BOT
: Back On Topic slang, Usenet, IRC

BOT
: Beginning Of Tape

BOT

Build, Operate and Transfer networke

BP

Base Pointer [register] CPU, Intel, assembler

BPB

BIOS Parameter Block (BIOS, DOS, HDD, FDD)

BPDU

Bridge Protocol Data Unit (PDU)

BPF

Berkeley Packet Filter (BSD, Unix)

BPI

Bits Per Inch (HDD)

BPIP

Best Play for Imperfect Player (NEC)

BPL

Bytes Per Line

BPN

Back-Propagation Net (NN)

BPP

Bits Per Pixel

BPP

Bridge Port Pair

BPS

Bits Per Second (MODEM)

BPSK

Bi-Phase Shift Keying [modulation]

BPU

Branch Prediction Unit (CPU, MMX, Intel)

BPU

Branch Processing Unit (CPU)

BPV

BiPolar Violation [error event] (DS1/E1, DS3/E3)

BR

Boundary Representation (CAD, CAM)

BRAIN

Berlin Research Area Information Network network

BRAM

Broadcast Recognition Access Method (MAC)

BRAP
: Broadcast Recognition with Alternating Priorities (MAC)

BRB
: Be Right Back slang, Usenet, IRC

BRF
: Benchmark Reporting Format (PLB)

BRI
: Basic Rate Interface (ISDN)

BRI
: Bridge Router Interface ??? (Banyan)

BRIEF
: Basic Reconfigurable Interactive Editing Facility

BRIM
: Bridge/Router Interface Module

BRS
: Big Red Switch

BRTB
: Berlin Reginal TestBed (WIN)

BS
: BackSpace

BS
: Banded Signaling

BS
: Base Station (LA, GSM)

BS
: BetriebsSystem

BS2000
: BetriebsSystem 2000 (SNI, OS)

BSA
: Backbone Service Area

BSA
: Business Software Alliance Org., manufacturer

BSAM
: Basic Sequential Access Method (IBM, MVS)

BSC
: Base Station Controller (BS, BTS, GSM)

BSC

Binary Synchronous Coded

BSC

Binary Synchronous Communications [protocol] (IBM)

BSCM

Binary Synchronous Communications Module

BSD

Berkeley System / Software Distribution manufacturer, Unix

BSDI

Berkeley Software Design Incorporated manufacturer

BSI

Bentley Systems, Incorporated manufacturer

BSI

British Standards Institute org., UK

BSI

Bundesamt fuer Sicherheit in der Informationstechnik org.

BSIC

Base Station Identification Code (BS, BCCH, GSM)

BSP

BetriebsSystemProzessor Windows NT, SMP

BSP

Binary Space Partioning [tree]

BSS

Block Started by Symbol IBM, assembler, Unix

BSS

Broadband Switching System

BSSMAP

Base Station System Management Application Part (RR, BS, MTP, GSM)

BST

Bering Strait Time [-1100] (TZ)

BST

Brasil Standard Time [-0300] (TZ)

BST

British Summer Time [+0100] (TZ, UK)

BSVC

Broadcast Switched Virtual Connections (ATM)

BT

Baghdad Time [+0200] (TZ)

BT
: Bering Time [-1100] (TZ)

BT
: Burst Tolerance

BT
: Bus Terminator

BTAM
: Basic Tape Access Method (BS2000)

BTAM
: Basic Telecommunications Access Method (IBM)

BTB
: Branch Target Buffer (CPU)

BTC
: Biting The Carpet slang, Usenet

BTE
: Broadband Terminal Equipment (B-ISDN), "B-TE"

BTL
: Bell Telephone Laboratories

BTLB
: Block Translation Look-aside Buffer (CPU)

BTM
: Benchmark Timing Methodology (PLB)

BTOA
: Binary TO ASCII (ASCII)

BTRON
: Business TRON (TRON)

BTS
: Base Tranceiver Station entities (BCF, BS, GSM)

BTU
: Basic Transmission Unit (IBM, SNA)

BTW
: By The Way slang, Usenet, IRC

BTX
: BildschirmTeXt network, Datex-J, Telekom, "Btx"

BTXVST
: BildschirmTeXt-VermittlungsSTelle (BTX), "Btx-VSt"

BUI

Bus Interface Unit (DEC)

BUS
: Broadcast and Unknown Server (ATM, LANE)

BV
: BildVerarbeitung

BVB
: BundesVerband fuer Buero- und informationssysteme [e.v.] org.

BVCP
: [PPP] Banyan VINES Control Protocol (RFC 1763, Banyan, VINES, PPP)

BVIT
: BundesVerband InformationsTechnologien [e.v.] org.

BWBM
: Bandwidth Balancing Mechanism (DQDB)

BWCC
: Borland Windows Custom Controls (Borland, DLL)

BZR
: BefehlsZaehlRegister

BZR
: Bit Zone Recording (ROD)

BZR
: BundesZentralRegister

BZT
: Bundesamt fuer Zulassungen in der Telekommunikation org., Telekom

Go to the previous, next section.

Go to the previous, next section.

-- C ---

C128
: Commodore 128 [computer] (Commodore)

C2IS
: Command and Control Information Systems mil., USA

C3I
: Command, Control, Communications and Intelligence mil., USA

C3IIS
: Command, Control, Communications and Intelligence Information Systems mil., USA, "C3I/IS"

C4
: Command, Control, Communications and Computers mil., USA

C4I
: Command, Control, Communications, Computers and Intelligence mil., USA

C64
: Commodore 64 [computer] (Commodore)

CA
: Cell Arrival (ATM)

CA
: Certification Authority cryptography

CA
: Computer Animation

CA
: Computer Associates manufacturer

CAAD
: Computer Aided Architectural Design

CABS
: Carrier Access Billing System

CAC
: Computer Aided Crime

CAC
: Connection Admission Control (UNI, ATM)

CACEAS
: Computer-Assisted Circuit Engineering and Allocating System

CACTIS
: Community Automated Counter-Terrorism Intelligence System mil., USA

CAD
: Computer Aided Dispatch / Drafting

CAD
: Computer Aided Design (CIM)

CADE
: Computer Aided Document Engineering (Microstar)

CADS
: Computer-Assisted Display System

CAE
: Client Application Enabler (IBM, DB)

CAE
: Common Application Environment (X/Open)

CAE
: Computer Aided Education

CAE
: Computer Aided Engineering (CIM)

CAI
: Computer Aided Inspection (CIM)

CAI
: Computer Aided Instruction

CAIP
: Computer Analysis of Images and Patterns conference

CAIS
: Common APSE Interface Specification

CAIT
: Central Academy of Information Technology org., MITI

CAL
: Computer Aided Logistics

CAL
: Computer Assisted Learning

CALS
: Computer aided Acquisition and Logistics Support

CALS
: Continuous Aquisition and Life-cycle Support

CAM

Common Access Method (SCSI)

CAM

Computer Aided Manufacturing

CAMAC

Computer Automated Measurement And Control

CAMM

Computer Assisted Material Management

CAN

Complete Area Networks (SNI)

CAN

Controller Area Network

CAO

Computer Aided Office

CAP

Carrierless Amplitude Phase [modulation] (ADSL, AT&T)

CAP

Communications-electronics Accommodation Program

CAP

Component Approval Process

CAP

Computer Aided Planning (CIM)

CAP

Computer Aided Publishing

CAPE

Computer Applications in Production and Engineering conference, IFIP

CAPI

Communication Application Program Interface (ISDN, API)

CAPI

Cryptography Application Pprogramming Interface cryptography, API

CAPP

Computer Aided Process Planning

CAPSL

CAnnon Printing System Language (Canon), "CaPSL"

CAQ

Computer Aided Quality [control]

CAR

Central Access Routing (RND)

CAR
: Computer Aided Retrieval

CAR
: Computer Assisted Radiology

CARCAS
: Computer Aided aRchiving and Change Accounting System

CARDS
: Central Archive for Reusable Defense Software mil., USA

CARE
: Computer Assistance Resource Exchange

CARLOS
: Computer Aided Real Language Orthographic System

CAS
: Column Address Strobe (IC, DRAM)

CAS
: Communicating Applications Specification (FAX, Intel, DCA)

CAS
: Computer Aided Selling

CAS
: Computer Algebra System

CAS
: Computerized Autodial System

CASE
: Common Application Service Element (ISO, OSI)

CASE
: Computer Aided Software Engineering

CASH
: Computer Aided Service Handling (Ashton-Tate), "C.A.S.H."

CAST
: ??? cryptography

CAST
: Computer Aided Software Testing

CAT
: Central Alaska Time [-1000] (TZ)

CAT
: Common Authentication Technology (IETF, RFC 1511)

CAT

Computer Aided Technology fair

CAT

Computer Aided Telephony

CAT

Computer Aided Testing

CATI

Computer Aided Telephone Interviewing

CATIS

Computer-Assisted Tactical Information System mil., USA

CATNIP

Common ArchiTecture for Next generation Internet Protocol (IPNG, RFC 1707)

CAUCE

Coalition Against Unsolicited Commercial Email org., Internet, Spam, UCE

CAV

Constant Angular Velocity (CD, HDD, MOD)

CAVE

Cave for Automated Virtual Environment (VR)

CAVO

Computer Associates - Visual Objects (CA, DB), "CA-VO"

CBASIC

Commercial Beginners All purpose Symbolic Instruction Code (BASIC)

CBC

Cipher Block Chaining [mode] DES, DESE, RC5, cryptography

CBCP

CallBack Control Protocol

CBCS

Computer Based Conversation System (BBS)

CBDF

Character Bitmap Distribution Format (Adobe)

CBDS

Connectionless Broadband Data Service Europe

CBE

Certified Banyan Engineer (Banyan)

CBEMA

Computer & Business Equipment Manufacturers Association org.

CBF

Code Behind Form (MS, Access, DB)

TOC

CBGA
: Ceramic Ball and Grid Array (IC)

CBIOS
: Compatibility Basic Input Output System (IBM, BIOS)

CBMS
: Connectionless Broadband Data Service

CBR
: Constant Bit Rate (ATM, VBR, ABR, UBR, QOS)

CBS
: Certified Banyan Specialist (Banyan)

CBT
: Canon Buffer Transmission (Fax)

CBT
: Computer-Based Training

CBT
: Core Based Tree [multicast protocol] (IP, RFC 1949)

CBX
: Computerized Branch eXchange (PBX)

CC
: Carbon Copy

CC
: Continuity Cell (ATM)

CC
: Country Code (MS-ISDN, GSM)

CC
: Cross Connector

CCAF
: Call Control Agent Function (IN)

CCC
: Chaos Computer Club org.

CCC
: Computer Control Center

CCC
: Cube Connected Cycles (MP)

CCCA
: Campus Computer Communication Association org., USA

CCCH

Common Control CHannel (GSM)

CCCI

Center for Cyber Communities Initiative org., Japan

CCD

Charge Coupled Device

CCE

[visual basic] Control Creation Edition (VB, ActiveX, MS)

CCE

Connection Control Entity

CCETT

Centre Commun d'Etudes de Telediffusion et Telecommunications Org., France

CCF

??? org.

CCF

Central Computer Facility

CCF

Connection Control Function (IN)

CCFL

Cold Cathode Fluorescent Lamp

CCFT

Cold Cathode Fluorescent Tube (LCD, Display)

CCIRN

Coordinating Committee of International Networks org.

CCIS

Common Channel Interoffice Signaling (AT&T)

CCITT

Comite Consultatif International Telegraphique et Telephonique org., ITU, Vorlaeufer

CCL

Cerberus Central Limited manufacturer

CCM

Change Configuration Management

CCNC

Common Channel Network Controller

CCNC

Computer / Communications Network Center

CCNUMA

Cache-Coherent Non Uniform Memory Access (SMP, NUMA), "cc-NUMA"

CCP
: [PPP] Compression Control Protocol (PPP, RFC 1962)

CCP
: Command Console Processor (CP/M)

CCP
: Compact Communication Products (TPS)

CCR
: Commitment, Concurrency and Recovery (OSI)

CCR
: Current Cell Rate (ATM)

CCRMA
: Center for Computer Research in Music and Acoustics org., Stanford, UK

CCS
: Cambridge Cybernetic Society org.

CCS
: Common Channel Signaling (IN)

CCS
: Common Command Set (SCSI)

CCS
: Common Communications Support (IBM, SAA)

CCS
: Communications-Computer Systems, "C-CS"

CCS7
: Common Channel signaling System 7 (IN, Telekom, CCITT)

CCSID
: Coded Character Set IDentification (IBM)

CCSY
: Cooperative Computing System Program (HP)

CCT
: China Coast Time [+0800] (TZ)

CCTA
: Central Computer and Telecommunications Agency org., UK

CCU
: Cache Control Unit (Wyse)

CCV
: C-bit Coding Violation [error event] (DS3/E3)

CD

Carrier Detect (MODEM, RS-232)

CD
: Change Directory (DOS, Unix, OS/2)

CD
: Committee Draft (ISO)

CD
: Compact Disk

CDA
: Communications Decency Act (Internet, USA)

CDA
: Compound Document Architecture (DEC)

CDBS
: Connectionless Data Bearer Service

CDBX
: Computerized Digital Branch eXchange (PBX)

CDC
: Control Data Corporation manufacturer

CDD
: Component Design Document

CDDA
: Compact Disk - Digital Audio CD, digital audio, "CD-DA"

CDDI
: Copper Distributed Data Interface (FDDI, UTP)

CDE
: Common Desktop Environment

CDE
: Compact Disk - Erasable (CD), "CD-E"

CDE
: Cooperative Development Environment (Oracle)

CDF
: Channel Definition Format (MS, Internet, XML)

CDF
: Compound Document Framework (IBM, OLE)

CDFS
: Compact Disk File System (CD, OS/2, IBM)

CDG
: Compact Disk + Graphics (CD), "CD+G"

TOC

CDI
: Compact Disk - Interactive (CD), "CD-I"

CDIF
: CASE Data Interchange Format (CASE)

CDK
: Control Development Toolkit (MS, VB)

CDM
: Compressed Data Mode

CDMA
: Code Division Multiple Access (DFUE)

CDMIDI
: Compact Disk + Musical Instruments Digital Interface (CD, MIDI), "CD+MIDI"

CDMO
: Compact Disk - Magneto Optical (CD), "CD-MO"

CDMS
: Communication Driver Maintenance System (ISDN, HST)

CDPC
: Cellular Digital Packet Data

CDPD
: Cellular Digital Packet Data (Mobile Systems)

CDR
: Compact Disk - Recordable (CD), "CD-R"

CDRA
: Character Data Representation Architecture

CDRAM
: Cached Dynamic Random Access Memory (RAM, DRAM, IC)

CDRM
: Cross DOMAIN Ressource Manager (VTAM, SSCP, IBM)

CDROM
: Compact Disk - Read Only Memory (CD), "CD-ROM"

CDROMXA
: Compact Disk - ROM / eXtended Architecture (CD, MPC), "CD-ROM/XA"

CDRW
: Compact Disk - ReWritable (CD), "CD-RW"

CDS
: Cell Directory Service (DCE)

CDS

Current Directory Structure (BIOS. DOS)

CDSA

Common Data Securuty Architecture HP, cryptography

CDSS

Creative Decision Stimulation Systems (AI, DSS)

CDT

Cell Delay Tolerance (ATN)

CDT

Central Daylight Time [-0500] (TZ, CST, USA)

CDTV

Commodore Dynamic Total Vision (Commodore)

CDV

Cell Delay Variation (UNI, ATM, QOS)

CDVT

Cell Delay Variation Tolerance (UNI, ATM, CDV)

CDWO

Compact Disk - Write Once (CD), "CD-WO"

CE

Communaute / Comunique Europeenne Europe

CE

Communications-Electronics, "C-E"

CE

Compact Edition (MS, Windows)

CE

Connection Endpoint (UNI)

CEARCH

Cisco Educational ARCHive (Cisco, WWW)

CEBIT

welt CEntrum Buero Information Telekommunikation fair, "CeBIT"

CEECEB

Central and Eastern European Countries EDIFACT Board org., EDIFACT, "CEEC/EB"

CEG

Continuous Edge Graphics (Grafik, IC)

CEI

Connection Endpoint Identifier (UNI)

CELP

Card Edge Low Profile [socket]

CELP

Code Excited Linear Prediction

CEN

Comite Europeen de Normalisation Europa, Brussels

CENELEC

Comite Europeen de Normalisation ELECtrotechnique Org., CEN, Europe

CEPAC

CMOS-Ein-Platinen-Allzweck-Computer (IC, CMOS, C'T)

CEPIS

Council of European Professional Informatics Societies Org., Europe

CEPT

Conference of European Postal and Telecommunications administrations Org., CCITT, conference, Europa

CER

Cell Error Ratio (ATM)

CERFNET

California Educational and Research Federation NETwork network, "CERFNet"

CERN

Conseil Europeenne pour la Recherche Nucleaire Org., Europe, Genf

CERT

Computer Emergency Response Team (DARPA, CMU, Internet)

CES

C-bit Errored Seconds (DS3/E3)

CES

Circuit Emulation Service

CES

Consumer Electronics Show fair, USA

CESAR

Central Employment Search And Retrieval (WWW)

CET

Central European Time [+0100] (TZ, MET)

CET

Centro de Estudes de Telecomunicoes org., Portugal

CF

Carry Flag assembler

CFA

Center for Architecture org., JIEO, DISA

TOC

CFA
: Code Fiels Address (Forth)

CFB
: Cipher FeedBack [mode] cryptography, DES

CFD
: Call For Diskussion (Internet)

CFD
: Computational Fluid Dynamics [applications]

CFE
: Center for Engineering org., JIEO, DISA

CFF
: Compact Font Format (Adobe)

CFI
: CAD Framework Initiative org., CAD

CFM
: ConFiguration Management (FDDI, SMT)

CFMC
: Committee to Fight Microsoft Corporation org., MS

CFP
: Call For Papers

CFS
: Center for Standards org., JIEO, DISA

CFV
: Call For Vote (Internet, Usenet), "CfV"

CGA
: Colour Graphics Adapter

CGA
: Graphics Communications Association org.

CGARI
: Graphics Communications Association Research Institute org., CGA

CGI
: Common Gateway Interface (WWW)

CGI
: Computer Generated Imagery

CGI
: Computer Graphics Interface

CGI
: Computer Graphics International conference

CGM
: Computer Graphics Metafile (ISO 8632)

CGMS
: Copy Generation Management System (CD)

CGRM
: Computer Graphics Reference Model (ISO, IEC, ISO/IEC 11072)

CGVDI
: ??? [Grafikbibliothek], "CG-VDI"

CHAID
: CHisquard Automatic Interaction Detector / Detection (SPSS)

CHAP
: [PPP] Challenge Handshake Authentication Protocol (PPP, RFC 1334/1994)

CHCP
: CHange Code Page (DOS)

CHDL
: Computer Hardware Description Language (HDL)

CHEST
: Computers in Higher Education Software Team org., UK

CHILL
: CCITT HIgh Level programming Language (CCITT)

CHRP
: Common Hardware Reference Platform (AIM)

CHS
: Cylinder Head Sectors

CHSM
: C Hardware Specific Module (NEST, MLID, Novell)

CI
: Check In (RCS)

CI
: Coded Information

CI
: Configuration Item (CM)

CI
: Congestion Indicator

CIAC

Computer Incident Advisory Capability org., LLNL, Internet

CIB

Computer Integrated Business

CIC

Carrier Identification Code

CIC

Coordination and Information Center (CSNET)

CICA

Center of Innovative Computer Applications org.

CICS

Customer Information Control System (IBM)

CICSESA

Customer Information Control System/Enterprise Systems Architecture (IBM), "CICS/ESA"

CICSVS

Customer Information Control System / Virtual Storage (IBM), "CICS/VS"

CID

Configuration - Installation - Distribution (IBM)

CIDIR

Classless Internet DOMAIN Routing [protocol] (???, CIDR)

CIDR

Classless Internet DOMAIN Routing [protocol] (RFC 1519)

CIE

Commission Internationale de l'Eclairage org.

CIFS

Common Internet File System

CIL

Computer Integration Laboratories org., Apple, IBM, Novell, Sun, ...

CILABS

Component Integration LABoratorieS org.,, OpenDoc, Apple, IBM, Adobe, ..., CILabs

CIM

Computer Integrated Manufacturing

CIO

Cisco Information Online (Cisco, WWW)

CIP

Carrier Identification Parameter

CIP

Classical IP over ATM (IP, ATM, IETF)

CIP
: Computer Integrated Processing

CIP
: Computer-Investitions-Programm

CIR
: Committed Information Rate (ATM)

CIRC
: Cross Interleaved Reed-salomon Code (CD)

CIRCIT
: Centre for International Research on Communication and Information Technology Org., Australia

CIS
: Card Information Structure / Space (PCMCIA)

CIS
: Command Information System mil., USA

CIS
: Compuserve Information Systems network

CISC
: Complex Instruction Set Computer (CPU)

CISE
: Computer and Information Science Directorate org., NSF

CISPR
: Comite International Special des Pertubations Radioelectriques org.

CISS
: Center for Information Systems Security Org., JIEO, DISA, mil., USA

CIT
: Computer Integrated Telephony

CIT
: Computer Intergrated Tooling

CITED
: Copyright In Transmitted Electronic Documents (ESPRIT)

CIVIC
: Cyclone Integrated Video Interfaces Controller (Apple)

CIX
: Commercial Internet eXchange (ISP)

CL
: ConnectionLess (CO)

CLASS

Centralized Local Area Selective Signaling

CLASS
: Custom Local Area Signaling Service

CLE
: Certified Lotus Engineer (Lotus)

CLI
: Call Level Interface (SAQ, X/Open, Informix, ...)

CLI
: CLear Interupt assembler

CLI
: Command Line Interpreter / Interface (OS)

CLID
: Calling Line IDentification

CLIM
: Common LISP Interface Manager (CLOS, LISP)

CLIW
: Configurable Long Instruction Word (IC, CPU)

CLL
: ConnectionLess Layer (UNI, NNI, ATM)

CLNAP
: Connectionless Network Access Protocol (UNI, NNI, ATM)

CLNP
: ConnectionLess Network Protocol (OSI, ISO 8473)

CLNS
: ConnectionLess Network Service

CLOS
: Common LISP Object System (LISP)

CLP
: Cell Loss Priority (UNI, ATM, CLR)

CLR
: Cell Loss Ratio (UNI, ATM, QOS)

CLS
: Card Loading Signal

CLSF
: ConnectionLess Service Function

CLSID
: CLasS IDentifier (COM)

CLTP
: ConnectionLess Transport Protocol (OSI)

CLUT
: Color LookUp Table (VGA)

CLV
: Constant Linear Velocity (CD, MOD)

CM
: Configuration Management

CM
: Configuration Manager (BIOS, PNP)

CM
: Connection Management (RR, MM, GSM)

CM2
: Communication Manager /2 (IBM), "CM/2"

CM5
: Connection Machine 5 (TMC)

CMC
: Common Messaging Calls [interface] (XAPIA)

CMC
: Complement Carry Flag assembler

CMC
: Computer Mediated Communications [studies centre] org., USA

CME
: Component Management Entity

CMGS
: Cyan Magenta Gelb Schwarz color system, DTP

CMI
: Coded Mark Inversion

CMI
: Connection Manager Interface (IBM, SNA)

CMIA
: China Medical Informatics Association org., China

CMIP
: Common Management Information Protocol (OSI, ISO, DP 9506, X.700)

CMIS
: Common Management Information Service (OSI)

CMISE

Common Management Information Service Element (CMIS)

CMM

Color Management Method (DTP, ICM)

CMMU

Cache/Memory Management Unit

CMOL

CMIP Over LLC (OSI, LLC)

CMOS

Complementary-symmetry Metal-Oxide Semiconductor (IC)

CMOT

CMIP Over TCP (OSI, RFC 1189, CMIP, TCP)

CMP

Cooperative Marketing Partner (DEC)

CMR

Cell Misinsertion Rate (ATM)

CMS

Code Management System (DEC, CM)

CMS

Color Management System

CMS

Conversational Monitor System (IBM, VME)

CMT

Connection ManagemenT (FDDI)

CMU

Carnegie-Mellon-University org.

CMY

Cyan Magenta Yellow color system, DTP

CMYK

Cyan Magenta Yellow blacK color system, DTP

CN

Communications Network

CN

Connection Management (Mobile Systems)

CN

Coordination Message (ISO 9646-3, TTCN)

CN

Copy Network

CN
: Corporate Network

CNA
: Communication Network Architecture (SEL)

CNA
: Communications Network Application

CNBC
: Center for Neural Basis of Cognition org., CMU, AI

CNC
: Communications Network Control

CNC
: Computerized Numerical Control

CND
: Caldera Network Desktop (Linux)

CND
: Caller Number Delivery (MODEM)

CNE
: Certified Netware Engineer (Novell, Netware)

CNEPA
: Certified Netware Engineers Professional Association org., Netware

CNET
: Centre Nationale d'Etudes des Telecommunications Org., France

CNI
: Certified Netware Instructor (Novell, Netware)

CNI
: Common Network Interface

CNIDIR
: Coalition for Networked Information DIRectories (Internet)

CNM
: Communications Network Management

CNM
: Customer Network Management

CNMA
: Communications Network for Manufacturing Applications

CNMS
: Cylink Network Management System

CNP

Corparate Network Products (TPS)

CNRCVUUCP

Compressed News ReCeive Via UUCP

CNS

Complimentary Network Service

CNS

Copuserve Network Services (CIS)

CO

Check Out (RCS)

CO

Connection Oriented (CL)

COAST

Cache On A STick (Intel)

COAST

Computer Operations, Audit and Security Technology org.

COBOL

COmmon Business Orientated Language

COBRA

??? Org., Netherlands

COCA

Cost Of Cracking Adjustment cryptography

COCOT

Customer Owned Coin Operated Telephone

COD

Connection Oriented Data

CODASYSL

Conference On DAta Systems Languages conference

CODCF

Central Office Data Connecting Facility

CODE

Client/server Open Development Environment (Powersoft)

CODE

COlor Depth Enhancement (ATI)

CODEC

COder - DECoder

COE

Central Office Equipment

COEES
: COE Engineering System (COE)

COFDM
: ??? digital audio

COFF
: Common Object File Format (Unix)

COIPX
: Connection Orientated Internet Packet eXchange (Novell, Netware, IPX), "CO-IPX"

COL
: Caldera Open Linux (Caldera, Linux)

COLD
: Computer Output on LaserDisk

COLIBRI
: COprozessor fuer LISP auf der Basis von RISC (RISC)

COM
: Component Object Model (OLE, OLE2, OCX, ActiveX, MS)

COM
: Computer Output on Microfilm

COM
: Continuation of Message

COMA
: Cache Only Memory Architecture (SMP)

COMAL
: COMmon Algorithmic Language

COMDEX
: COMputer Dealer's EXposition fair

COMPARTS
: COMputergestuetztes PARtner-TeilebestandsSystem (MBAG)

COMTECH
: COMputer TECHnologies fair

CONCERT
: Communications for North Carolina Education, Research and Technology network

CONCISE
: COSINE Network's Central Information Service for Europe COSINE, network

CONLAN
: ??? [hardware desription language] (HDL)

CONS

Connection Oriented Networking Service

CONTAC

Central Office NeTwork ACcess

COOL

COBOL Object Orientated Language (OOP, COBOL)

COOP

Concurrent Object Orientated Programming (OOP)

COP

Character-Oriented Protocol

CORAN

Communication ORiented Application aNalysis

CORBA

Common Object Request Broker Architecture (OMG)

CORDIS

COmmunity Research and Development Information Service Europe

COS

Class of Service

COS

Clip On Socket (CPU)

COS

Corporation for Open Systems org., OSI, User group

COSA

COmputerunterstuetzte SAchbearbeitung

COSA

COmputing in der Sozialen Arbeit Org, Koeln, Germany

COSE

Common Open Software Environment (HP, Sun, IBM, SCO, USL, Univel)

COSINE

Cooperation for OSI Networking in Europe org.

COSMOS

COmputer System for Mainframe OperationS

COSS

Common Object Services Specification

COST

COpenhagen SGML Tool (SGML), "CoST"

COT

Central Office Terminal

COTP
: Connection-Oriented Transport Protocol (OSI, ISO 8073)

COTS
: Connection-Oriented Transport [layer] Service

COW
: Character Orientated Windows (MS, SAA, UI)

CP
: Connection Processor

CP
: Control Point (IBM, SNA)

CP
: Coordination Point (ISO 9646-3, TTCN)

CPAN
: Comprehensive PERL Archive Network (PERL, FTP)

CPC
: Cost Per Copy

CPCS
: Common Part Convergence Sublayer (ATM)

CPDP
: Cellular Digital Packet Data

CPE
: Customer Premises Equipment

CPGA
: Ceramic Pin Grid Array (CPU)

CPH
: Cost Per Hour

CPI
: Characters Per Inch

CPI
: Common Part Indicator (ATM)

CPI
: Common Programming Interface (IBM, SAA)

CPI
: Computer Private branch exchange Interface

CPIC
: Common Programming Interface for Communications (IBM, SAA, API), "CPI/C"

CPIO

CoPy In/Out (Unix)

CPL

Combined Programming Language (DEC, PL/1)

CPL

Conversational Programming Language (DEC)

CPM

Control Program for Microcomputers (OS), "CP/M"

CPM

Cost Per Minute

CPM

Critical Path Method

CPMS

Central Point Management Services (Central Point)

CPN

Calling Party Number

CPN

Compuserve Packet Network network

CPN

Customer Premises Network

CPODA

Contention Priority Orientated Demand Assignment (MAC, PODA)

CPS

Central Processing System

CPS

Characters Per Second

CPSI

Configurable PostScript Interpreter

CPSR

Computer Professionals for Social Responsibility org., USA

CPU

Central Processing Unit

CPUID

Central Processing Unit IDentifier (CPU)

CR

Carriage Return (ASCII)

CRAFT

Cray Research Adaptive FORTRAN (Cray, MPP, FORTRAN)

TOC

CRAM
: Cache RAM (RAM)

CRAM
: Card Random Access Memory (RAM, IC)

CRAS
: Cable Repair Administrative System

CRC
: Cyclic Redundancy Check

CRCG
: [fraunhofer] Center for Research in Computer Graphics org., USA

CRCG
: Common Routing Connection Group

CRD
: Color Rendering Dictionary (PS)

CREN
: Corporation for Research and Educational Networking network

CRET
: Color - Resolution Enhancement Technology (HP), "C-REt"

CRFVC
: Connection Related Function Virtual Channel (UPC, UNI), "CRF(VC)"

CRFVP
: Connection Related Function Virtual Path (UPC, UNI), "CRF(VP)"

CRI
: Cray Research, Inc. manufacturer

CRIN
: Centre de Recherche en Informatique de Nancy Org., France

CRISC
: Complex-Reduced Instruction Set Computer

CRISP
: Complex-Reduced Instruction Set Processor

CRJE
: Conversational Remote Job Entry (RJE)

CRL
: Certificate Revocation List

CRLF
: Carriage Return - Line Feed (ASCII, DOS)

CRP

Common Reference Platform (PowerPC)

CRPC
: Center for Research on Parallel Computation (STC)

CRS
: Cell Relay Service (UNI, ATM)

CRT
: Cathode Ray Tube

CRT
: Computer Technology Research [corporation] provider

CRTC
: Cathose Ray Tube Controller (EGA, VGA, MCGA)

CS
: Carrier Selection

CS
: Chip Select (IC)

CS
: Client/Server, "C/S"

CS
: Code Segment [register] CPU, Intel, assembler

CS
: Computer Science

CS
: Controlled Slip [error event] (DS1/E1)

CS
: Convergence Sublayer (ATM)

CS1
: Capability Set 1 (IN), "CS-1"

CS2
: Capability Set 2 (IN)

CSA
: Callpath Service Architecture (IBM, CTI)

CSA
: Client Service Agent

CSA
: Configuration Status Accounting

CSAPI
: Common Speller Application Program Interface (API)

CSC
: Computer Sciences Corporation provider

CSCC
: Concurrent SuperComputing Consortium org.

CSCSI
: Canadian Society for the Computational Studies of Intelligence Org., Canada, AI

CSCW
: Computer Supported Cooperative Work

CSD
: Control flow Specification Diagram (CASE)

CSD
: Corrective Service Diskettes (IBM)

CSDC
: Circuit Switched Digital Capability

CSDC
: Code Segment Descriptor Cache [register] (CS, Intel, CPU)

CSE
: Client-Server Environment

CSEIA
: [conference on] Computer Support for Environmental Impact Assesment IFIP, conference

CSELT
: Centro Studi E Laboratori Telecomunicazioni [s.p.a.] Org., Italy

CSES
: C-bit Severely Errored Seconds (DS3/E3)

CSG
: Constructive Solid Geometry (CAD, CAM)

CSH
: C SHell (Unix, BSD, Shell)

CSH
: Complementary Software House (DEC)

CSI
: CompuServe Incorporated (ISP)

CSI
: Convergence Sublayer Indication (ATM)

CSID
: Caller Station IDentifikation (Fax)

CSIDS

Central command/Southern command Integrated Data System mil., USA

CSII

Center for Systems Interoperability and Integration org., ???

CSIRO

Commonwealth Scientific and Industrial Research Organization org., UK

CSIS

Central Schengen Information Systen SIS, Europe, Strassburg

CSL

Computer SoLutions [software gmbh] (Haendler)

CSLIP

Compressed [headers] Serial Line Internet Protocol (SLIP, IP)

CSMA

Carrier Sense Multiple Access

CSMACA

Carrier Sense Multiple Access with Collision Avoidance, "CSMA/CA"

CSMACD

Carrier Sense Multiple Access with Collision Detection IEEE 802.3, ethernet, "CSMA/CD"

CSMS

C Specific Media Support (NEST, MLID, Novell)

CSMUX

Circuit Switching MUltipleXer (FDDI), "CS-MUX"

CSN

Card Select Number (PNP)

CSNET

Computer + Science NETwork USA, network, BITNET

CSP

Centro Supercacolo Piemonte Org., Italy, HPC

CSP

Chip Scale Package (IC)

CSP

Communicating Sequential Processes

CSP

Cross System Product (IBM)

CSPDN

Circuit Switched Public Data Network (IN)

CSPDU

Convergence Sublayer Protocol Data Unit (ATM, PDU), "CS PDU"

CSPP
: Computer Systems Policy Project [group] Org., USA. manufacturer

CSR
: Cell misSequenced Ratio (ATM)

CSR
: Cell Switch Router (Toshiba)

CSS
: Cascading Style Sheets (HTML, WWW)

CSS
: Computer Sub System

CSS
: Content Scrambling System (DVD, Matsushita, IBM)

CSS
: Controlled Slip Seconds (DS1/E1)

CSS
: Customer Switching System

CST
: Central Standard Time [-0600] (TZ, CDT, USA)

CSTA
: Computer Supported Telecommunications Applications (ECMA, CTI)

CSTB
: Computer Science and Technology Board org., NRC

CSTC
: Computer Security Technology Center org., CIAC

CSTO
: Computing Systems Technology Office org., ARPA

CSTS
: Computer Supported Telecommunications Standard

CSU
: Channel Service Unit (ATM)

CSUNET
: California State University NETwork network, USA

CSV
: Comma Separated Values

CT
: [magazin fuer] Computer Technik, "c't"

CT

TOC

Chips & Technologies manufacturer, "C&T"

CTAN
: Comprehensive Tex Archive Network (TeX, FTP)

CTB
: Communication ToolBox (Apple)

CTCA
: Channel To Channel Adapter (IBM, System/370)

CTCP
: Client To Client Protocol (IRC)

CTCPEC
: Canadian Trusted Computer Product Evaluation Criteria (Canada)

CTD
: Cell Transfer Delay (UNI, ATM, QOS)

CTE
: Compliance Test and Evaluation, "CT & E"

CTERM
: Command TERMinal (DEC)

CTI
: Computer Telephony Integration

CTR
: [columbia university] Center for Telecommunications Research org., USA

CTR
: Common Technical Regulations Europe

CTRON
: Central TRON (TRON)

CTS
: Cipher Text Stealing [mode] cryptography

CTS
: Clear To Send (MODEM, RS-232)

CTS
: Conformance Testing Service (OSTC)

CTSM
: C Topology Specific Module (NEST, MLID, Novell)

CTSS
: Compatible / Cray Time Sharing System Unix, predecessor, OS

CTT
: Cartridge Tape Transport

CTT
: Character Translation Table

CTTC
: Cartridge Tape Transport Controller

CTV
: Cell Tolerance Variation (ATM)

CTY
: Console teleTYpe

CU
: Call Up (Unix)

CU
: [L] [see you [Later]] slang, Usenet, IRC

CUA
: Common User Application

CUI
: Character User Interface (UI)

CUI
: Common User Interface (AMS, UI)

CUSI
: Configurable Unified Search Interface (WWW)

CUSP
: Commonly Used System Programm (DEC)

CUT
: Control Unit Terminal

CUU
: ComputerUnterstuetzte Unterweisung

CVF
: Compressed Volume File (DOS)

CVIA
: Computer Virus Industry Association org., USA

CVS
: Computer Vision Syndrome

CVSELP
: Codex Vector Sum Excited Linear Prediction [algorithm] (Motorola, VOFR)

CWI
: Centrum voor Wiskunde en Informatica Org., Netherlands

CWIS

Campus Wide Information System

CWP

Current Workspace Pointer (SPARC, CPU)

CWT

Character Width Table

CXI

Common X-windows Interface (Unix)

CYMK

Cyan, Yellow, Magenta, blacK color system

Go to the previous, next section.

Go to the previous, next section.

-- D ---

D3D
: Direct3D (DirectX, MS)

D3DRM
: Direct3D Retained Mode (DirectX, MS)

DA
: Destination [MAC] Address (SNA, Token Ring, ATM, FDDI, ...)

DA11
: DatenAustauschphase 11 [allgemeine bauabrechnung] (GAEB)

DA81
: DatenAustauschphase 81 [leistungsverzeichnis] (GAEB)

DA82
: DatenAustauschphase 82 [kostenanschlag] (GAEB)

DA83
: DatenAustauschphase 83 [angebotsanforderung] (GAEB)

DA84
: DatenAustauschphase 84 [angebotsabgabe] (GAEB)

DA85
: DatenAustauschphase 85 [nebenangebot] (GAEB)

DA86
: DatenAustauschphase 86 [zuschlag/auftragserteilung] (GAEB)

DAA
: Device Access Architecture (Vireo)

DAA
: Digest Access Authentication (HTTP)

DAB
: Digital Audio Broadcasting

DAC
: Digital to Analog Converter

DAC
: Discretionary Access Control

DAC
: Dual Attached Concentrator (FDDI)

DACAPO
: ??? [hardware desription language] (HDL)

DACNOS
: Distributed Academic Computing Network Operating System (OS, HECTOR)

DACS
: Digital Access Control System (ISDN, DES)

DACT
: DAta Compression Technology

DAD
: Desktop Application Director (WordPerfect)

DAEMON
: Disk And Execution MONitor (Unix)

DAF
: Distributed Application Framework (CCITT)

DAG
: DatenAnschaltGeraet

DAI
: ??? (Sun)

DAI
: Device Application Interface (Novell, Netware, SMS)

DAI
: Distributed Artificial Intelligence (AI)

DAINET
: Deutsches AgrarInformationsNETz WWW, org.

DAL
: Data Access Language (Apple)

DAM
: Direct Access Mode

DAM
: Distributed Abstract Machine

DAM
: Draft AMendment (ISO)

DAMQAM
: Dynamically Adaptive Multicarrier Quadrature Amplitude Modulation

DANA
: De.Admin.News.Announce (Usenet), "D.A.N.A."

DANTE

Delivery of Advanced Network Technology to Europe Org., Europe

DANTE

Deutschsprachige ANwendervereinigung TEx [e.v.] TeX, user group

DAO

Data Access Objects (DB)

DAO

Destination Address Omitted [flag] (CATNIP)

DAO

Disk At Once (CD-R)

DAP

Data Access Protocol (DEC, DNA)

DAP

Developers Assistance Program (IBM)

DAP

Directory Access Protocol (X.500, DS)

DAP

Directory Application Protocol (IN)

DAP

Document Application Profile (JTC1, ODIF, ODA)

DAPHNE

Document Application Processing in a Heterogeneous Network Environment

DAPIE

Developper API Extensions (IBM, OS/2, API)

DARI

Database Application Remote Interface (IBM, DB)

DARMP

Defense Automation Resources Management Program mil., USA

DARPA

Defense Advanced Research Projects Agency org., USA

DAS

Directory Assistance Service [protocol] (RFC 1202)

DAS

Disk Array Subsystem (Unix, HP-UX)

DAS

Dual Attached Station (FDDI)

DAS

Dual Attachment Station (FDDI, Schneider & Koch)

DAS
: Dynamic Allocation Scheme [protocol]

DASD
: Direct Access Storage Device

DASI
: Dial Access Signaling Interface

DASP
: Drive Active, Slave Present (IDE)

DASS
: Distributed Authentication Security Service (RFC 1507)

DAT
: Digital Audio Tape digital audio

DATEV
: DATEnVerarbeitungszentrale der steuerberatenden Berufe Org., Nuernberg, Germany

DATEX
: DATa EXchange

DATEXJ
: DATa EXchange - Jedermann ??? (Telekom), "DATEX-J"

DATEXL
: DATa EXchange - Leitungsvermittlung

DATEXM
: DATa EXchange - Mbit ??? (Telekom), "DATEX-M"

DATEXP
: DATa EXchange - Packetized / Packetvermittlung (Telekom, X.25), "DATEX-P"

DAU
: Duemmster Anzunehmender User slang, Usenet, IRC

DAV
: Digital Audio Video Apple, digital audio

DAVIC
: Digital Audio Visual Interoperatibility Council Org., digital audio

DAVID
: Digital Audio Video Interactive Decoder digital audio

DAVIS
: DAimler-Benz Vertragspartner-InformationsSystem (MBAG)

DAWN
: Defense Attache Worldwide Network Netzwerk, mil.

DAX

Developper API eXtension (OS/2, IBM, API)

DB

DataBase

DB2

DataBase 2 (IBM, DB)

DB2CS

DataBase 2 Client/Server (IBM, DB2, DB), "DB2 C/S"

DB2SDK

DataBase 2 Software Development Toolkit (DB2, IBM, DB), "DB2 SDK"

DBA

DataBase Administrator (DB)

DBA

Drei Buchstaben Akronym

DBAC

DataBase Administration Center (DB)

DBAS

DataBase Administration System (DB)

DBCS

Double-Byte Character Set

DBDMA

??? Direct Memory Access (Apple)

DBI

DataBase Interface (DB)

DBL

DataBase Language (DB)

DBLT

Dynamic Back Link Technology (WWW)

DBM

DataBase Manager (DB)

DBME

DataBase Managment Environment (DB)

DBMS

DataBase Management System (DB)

DBP

DataBase Publishing (DB)

DBRAD

Data Base Relational Application Directory (DB)

DBRM
: Data Base Request Module (DB)

DBS
: Deutscher Bildungs-Server (DFN, WWW)

DBS
: Duplex Bus Selector

DBSC
: Dynamic Beam Spot Control (Eizo)

DBTG
: Data Base Task Group (CODASYL, DB)

DBVS
: DatenBankVerwaltungsSystem (DB)

DC
: Data Cartridge

DC
: Device Context

DCA
: Defense Communications Agency Org., USA, mil., Vorlaeufer, DISA

DCA
: Digital Communication Associates

DCA
: Digital Controlled Amplifier (VCA)

DCA
: Distributed Communication Architecture (Sperry Univac)

DCA
: Document Center Architecture

DCA
: Document Content Architecture (IBM, CCS)

DCA2
: Dynamic Cache Architecture [level] 2

DCAF
: Distributed Console Access Facility

DCAP
: Data link switching Client Access Protocol (DLSW, RFC 2114)

DCB
: Data Control Block

DCB

Disk Coprocessor Board (Novell, SCSI, HBA)

DCC

Data Communications Computer

DCC

Data Country Code (ATM)

DCC

Direct Client to Client (IRC)

DCC

Display Combination Code

DCC

DOS Command Center

DCCA

Dependable Computing for Critical Applications conference

DCCA

Distributed Component Computing Architecture (Star, C/S)

DCCH

Dedicated Control CHannel (GSM)

DCCS

DisContiguous Shared Segments

DCD

Data Carrier Detect (MODEM, RS-232)

DCDB

DOMAIN Control DataBase (DOMAIN)

DCE

Data Circuit terminating Equipment (X.25, CCITT, IBM, HP, DEC, Tandem, Sun)

DCE

Data Communications Equipment

DCE

Distributed Computing Environment (OSF)

DCERPC

Distributed Computing Environment / Remote Procedure Call (DCE, RPC), "DCE/RPC" , "DCE RPC"

DCI

Data Capture Interface (UMA)

DCI

Device Control Interface

DCI

Display Control Interface (MS, Windows, Intel)

DCL

Data Control Language

DCL

DEC Control Language (DEC)

DCL

Digital Command [scripting] Language (DEC, VMS)

DCLU

Digital Carrier Line Unit

DCLZ

Data Compression Lempel-Ziv

DCM

Digital Carrier Module

DCNA

Data Communication Network Architecture

DCO

Digital Controlled Oscillator

DCOM

Distributed Component Object Model (COM, MS, OLE, ActiveX)

DCP

Data Compression Protocol (Motorola)

DCPS

Data Communications Protocol Standards

DCR

Design Change Request (AIX, IBM)

DCRC

Digital Cellular Radio Conference GSM, conference

DCS

Data sharing Control System (NEC)

DCS

Defense Communications System mil., USA

DCS

Desktop Color Separation

DCS

Digital Cellular System (Mobile Systems)

DCS

Digital Colour System (Adobe, Photoshop)

DCS

Digital Control System (NEC)

DCS

Digital Cross-connect System (DEC)

DCT

Data Collection Terminator (BTX)

DCT

Discrete Cosinus Transformation (MPEG, JPEG)

DCTN

Defense Commercial Telephone Network mil., USA, Netzwerk

DCU

Data Cache Unit (CPU, POWER)

DD

Data Dictionary (SA, CASE, DB)

DD

Depacketization Delay

DD

Double Density [disks] (FDD)

DDA

DOMAIN Defined Attribute (DOMAIN)

DDBMS

Distributed DataBase Management System (DBMS, DB)

DDC

Device Color Characterization (XCMS)

DDC

Display Data Channel (VESA)

DDCMP

Digital Data Communication Message Protocol

DDCS2

Distributed Database Connection Services /2 (IBM, DB, DRDA), "DDCS/2"

DDD

Data Display Debugger (GNU)

DDE

DatenenDEinrichtung

DDE

Dynamic Data Exchange

DDES
: Digital Data Exchange System (ANSI)

DDI
: Device Dependent Interface

DDIMM
: Dual [RAS] Dual Inline Memory Module (DIMM, RAS), "D-DIMM"

DDK
: Device Development / Driver Kit (MS)

DDL
: Data Definition Language

DDL
: Document Description Language

DDM
: Distributed Data Management (IBM, CCS)

DDN
: Defense Data Network USA, Netzwerk, mil.

DDNS
: Distributed DOMAIN Naming Service (TCP/IP)

DDNS
: Dynamic DOMAIN Name Service (OS/2, IBM)

DDP
: Datagram Delivery Protocol (AppleTalk)

DDP
: Distributed Data Processing

DDR
: Dynamic Desktop Router (Cogent)

DDRS
: Defense Data Repository System mil., USA

DDS
: Digital Data Service / System

DDS
: Digital Data Storage (Sony, HP, DAT, ISO, ANSI, ECMA)

DDS
: Direct Digital Sampling (CD-RW, SCSI)

DDS
: Distributed Directory Service (DCE)

DDSA

Digital Data Service Adapter

DDSDC

Digital Data Storage - Data Compression (DDS, DCLZ), "DDS-DC"

DDSS

Double Dynamic Suspension System (Asus, CD-ROM)

DDT

Dynamic Debugging Tool (DEC)

DDU

Dialog Data Unit (BTX)

DDV

DatenDirektVerbindung (Telekom)

DDV

Dialog Data Validation

DDVS

Daimler-benz DatenVerbundSystem (MBAG)

DDVT

Dynamic Dispatch Virtual Tables

DDX

Distributed Data eXchange

DEA

Deterministischer Endlicher Automat

DEBI

DMA Extended Bus Interface (Acorn, DMA)

DEC

Digital Equipment Corporation manufacturer

DECIX

DEutscher Commercial Internet eXchange (Internet), "DE-CIX"

DECNET

Digital Equipment Corporation NETwork (DEC)

DECT

Digital European Cordless Telecommunications (DFUE)

DECUS

Digital Equipment Computer Users Society org., DEC, User group

DEE

DatenEndEinrichtung

DELNI

Digital Ethernet Local Network Interconnect ethernet

DELQA
: Digital Ethernet Lowpower Q-bus network Adapter ethernet

DELTA
: Developing European Learning through Technology Advance

DELUA
: Digital Ethernet Lowpower Unibus network Adapter ethernet

DELUG
: DEutsche Linux User Group org., User group, Linux

DEMARC
: Distributed Enterprise Managemant ARChitecture (Banyan), "DeMarc"

DEMPR
: Digital Ethernet Multi-Port Repeater ethernet

DEN
: Document Enabled Networking (Novell, Xerox)

DENIC
: DEutsches Network Information Center, org., Internet, "DE-NIC"

DEPCA
: Digital Ethernet Personal Computer-bus Adapter ethernet

DEQNA
: Digital Ethernet Q-bus Network Adapter ethernet

DEREP
: Digital Ethernet REPeater ethernet

DES
: Data Encryption Standard cryptography, NIST, IBM

DES
: Destination End System

DESCBC
: Data Encryption Standard/Cipher Block Chaining (DES), "DES/CBC"

DESE
: [PPP] Data Encryption Standard Encryption protocol (PPP, RFC 1969)

DESP
: Data Element Standardization Program

DESPR
: Digital Ethernet Single Port Repeater ethernet

DESTA
: Digital Ethernet thin-wire STation Adapter ethernet

DETEBERKOM

DEutsche TElekom BERliner KOMmunikationssystem, "DeTeBerkom"

DEUNA

Digital Ethernet Unibus Network Adapter ethernet

DF

Direction Flag assembler

DF

Disk Free (Unix)

DFD

Data Flow Diagram / DatenFlussDiagramm (CASE, SA)

DFDSM

Data Facility Distributed Storage Management (IBM)

DFG

Deutsche ForschungsGemeinschaft org.

DFI

Digital Facility Interface

DFKI

Deutsches Forschungszentrum fuer Kuenstliche Intelligenz org., KI

DFN

Deutsches ForschungsNetz [e.V.] org., ISP

DFNCERT

Deutsches ForschungsNetz Computer Emergency Response Team (DFN, Internet), "DFN CERT"

DFP

Data Facility Product

DFP

Distributed Functional Plane (IN)

DFS

Direct File System (Novell, Oracle)

DFS

Distributed File System (DCE)

DFT

Discrete Fourier Transformation

DFT

Distributed Function Terminal (IBM)

DFU

Data File Utility (IBM, ADT)

DFUE

DatenFernUEbertragung

DFV
: DatenFernVerarbeitung

DGCC
: DISA Global Control Center (DISA)

DGD
: Deutsche Gesellschaft fuer Dokumentation [e.v.] org.

DGIS
: Direct Graphics Interface Standard

DGP
: Dissimilar Gateway Protocol

DGPS
: Differential Global Positioning System (GPS)

DGS
: Display GhostScript (GNU, GNUStep, PS)

DGSA
: Defense Goal Security Architecture mil., USA

DGUX
: Data General / UniX (Unix), "DG/UX"

DHCF
: Distributed Host Command Facility (IBM, CCS)

DHCP
: Dynamic Host Configuration Protocol (TCP/IP, IETF, RFC 2131)

DHIS
: Distributed Heterogeneous Information Systems

DI
: Destination Index [register] CPU, Intel, assembler

DIA
: Deutsche Informatik Akademie org.

DIA
: Document Interchange Architecture (IBM, CCS)

DIAMOND
: Development and Integration of Accurate Mathematical Operations in Numerical Data-processing (ESPRIT)

DIANE
: DIrect Access Network for Europe

DIB
: Defense Information Base mil., USA

DIB
: Device Independent Bitmap

DIB
: Directory Information Base (X.500, DS)

DIB
: DOS Info Block (BIOS, DOS)

DIB
: Dual Independent Bus

DIC
: Digital Interface Controller

DID
: Digital Image Design

DIESEL
: Dumb Interpretatively Evaluated String Expression Language (AutoCAD)

DIF
: Document Interchange Format

DIFMOS
: Double Injection Floating Gate MOS

DIGI
: Deutsche InteressenGemeinschaft Internet [e.v.] org., ISP

DII
: Defense Information Infrastructure mil., USA, DISA

DII
: Dynamic Invocation Interface

DIICC
: Defense Information Infrastructure Control Concept mil., USA

DIICOE
: Defense Info Infrastructure Common Operating Environment DISA, mil., USA, "DII COE"

DIL
: Dual InLine

DIME
: Desktop Integrated Media Environment (COSE)

DIME
: DIrect Memory Execute (AGP)

DIMSS
: DSN Integrated Management Support System DSN, mil., USA

DIN
: Deutsches Institut fuer Normung org.

DINAH
: Desktop INterface to AUTODIN Host AutoDIN, mil., USA

DINO
: Deutsches InterNet Organisationssystem WWW, Uni Goettingen, Germany

DIP
: Dial-up Internet Protocol (Linux)

DIP
: Dual In-line Package (IC, DRAM)

DIRMU
: DIstributed and Reconfigurable MUltiprocessor (MP)

DIS
: Defense Information System mil., USA

DIS
: Digital Identification Signal (HDLC)

DIS
: Draft International Standard (ISO)

DISA
: Data Interchange Standards Association org.

DISA
: Defense Information Systems Agency Org., mil., USA

DISAIS
: DISA Information System mil., USA, "DISA-IS"

DISANET
: DISA Information Network mil., USA, DISA, Netzwerk, "DISANet"

DISC
: Defense Information System Council mil., USA

DISK
: Deutschsprachige Internationale SAS-benutzer Konferenz

DISN
: Defense Information Systems Network mil., USA

DISNET
: Defense Integrated Secure Network Netzwerk, mil., USA, Vorlaeufer, DSNET

DISNNT
: Defense Information Systems Network - Near Term DISN, mil., USA, "DISN-NT"

DISOSS

DIStributed Office Support System (IBM, MVS)

DISP
: Directory Information Shadowing Protocol

DISP
: Draft International Standardized Profiles (ISO)

DISP
: Dutch Independent Shareware Programmer Org., Netherlands, "D.I.S.P."

DISSP
: Defense Information System Security Program mil., USA

DIT
: Directory Information Tree (X.500)

DITCO
: Defense Information Technology Contracting Office Org., mil., DISA, USA

DIU
: Digital Interface Unit

DIVE
: Direct Interface Video Extensions (IBM, MMPM/2)

DIVX
: DIgital Video eXpress (DVD), "Divx"

DIX
: DEC, Intel, Xerox ethernet, DEC, Intel, Xerox

DKE
: Deutsche Elektrotechnische Kommission org., DIN, VDE

DKRZ
: Deutsches KlimaRechenZentrum org.

DL
: Distribution List

DLC
: Data Link Control

DLCI
: Data Link Connection Identifier (ATM)

DLD
: Deutsche Linux Distribution (Linux)

DLE
: Data Link Escape

DLG
: Digital Line Graph

DLKL
: ??? (DB)

DLL
: Dynamic Link Library

DLL
: Dynamic Link Loader (BS2000)

DLP
: Digital Light Processing (TI)

DLP
: Discrete Logarithmic Problem

DLPI
: Data Link Provider Interaface (X/Open)

DLR
: Dynamic Link Routine

DLS
: DOS LAN Services (IBM, LAN Server)

DLSW
: Data Link SWitching (APPN, MPTN, RFC 1795, SNA, NETBIOS), "DLSw"

DLT
: Digital Line Tape (DEC)

DM
: Delta Modulation

DM
: Disconnect Mode (LAPB)

DMA
: Direct Memory Access

DMAC
: Direct Memory Access Controller

DMB
: Digital Multimedia Broadcasting (Telekom, Blaupunkt, DAB)

DMC
: Desktop Multimedia Conferencing

DMD
: Device Manager Driver (OS/2)

DMD
: Differential Mode Delay Gigabit-, ethernet

DMD

Digital Micromirror Device (IC, DLP, TI)

DMD

Directory Management DOMAIN (OSI, DS)

DMDAC

Dual MAC Dual Attached Concentrator (FDDI, DAC)

DMDD

Distributed Multiplexing Distributed Demultiplexing

DMDF

Distributed Management Data Facility (DCE, DME)

DME

Distributed Management Environment (OSF)

DMF

Digest Message Format (Internet, RFC 1153)

DMF

Distribution Media Format [diskette] (FDD)

DMI

Definition of Management Information (OSI)

DMI

Desktop Management Interface (DTMF)

DMI

Digital Multiplexed Interface

DML

Data Manipulation Language

DML

Data Manipulation Logic

DMOS

Diffusion Metal Oxide Semiconductor (IC)

DMP

Dot Matrix Printer

DMS

Data Management System

DMS

Defense Message System mil., USA

DMS

Digital Multiplexed System

DMS

Distributed Media Services (COSE)

DMS
: Document Management System

DMSCMS
: Display Management System/Conversional Monitor System, "DMS/CMS"

DMSIG
: Defense Message System Implementation Group Org., DMS, mil., USA

DMSP
: Distributed Mail System Protocol (Internet)

DMSTWG
: Defense Message System Transition Working Group Org., DMS, mil., USA

DMT
: Discrete Monitor Timings (VESA)

DMT
: Discrete Multitone Technology (ADSL, Amati Communications, ANSI)

DMTF
: Desktop Management Task Force

DMU
: Data Manipulation Unit

DMV
: Daten- und MedienVerlag

DMV
: Deutsche Mathematiker-Vereinigung org.

DN
: Distinguished Name (X.500)

DN
: Distribution Network

DNA
: Digital Network Architecture (DEC)

DNA
: Direct Network Attach (Xyratex, RAID)

DNA
: Distributed Network Architecture (NCR)

DNAE
: DatenNet-AnschlussEinrichtung (Telekom)

DNAT
: Dynamic Network Address Translation

DNC

Direct Numerical Control (CNC)

DNC

Dynamic Network Controller

DNCMPE

Direct Numerical Control / ??? (CNC), "DNC/MPE"

DNCP

[PPP] DECNet phase iv Control Protocol (RFC 1762, DECNET, PPP)

DNCRI

Division of Networking and Communication Research and Infrastructure

DNHR

Dynamic Non Hierarchical Routing

DNI

DECnet Network Interface

DNIC

Data Network Identification Code (X.121)

DNR

Digital Noise Reduction

DNS

DOMAIN Name System (Internet)

DNSO

Defense Network Systems Organization Org., USA, mil.

DNSTAN

Digitale NebenSTellenANlagen (Telekom), "DNStAn"

DNX

Departmental Network eXchange [bridging router] (SNA, SDLC, Proteon)

DO

Distributed Objects (NeXT)

DOAM

Distributed Office Applications Model (ISO, IEC, DIS 10031-1 f.)

DOCC

DISA Operations Control Complex DISA, mil., USA

DODISS

Department Of Defense Index of Specifications and Standards mil., USA

DOE

Distributed Objects Everywhere (Sun)

DOMAIN

Distributed Operating Multi Access Interactive Network (Apollo, Internet)

DOMF
: Distributed Object Management Facility (Sun)

DOOM
: Decentralised Object Orientated Machine

DOP
: Directory Operational binding management Protocol

DOS
: Disk Operating System (OS)

DOSS
: Dedicated Office Systems and Services

DOV
: Data Over Voice [MODEM]

DOW
: Direct OverWrite (MO, ...)

DP
: Data Processing

DP
: Detection Point (IN)

DP
: Draft Proposal (ISO)

DPB
: Drive Parameter Block (DOS, BIOS, FDD, HDD)

DPC
: Database Promotion Center org., Japan, DB

DPE
: Distributed Processing Environment (IN)

DPG
: ??? org.

DPI
: [SNMP] Distributed Program Interface (SNMP, RFC 1228)

DPI
: [SNMP] Distributed Protocol Interface (SNMP, RFC 1592)

DPI
: Data Processing Installation

DPI
: Dots Per Inch

DPL

Descriptor Privilege Level (OS/2, NT)

DPM

Defects Per Million

DPMA

Data Processing Management Association org.

DPMA

Demand Priority Access Method

DPMI

DOS Protected Mode Interface (DOS, MS, Intel)

DPMS

Display Power Management Signalling [standard] (VESA)

DPMS

DOS Protected Mode Services (Novell, DOS)

DPN

Data Packet Network (Nortel)

DPN

Data Processing Node (MODEM)

DPN

Deutsches Provider Network (ISP)

DPNPH

Data Packet Network-Packet Handler, "DPN-PH"

DPP

Distributed Parallel Processing

DPQ

Data Processing Quality

DPRL

Digital Property Rights Language (Xerox)

DPS

Display PostScript (NeXT, GUI)

DPSK

Differential Phase Shift Keying (DFUE)

DPT

Distributed Processing Technology manufacturer

DPT

Drive Parameter Table

DPV

DatenPaketVermittlung [srechner]

DQDB
: Distributed Queue Dual Bus (ISO, IEC, IS 8802/6, IEEE 802.6)

DQL
: Database Query Language (DB)

DR
: Developer Release (Linux)

DR
: Digital Research manufacturer

DRAM
: Dynamic Random Access Memory (RAM, IC)

DRB
: DRAM Row Boundary [register] (DRAM, PCI)

DRC
: Design Rule Checks (CAD)

DRD
: Data Reading Device

DRDA
: Distributed Relational Database Architecture (IBM, DB)

DRDAAS
: Distributed Relational Database Architecture Application Server (IBM, DB), "DRDA AS"

DRDOS
: Digital Research Disk Operating System (DR, OS), "DR-DOS"

DREN
: Defense Research and Engineering Network network

DRG
: Developer Relations Group (MS)

DRI
: Defense Research Internet ARPANET, successorr, Netzwerk

DRM
: Destination Release Mechanism (DQDB)

DRMU
: Digital Remote Measurement Unit

DRP
: DECnet Routing Protocol (DEC)

DRSN
: Defense Red Switch Network [hopefully no traffic ever] Netzwerk, mil. USA

DS

Data Segment [register] CPU, Intel, assembler

DS

Deutschsprachige Shareware org.

DS

Digital Services [level]

DS

Directory Service (OSI, ISO, DP 9594)

DS

Distribution Services (SNA)

DS

Double Sided [disks] (FDD)

DS0

Digital Signal level 0 (ISDN, T1), "DS-0"

DS1

Digital Signal level 1 (ISDN, T1), "DS-1"

DS1E1

??? (RFC 1406), "DS1/E1"

DS2

Digital Signal level 2, "DS-2"

DS3

Digital Signal level 3 (T3), "DS-3"

DS3

Digital Signal level 3, "DS-3"

DS3E3

??? (RFC 1407), "DS3/E3"

DSA

Data Service Adapter

DSA

Digital Signature Algorithm cryptography, NIST

DSA

Digital Storage Architecture

DSA

Directory System Agent (X.500, DSA)

DSA

Distributed Systems Architecture (Bull)

DSA

Dynamic Scalable Architecture (DB, Informix)

DSAP
: Destination link Service Access Point (SAP, LLC)

DSAT
: Deep Shit Alert Table (Apple, MACOS)

DSBAM
: Double-SideBand Amplitude Module

DSC
: Document Structuring Conventions (Adobe)

DSD
: Data Structure Diagram (CASE)

DSDC
: Data Segment Descriptor Cache [register] (DS, Intel, CPU)

DSDL
: Data Storage Definition ??? Language (DB)

DSE
: Data Switching Equipment (X.25, CCITT)

DSE
: Distributed Systems Environment (Honeywell, Bull)

DSEA
: DataStation Emulation Adapter (IBM, AS/400, ...)

DSEE
: Distributed Software Engineering Environment ??? (Apollo, CM)

DSH
: Desperately Seeking Help slang, Usenet

DSI
: Defense Simulation Internet Netzwerk, mil., USA

DSI
: Dial Services Interface [API] (API, IBM)

DSI
: Digital Speech Interpolation (VOFR)

DSI
: Dynamic Skeleton Interface (CORBA, ORB, OA)

DSID
: Destination Signaling IDentifier

DSIMM
: Dual [RAS] Single Inline Memory Module (IC), "D-SIMM"

DSL

Dialogue Scripting Language (DCE, UIL)

DSL

Digital Subscriber Line

DSL

Digital system Specification Language (HDL)

DSL

Distributed Service Logic (IN)

DSLCP

Dynamically Switched Link Control Protocol (RFC 1307)

DSMN

Directory Service Manager for Netware (MS, Windows NT, FPNW)

DSN

Defense Switched Network mil., USA

DSN

Developer Support News (IBM, OS/2)

DSN

Distributed Systems Network (HP)

DSNET

Defense Secure NETwork mil., USA

DSOM

Distributed System Object Model (IBM)

DSP

Digital Signal Processing / Processor

DSP

Directory System Protocol (X.500, DS)

DSP

Document Services for Printing (Xerox), "DS/P"

DSP

DOMAIN Specific Part (NSAP, IDL)

DSR

Data Set Ready (MODEM, RS-232)

DSR

Device Status Report

DSRI

Digital Standard Relational Interface

DSRS

Defense Software Repository System mil., USA

TOC

DSS
: Decision Support System (IM)

DSS
: Defense Switched Services mil., USA

DSS
: Digital Signature Standard NIST, cryptography

DSS
: Directory and Security Services (DCE, IBM, LAN)

DSS
: Distributed Security Service (DCE)

DSS2
: [setup] Digital Subscriber Signaling #2

DSSCS
: Defense Special Security Communications System mil., USA

DSSI
: Digital Storage Systems Interconnect (VAX)

DSSS
: Direct Sequence Spread Spectrum (Wireless LAN)

DSSSL
: [standard] Document Style Semantics and Specification Language (DTD, ISO, IEC, DIS 10179)

DST
: Daylight Saving Time (TZ)

DSTN
: Double SuperTwisted Nematic (LCD)

DSU
: Digital Service Unit (ATM)

DSU
: Distribution Service Unit (IBM, SNADS)

DT
: Display Terminal

DTA
: Direct Tape Access (Seagate)

DTA
: Disk Transfer Area (DOS)

DTAM
: Document Transfer, Access and Manipulation (CCITT, T.400, ODIF)

DTAP

Direct Transfer Application Part (MS, MM, MTP)

DTC

DeskTop Conferencing

DTC

Distributed Transaction Coordinator (MS, SQL Server, DB)

DTD

Document Type Definition (SGML)

DTE

Data Terminal Equipment (X.25, CCITT)

DTE

DatenTransferEinrichtung

DTH

DialogTestHilfe (BS2000)

DTLB

Dual Translation Lookaside Buffer (CPU)

DTMF

DeskTop Management task Force (Intel)

DTMF

Dual Tone Multi Frequency

DTMP

DCPS Management Panel (DCPS)

DTMS

Document Transfer and Manipulation Services (CCITT, T.400)

DTP

DeskTop Publishing

DTP

Distributed Transaction Processing (X/Open, OLTP)

DTP

Document Transfer Profile SPAC, ODA, DAP, predecessor

DTR

Data Terminal Ready (MODEM, RS-232)

DTR

DeskTop Reproduction (DTP)

DTR

Document Filing and Retrieval (DOAM, ISO, IEC, DIS 10166-1 f.)

DTR

Draft Technical Report

DTS
: Direct To SOM

DTS
: Distributed Time Server

DTS
: Distributed Time Service (DCE)

DTU
: Demand Transmission Unit

DU
: DatenUEbertragungsEinrichtung

DU
: Disk Used (Unix)

DUA
: Directory User Agent (X.500, DS)

DUEE
: DatenUEbertragungsEinrichtung (Telekom)

DUOW
: Distributed Unit Of Work (DRDA, IBM), "DUoW"

DV
: DatenVerarbeitung

DV
: Digital Video

DVA
: DatenVerarbeitungsAnlage

DVB
: Digital Video Broadcasting Europe

DVD
: Deutsche Vereinigung fuer Datenschutz [e.V.] org.

DVD
: Digital Versatile Disk (CD, MPEG)

DVD
: Digital Video Disk [veraltet/old term]

DVDR
: Digital Versatile Disk Recodable (DVD), "DVD-R"

DVDRAM
: Digital Versatile Disk Random Access Memory (DVD), "DVD-RAM"

DVDROM

Digital Versatile Disk Read Only Memory (DVD), "DVD-ROM"

DVDRW

Digital Versatile Disk + RWritable (PC-RW, DVD, Sony, Philips, HP, Mitsubishi, Ricoh, Yamaha), "DVD+RW"

DVI

DeVice Independent

DVI

Digital Video Interactive (Intel, IBM, Lotus)

DVL

Digital Video Link

DVMA

Direct Virtual Memory Access

DVMRP

Distance Vector Multicast Routing Protocol (IP)

DVR

??? (DTP, Truevision)

DVS

DatenVerwaltungsSystem (BS2000)

DVS

Digital Video Systems manufacturer

DVST

DatenVermittlungsSTelle (Telekom)

DVSTP

DatenVermittlungsSTelle mit Packetvermittlung (Telekom), "DVST-P"

DVT

Deutscher Verband Technisch-wissenschaftlicher vereine org.

DVX

Digital Voice eXchange

DWANGO

Dialup Wide-Area Network Game Organization (IVS Corporation)

DWF

Drawing Web Format (AutoCAD)

DWH

DISA Western Hemisphere DISA, mil., USA

DWM

Diskless Workstation Management (AIX, IBM)

DWMT

Discrete Wavelet MultiTone [modulation]

DWT

Discrete Wavelet Transformation

DXA

Directory eXchange Agent

DXI

Data eXchange Interface

DXJ

Datex J (Telekom), "DxJ"

DXJVST

Datex J VermittlungsSTelle (Telekom), "DxJ VSt

DXS

Directory eXchange Server

Go to the previous, next section.

Go to the previous, next section.

-- E ---

E3
: End-to-End Encryption cryptography

EA
: Eingabe/Ausgabe, "E/A"

EA
: Escrowed Authenticator cryptography, EES

EA
: Extended Attribute (OS/2)

EADAS
: Engineering and Administrative Data Acquisition System

EADASNM
: EADAS/Network Management, "EADAS/NM"

EADF
: Elliptical Aperture with Dynamic Focus

EAI
: External Authoring Interface (VRML)

EAM
: Evanescent Access Method (BS2000)

EAN
: European Article Numbering [system]

EANTC
: European Advanced Networking Test Center Org., Berlin, Germany, ANTC, FDDI

EARN
: European Academic Research Network network

EAROM
: Electrically Alterable Read Only Memory

EAS
: Enterprise Access System (Dynatech)

EASE
: Easy Access System Europe (Novell, FTP)

EASI
: Enhanced Asynchronous SCSI Interface

EATA
: Enhanced AT Bus Attachment

EATCS
: European Association for Theoretical Computer Science Org., Europe

EAZ
: EndgeraeteAuswahlZiffer (ISDN)

EB
: Electronic Banking banking

EBAM
: Electronic Beam-Addressable Memory (IC)

EBAS
: Elektronisches teile-BestellAbwicklungsSystem (MBAG)

EBC
: EISA Bus Controller (Wyse)

EBCDIC
: Extended Binary-Coded Decimal Interchange Code

EBNF
: Extended Backus-Naur-Form

EBR
: Enterprise Backup and Restore (ENS, Banyan)

EBROM
: Electronic Book - Read Only Memory, "EB-ROM"

EBU
: European Broadcasting Union Org., Europe

EBUS
: Elektronisches teile-BUchungsSystem (MBAG)

EBV
: Elektronische BildVerarbeitung

EC
: Electronic Commerce

EC
: Error Correction (MODEM)

ECAI
: European Conference on Artificial Intelligence conference, AI, ECCAI, Europe

ECB
: Electronic CodeBook [mode] cryptography, DES

ECB

Event Control Block (IPX)

ECC

Electrical Connectivity Checks (CAD)

ECC

Elliptic Curve Cryptosystem Certicom, cryptography

ECC

Error Checking and Correction

ECC

Error Correction Circuit (CPU, POWER)

ECC

Error Correction Code (CD)

ECCAI

European Coordinating Committee for Artificial Intelligence Org., AI, Europe

ECCRAM

Error Checking and Correction ??? Random Access Memory (RAM)

ECEDI

Electronic Commerce/Electronic Data Interchange (EC, EDI), "EC/EDI"

ECF

Enhanced Connectivity Facilities (IBM)

ECHO

European Community Host Organisation Org., Europe

ECHT

European Conference on Hypermedia Technology INRIA, conference

ECI

Efficient Channel Integration (ADC, EDI)

ECL

EClectic Language (Harvard, TOPS)

ECL

Emitter Coupled Logic

ECM

Entity Coordination Management (FDDI, SMT)

ECM

Error Correction / Correcting Mode (FAX, HDLC)

ECMA

European Computer Manufacturers Association Org., Europe

ECN

European Counter Network

ECOC
: European Conference on Optical Communications conference

ECOM
: Electronic Computer Originated Mail, "E-COM"

ECOOP
: European Conference on Object Orientated Programming OOP, conference

ECP
: [PPP] Encryption Control Protocol (PPP, RFC 1968)

ECP
: Enhanced Capability Port (MS)

ECP
: Enhanced Communication Port / Protocol

ECP
: Exessive CrossPosting Usenet, EMP, spam

ECPA
: Electronic Communications Privacy Act (USA)

ECRC
: European Computer industry Research Centre [gmbh] ISP, Bull, SNI, ICL, org.

ECS
: Elitegroup Computer Systems manufacturer, Taiwan

ECS
: Enhanced Chip Set (Amiga, Commodore)

ECSA
: Exchange Carriers Standards Association org.

ECSC
: European Customer Support Centre (HP)

ECTS
: European Computer Trade Show fair, London

ECU
: EISA Configuration Utility (EISA)

ED
: End Delimiter (FDDI, Token Ring)

ED
: Enhanced Density

EDA
: Electronic Design Automation

EDAC

Electromechanical Digital Adapter Circuit

EDAC

European Conference on Design Automation IEEE-CS, conference

EDBS

Einheitliche DatenBank-Schnittstelle (DB)

EDC

Error Detection Code (CD)

EDD

Enterprise Data Distribution (ENS, Banyan)

EDGAR

Electronic Data Gathering, Analysis, and Retrieval [system] (DB, Internet)

EDI

Electronic Data Interchange (GOSIP)

EDID

Extended Display Identification Data [standard] (VESA, DDC)

EDIF

Electronic Design Interchange Format

EDIFACT

Electronic Data Interchange For Administration, Commerce and Transport

EDLC

Ethernet Data Link Control ethernet

EDM

Engineering Data Management

EDM

Extended Data Message

EDMCC

European Distributed Memory Computing Conference GI, ITG, IFIP, conference

EDMS

Engineering Document Management System

EDO

Extended Data Out [ram] (RAM, DRAM, IC)

EDODRAM

Extended Data Out Dynamic Random Access Memory (RAM), "EDO-DRAM"

EDORAM

Extended Data Out Random Access Memory (RAM, IC), "EDO-RAM"

EDOS

Elektronisches DateiOrganisationsSystem (MBAG)

EDP
: Electronic Data Processing

EDP
: Electronic Data Processing

EDP
: Enhanced Dot Pitch (Hitachi)

EDPS
: Electronic Data Processing System

EDR
: External Developer Release

EDRAM
: Enhanced Dynamic Random Access Memory (RAM, DRAM, IC)

EDS
: Electronic Data Systems [corporation] provider, USA

EDSAC
: Electronic Delay Storage Automatic Calculator

EDSRA
: Earth Data System Reference Application (ISH, USA)

EDT
: Eastern Daylight Time [-0400] (TZ, EST, USA)

EDV
: Elektronische DatenVerarbeitung

EDVAC
: Electronic Discrete Variable Automatic Computer

EEI
: Equipment to Equipment Interface

EEI
: External Environment Interface mil., USA

EELS
: Engineering Electronic Library, Sweden org., WWW, Schweden

EEM
: External Expansion Module (Sun)

EEMS
: Enhanced Expanded Memory Specification

EEP
: Early Experience Program (Borland)

EEP

Entry Exit Procedure (R:Base, DB)

EEPROM

Electronical Erasable Programmable Read Only MemorySerial Presence Detect

EES

Escrowed Encryption Standard cryptography, NSA

EESC

European EDIF Steering Committee Org., EDIF, Europe

EET

Eastern European Time [+0200] (TZ)

EET

Edge Enhancement Technology (Seikosha, Itoh)

EFAKS

Elektronisches FAKturierungs- und abrechnungsSystem (MBAG)

EFCI

Explicit Forward Congestion Indication (ATM)

EFF

Electronic Frontier Foundation Internet, org.

EFI

Electronics For Imaging manufacturer

EFL

Emitter Follower Logic (IC)

EFM

Eight-to-Fourteen-Modulation (CD)

EFSM

Extended Finite State Machine (TTCN, ...)

EFT

Electronic Funds Transfer

EFT

Euro-FileTransfer (ISDN, ETS 300 075)

EFTPOS

Electronic Funds Transfer at the Point-Of-Sale EFT, banking, "EFT-POS"

EFTS

Electronic Funds Transfer System

EFUE

EmmissionsdatenFernUEbertragung (DFUE)

EG

Evil Grin slang, Usenet, IRC

EGA
: Enhanced Graphics Adapter predecessor, VGA

EGB
: Elektrostatisch gefaehrdete Bauelemente

EGP
: Exterior Gateway Protocol (RFC 904)

EGPA
: Erlangen General Purpose Array (MP)

EGREP
: Extended Global Regular Expression Print (Unix, GREP)

EGS
: Enhanced Graphics System (Commodore)

EHF
: Encoding Header Field (Internet, RFC 1154)

EHKP
: Einheitliche Hoehere KommunikationsProtokolle (BTX, ER, Telekom)

EHLLAPI
: Emulator High Level Language API (IBM, 3270, API)

EHS
: European Home Systems [concept]

EHSA
: European Home Systems Association Org., Europe

EIA
: Electronics Industry Association USA, org.

EIB
: European Installation Bus

EIBA
: European Installation Bus Association Org., Europe

EICAVR
: European Institute for Computer Anti-Virus Research Org., Europe

EIDE
: Enhanced Integrated Drive Electronics (HDD, IDE), "E-IDE"

EINE
: EINE Is Not EMACS (EMACS, LISP)

EIP
: Extended Internet Protocol (Internet, RFC 1385)

EIS

European Information System Europe

EIS

Executive Information System (IM)

EISA

Enhanced Industry Standard Architecture (ISA)

EISA

European Imaging and Sound Association Org., Europe

EISB

Electronic Imaging Standards Board org.

EISS

Europaeisches Institut fuer SystemSicherheit Org., Karlsruhe, Europe

EIT

Encoded Information Type

EJC

Electronic Journal of Communication

EKMS

Electronic Key Management System cryptography

EKOS

Elektronisches KOmmunikationsSystem (MBAG)

EL

Electro Luminiscent [display]

EL1

Extensible Language one (ECLogic)

ELAN

Education LANguage

ELAN

Emulated Local Area Network (ATM, LANE)

ELAP

Ethernet Link Access Protocol LAP, ethernet

ELF

Executable and Linkable Format (Unix, OS/2)

ELH

Entity Life History (DB)

ELI

Embedded LISP Interpreter (Andrew mail system)

ELK

Extension Language Kit (Scheme)

ELLIS
: EuLisp LInda System (LISP)

ELM
: ELectronic Mailer (Unix)

ELO
: Elektronischer Leitz Ordner (OA)

ELOD
: Erasable Laser Optical Disk (OD)

ELP
: Equational Logic Programming [language]

ELRAD
: [magazin fuer] ELektronik und technische RechnerAnwenDungen

ELS
: ??? (Novell, Netware)

EM
: Extensions Manager (Apple)

EMA
: Electronic Messaging Association org., USA

EMA
: Enterprise Management Architecture (DEC)

EMA
: Extended Mercury Autocode

EMACS
: Editing MACroS (GNU, PD)

EMB
: Enhanced Master Burst (EISA)

EMC
: ElectroMagnetic Compatibility

EMEA
: [IBM] Europe, Middle East, Africa (IBM)

EMI
: ElectroMagnetic Interference

EMI
: External Machine Interface [protocol] (SMS)

EMISA
: EntwicklungsMethoden fuer InformationsSysteme und deren Anwendung org., GI

EML

Element Management Layer (TMN)

EMM

Expanded Memory Manager

EMMA

European MultiMedia Award

EMP

Excessive Multiple Posting Usenet, ECP, spam

EMR

Electro-Magnetic Radiation

EMS

European Mathematical Society org.

EMS

Expanded Memory Specification (DOS, Intel)

EMSC

Electronic Mail Standards Committee org.

EMV

ElektroMagnetische Vertraeglichkeit (EN 55022, DIN, VDE 0878)

EMX

Enterprise Messaging eXchange [switch]

EN

Europa Norm Europe

ENA

Electronic Networking Association org., Internet

ENA

Enterprise Networking Association Banyan, User group, org.

ENA

European Networking Associates Org., Europe

ENA

Extended Network Addressing (IBM, SNA)

ENDC

European Network Design Center (3COM)

ENDIVE

Enhanced Direct Interface Video Extensions (OS/2, MMPM/2, IBM), "EnDIVE"

ENIAC

Electronic Numerical Integrator And Computer

ENP

Embedded NPrinter (NEST, Novell)

ENS
: Enterprise Network Services (Banyan, VINES)

ENSIQ
: ENS - Information Query (Banyan, ENS), "ENS IQ"

ENSMT
: ENS - Management Tool (ENS, Banyan), "ENS-MT"

EO
: Europe Online network

EOA
: End Of Adress

EOB
: End Of Block

EOD
: End Of Discussion slang, Usenet, IRC

EOD
: Erasable Optical Disk (OD)

EOF
: End Of File

EOF
: Enterprise Objects Framework (NeXT)

EOI
: End Of Input

EOI
: End Of Interrupt

EOJ
: End Of Job

EOL
: End Of Line

EOM
: End Of Message

EON
: Enhanced Other Networks (RDS)

EOP
: End Of Procedure (Fax)

EOS
: Electrical OverStress

EOS

Extended Operating System (OS)

EOT

End Of Text

EOT

End Of Transmission

EOUG

European Oracle User Group Org., DB, Oracle, User group, Europe

EP3

[the] Extensible PERL PreProcessor (PERL)

EPA

Enhanced Performance Architecture

EPA

Environmental Protection Agency USA, org.

EPAC

Ein-Platinen-Allzweck-Computer (IC, C'T)

EPDF

Embedded Portable Document Format (Adobe, PDF, HTML)

EPH

Electronic Payment Handling banking

EPHOS

European Procurement Handbook for Open Systems

EPIC

Explicit Parallelism Instruction Computing (Intel)

EPIM

Ethernet Port Interface Module ethernet

EPIX

Enhanced Performance unIX (Unix), "EP/IX"

EPLD

Erasable Programmable Logic Device

EPN

Electronic-highway Platform Netherlands ISP, Netherlands

EPOC

??? (Psion, OS, PDA)

EPP

Education Purchase Plan (Adobe)

EPP

Enhanced Parallel Port / Protocol (Intel, Zenith, Xircom, IEEE 1284)

EPROM
: Erasable Programmable Read Only Memory

EPS
: Electronic Publishing Systems

EPS
: Embedded PServer (NEST, Novell)

EPSA
: Early Page Space Allocation (AIX, IBM)

EPSI
: Encapsulated PostScript Interchange

EPSIG
: Electronic Publishing Special Interest Group org.

ER
: Externer Rechner (T-Online)

ERCIM
: European Research Consortium for Information and Mathematics Org., GMD, INRIA, CWI, RAL, ..., Europe

ERD
: Entity-Relationship Diagram

ERIN
: Environmental Ressources Information Network Australia, Netzwerk

ERM
: Entity Relationship Model (DB)

ERO
: European Radiocommunications Office Org., Europe, CEPT

EROM
: Erasable Read Only Memory

ERS
: Enterprise Ressource Sharing (ENS, Banyan)

ERS
: Error Report Suppression [flag] (CATNIP)

ES
: End System (ATM)

ES
: Errored Seconds (DS1/E1)

ES
: Extended Services (OS/2)

ES
: Extra Segment [register] CPU, Intel, assembler

ES1
: EliteSwitch/1 (SMC, FDDI), "ES/1"

ESA
: Enterprise Systems Architecture (IBM, MVS/XA)

ESANN
: European Symposium on Artificial Neural Networks conference

ESCD
: Extended System Configuration Data (BIOS, PNP)

ESCON
: Enterprise Systems CONnect (IBM)

ESCP2
: Epson Specific Code / Protocol 2 (Drucker), "ESC/P2"

ESD
: Electronic Software Distribution

ESD
: ElectroStatic Discharge

ESDC
: Extra Segment Descriptor Cache [register] (ES, Intel, CPU)

ESDI
: Enhanced Standard Device Interface

ESDS
: Entry Sequenced Data Set (VSAM)

ESEC
: European Software Engineering Conference GI, BCS, AFCET, AICA, OEGI, SI, conference

ESER
: Einheitliches System dEr Rechentechnik GDR

ESF
: Extended Super Frame (ISDN, T1)

ESI
: End System IDentifier (ATM)

ESIOP
: Environment Specific Inter-ORB Protocol (OMG, CORBA, ORB, DCE)

ESIS
: Element Structure Information Set (SGML)

ESIS
: End System to Intermediate System [protocol / routing] (OSI), "ES-IS"

ESM
: External Storage Module (Sun)

ESMD
: Embedded Storage Module Disk

ESMD
: Enhanced Storage Module Device

ESMTP
: Extended Simple Message Transport Protocol (SMTP, RFC 1425/1869)

ESN
: Electronic Serial Number

ESN
: Electronic Switched Network

ESNET
: Energy Sciences NETwork network, USA, Internet

ESORICS
: European Symposium On Research In Computer Security conference

ESP
: [IP] Encapsulating Security Payload (IPSEC, IP, RFC 1827)

ESP
: Emulation Sensing Processor (QMS)

ESP
: Ethernet Serial Port ethernet

ESPCM
: Electronic Speech systems Pulse Code Modulation (PCM)

ESPRIT
: European Strategic Program of Research and Development in Information Technology Europe

ESR
: Event Service Routine (IPX)

ESSI
: European Software and Systems Initiative

EST
: Eastern Standard Time [-0500] (TZ, EDT, USA)

ESTO
: Electronic Systems Technology Office org., ARPA

ETAI

Electronic Transactions on Artificial Intelligence (AI)

ETANN

Electrically Trainable Analog Neural Network

ETAS

Elektronisches Teile-AuskunftsSystem (MBAG)

ETB

Elektronisches TelefonBuch (T-Online)

ETB

End of Transmission Blank / Block

ETC

European Test Conference VDE, IEEE-CS, conference

ETE

End-To-End

ETG

Enterprise Transaction Gateway (EDI, NT)

ETH

Eidgenoessische Technische Hochschule

ETHZ

Eidgenoessische Technische Hochschule Zuerich org., Schweiz

ETLA

Extended Three Letter Acronym

ETO

Essentials - Tools - Objects (Apple), "E.T.O."

ETR

Early Token Release (Token Ring)

ETS

Embedded ToolSuite (Phar Lap)

ETS

European Telecommunication Standard (ETSI)

ETS

Executable Test Suite (ISO 9646-1)

ETSI

European Telecommunication Standards Institute Org., Europe

ETX

End of Transmission teXt

EU

Execution Unit (CPU)

EUC
: Extended Unix Code (Unix)

EUNET
: European Unix NETwork network, Internet, ISP, Unix

EUS
: EntscheidungsUnterstuetzende Systeme

EUSUVP
: EntscheidungsUnterstuetzendes System UmweltVertraeglichkeitsPruefung EuropeS, XPS, Uni Hamburg

EUUG
: European Unix User Group Org., User group, Unix, Europe

EV
: EtagenVerteiler cable, EN 50 173

EVC
: Enhanced Video Connector (VESA)

EVE
: Extended Virtual Environment

EVN
: EinzelVerbindungsNachweis (Telekom)

EWAS
: Elektronisches WareneingangsAbwicklungsSystem (MBAG)

EWIKO
: [verein zur foerderung] Elektronischer WIssenschaftKOmmunikation org.

EWOS
: Euopean Workshop for Open Systems (OSI, ODA)

EWS
: Employee Written Software (IBM)

EWS
: European Workshop on SGML org., SGML

EXFCB
: EXtended File Control Block

EXTRA
: EXecutionTRace Analyser (IBM, OS/2)

EXUG
: European X Users Group Org., User group, Europe

EXZ
: EXcessive Zeros [error event] (DS1/E1, DS3/E3)

TOC

EZ

EchtZeit fair

Go to the previous, next section.

Go to the previous, next section.

-- F ---

F2C
: FORTRAN to C [converter]

F2F
: Frequency - Double Frequency

FA
: FernmeldeAmt (Telekom)

FAA
: Flow Admission Acknowledge [message] (LFAP)

FABS
: Fast Access Btree Structure

FAC
: Final Assembly Code (IMEI, GSM)

FACCH
: Fast Associated Control CHannel (GSM, DCCH)

FACE
: Framed Access Command Environment (Unix, SVR4)

FADU
: File Access Data Unit (FTAM)

FAG
: FernmeldeAnlagenGesetz

FAIS
: Finnish Artificial Intelligence Society Org., Finland, AI

FAL
: File Access Listener (DEC, DNA)

FAME
: FORMEX Applied to Multilingualism in Europe SGML, Europe

FAMOS
: Floating gate Avalanche injection Metal Oxide Semiconductor (IC)

FANP
: Flow Attribute Notification Protocol (Toshiba, RFC 2129)

FAPI
: Family Application Programmer Interface (DOS, VDM, API)

FAQ
: Frequently Asked Questions slang, Usenet

FAR
: Fixed Alternative Routing (SNI)

FAR
: Flow Admission Request [message] (LFAP)

FARNET
: Federation of American Research NETworks network

FAS
: Flow Admission Service

FASMI
: Fast Analysis of Shared Multidimensional Information (OLAP)

FAST
: First Application System Test

FAST
: Forschungsinstitut fuer Angewandte Software-Technologie [e.v.] org.

FAT
: File Allocation Table (DOS)

FAU
: Flow Admission Update [message] (LFAP)

FAU
: Friedrich-Alexander-Universitaet Org., Erlangen, Germany, Nuernberg

FAW
: Forschungsinstitut fuer Anwendungsorientierte Wissensverarbeitung org., KI, Ulm

FB
: Fiber optic Backbone

FBAS
: FarbBild-AustastSystem / Farb-Bild-Austast-Synchron-signal (Video)

FBL
: Frame Burst error Length (CD)

FBM
: Flexible Buffer Management (QMS)

FC
: Federal Criteria [for information technology security] (NIST, USA)

FC
: Feedback Control

FC

Fibre Channel

FC

Frame Control (FDDI, Token Ring)

FCA

Flow Control Ack (DLSW)

FCAL

Fibre Channel - Arbitrated Loop, "FC-AL"

FCAPS

Fault, Configuration, Accounting, Performance, Security [management areas]

FCB

File Control Block (DOS)

FCBS

File Control BlockS (DOS)

FCC

Federal Communications Commission org., USA

FCC

Forward Carbon Copy (DFUE)

FCCH

Frequency Correction CHannel (GSM)

FCCN

Fundacao para a Computacao Cientifica Nacional org., Portugal

FCFS

First Come, First Served

FCI

Fibre Channel Interface

FCI

Flow Control Indicator (DLSW)

FCI

Forward Cache Identifier (CATNIP)

FCO

Flow Control Operator Bits (DLSW)

FCP

[SCSI-3] Fibre Channel Protocol (SAM)

FCPH

Fibre Channel PHysical and signaling interface (SAM), "FC-PH"

FCR

Flow Change Request [message] (LFAP)

FCS
: Fast Circuit Switching

FCS
: Frame Check Sequence (FDDI, Token Ring)

FCS
: Frame Check Sum (MODEM)

FDA
: FORTRAN Design Aid (FORTRAN)

FDAD
: Functional Data ADministrator

FDC
: Floppy Disk Controller (FDD)

FDCT
: Fast Discrete Cosinus Transformation (DCT)

FDD
: Floppy Disk Drive

FDDI
: Fiber Distributed Data Interface (ANSI, ISO 8314)

FDDITPPMD
: FDDI Twisted Pair-Physical layer, Medium Dependend, "FDDI TP-PMD"

FDE
: Full Duplex Ethernet ethernet

FDES
: Full Duplex EtherSwitch (Kalpana)

FDI
: [fachverband der] Fuehrungskraefte der Druckindustrie und Informationsverarbeitung org.

FDL
: File Definition Language

FDM
: Frequence Division Multiplexing

FDMA
: Frequency Division Multiple Access (Mobile Systems)

FDT
: Formal Description Technique

FDX
: Full DupleX

FE

Forsching und Entwicklung, "F&E"

FE
: Functional Entity (IN)

FEA
: Functional Entity Action (IN, UNI)

FEAL
: Fast Data Encipherment Algorithm DES, cryptography

FEAST
: Fast Data Encyphering Algorithm cryptography

FEBE
: Far End Block Error (SONET)

FEC
: Forward Error Correction (GSM)

FECN
: Forward Explicit Congestion Notification (ATM)

FEFCO
: ???

FEIT
: Fujitsu Enhanced Imaging Technology (Fujitsu)

FEM
: Finite Elemente Methode

FEN
: Free-net Erlangen/Nuernberg

FEP
: Front End Processor

FER
: Forward Error Correction satellite

FERF
: Far End Receive Failure (UNI, ATM, SONET, OAM)

FESI
: Federacion Espanola de Sociedades de Informatica org.

FET
: Feld-Effekt Transistor

FF
: Formular Feed

FFAPI
: File Format API (MS, API)

FFC
: Fully Formed Character [printer]

FFDC
: First Failure Data Capture

FFDT
: FDDI Full Duplexing Technology (FDDI)

FFOL
: FDDI Follow-On-LAN

FFS
: Fallback Fault-tolerant Server (IBM, OS/2)

FFS
: Fast File System (Amiga, Commodore)

FFS
: Fast Filing System (BSD, Unix)

FFS
: Flexible Fertigungs-Systeme

FFST2
: First Failure Support Technology /2 (IBM), "FFST/2"

FFT
: Final Form Text

FGA
: Future Graphics Adapter (Spea)

FGCS
: Fifth Generation Computer Systems [project] (ICOT)

FGREP
: Fixed Global Regular Expression Print (Unix, GREP)

FHGS
: FreeHand Graphics Studio (Macromedia, DTP)

FHSS
: Frequency Hopping Spread Spectrum (Wireless LAN)

FIA
: Fiberoptic Industry Association org.

FIBU
: FInanzBUchhaltung

FIC
: First International Computer [inc.] manufacturer

FID

File Identifier Descriptor (UDF, CD-R)

FIF

Fractal Interchange Format

FIFF

Forum Informatikerinnen Fuer den Frieden org.

FIFO

First In First Out (CPU)

FIG

Forth Interest Group org., Forth

FIGLET

Frank, Ian and Glenn's LETters (ASCII, fonts)

FILO

First In Last Out

FIMAS

Financial Institution Message Authentication Standard banking, ANSI

FIOC

Frame Input/Output Controller

FIP

Facility Interface Processor

FIP

Factory Instrumentation Protocol

FIP

Fluorescent Indicator Panel

FIPA

Foundation for Intelligent Physical Agents org., Agents

FIPS

Federal Information Processing Standard (NIST, USA)

FIRE

Flexible Intelligent Routing Engine (3Com)

FIRMR

Federal Information Resources Management Regulations (USA)

FIRP

Federal Internet Requirements Panel (Internet, USA)

FIRST

ForschungsInstitut fuer Rechnerarchitektur und SoftwareTechnik Org., GMD, Berlin, Germany

FIRST

Forum of Incident Response and SecuriTy org., NIST

FIS
: FachInformationsSystem

FIS
: FuehrungsInformationsSysteme

FIT
: Failures In Time

FITS
: Functional Interpolating Transformation System

FIU
: Fingerprint Identification Unit (Sony)

FKS
: Fernmelde-Klein-Steckverbindung (Westernstecker)

FL
: Fiber optic Link

FLACC
: Full Level Algol Checkout Compiler

FLC
: Ferroelectric Liquid Crystal

FLEA
: Four Letter Extended Acronym

FLEXIP
: ??? (Novell, Netware), "FLeX/IP"

FLIM
: Faithful Library about Internet Message (EMACS, GNU)

FLOPS
: FLoating point Operations Per Second (CPU)

FLP
: Fast Link Pulse ethernet, LAN, NLP

FLS
: ???

FM
: Frequenz-Modulation

FMM
: Flash Memory Manager (Intel)

FMS
: FernMeldeSystem

FNC

Federal Networking Council org., USA

FO

Fiber Optic

FOA

Fiber Optic Association org.

FOC

Fiber Optic Cable / Communications

FOCS

[symposium on] Foundations Of Computer Science conference

FOD

Flexible Optical Disk (OD)

FODA

Formal specification of ODA document structures (ODA, ISO)

FODO

[international conference on] Foundations Of Data Organization and algorithms conference, INRIA

FOEBUD

[verein zur] Foerderung des OEffentlichen Bewegten und Unbewegten Datenverkehrs [e.v.] org., "FoeBud"

FOIRL

Fiber Optic InterRepeater Link (OWG)

FOKUS

Forschungszentrum fuer Offene KommUnikationsSysteme Org., GMD, Berlin, Germany

FOLDOC

Free OnLine Dictionary Of Computing (WWW, UK)

FOMAU

Fiber Optic MAU ethernet

FON

Fiber Optics Network

FOOBAR

FTP Operation Over Big Address Records (RFC 1639, FTP)

FOOT

Forum for Object Oriented Technology (CERN, OOP)

FORMEX

FORMalised EXchange of electronic publications SGML, Europe

FORML

Formal Object Role Modelling Language

FORTRAN
: FORmula TRANslation

FORTWIHR
: FORschungsverbund fuer Technisch-WIssenschaftliches HochleistungsRechnen Org., Bavaria

FORWISS
: bayerisches FORschungszentrum fuer WISensbasierte Systeme org., KI

FOSI
: Format Output Specification Instance (SGML, CALS)

FOSSIL
: Fido Opus Seadog Standard Interface Layer

FOT
: Fiber Optic Transceiver

FOTE
: Follow-on Operational Test and Evaluation, "FOT & E"

FOURCC
: FOUR Character Code (RIFF, TIFF)

FOX
: Field Operational X.500

FPA
: Floating Point Accelerator

FPC
: Floating Point Coprocessor

FPDI
: Flat Panel Display Interface (VESA)

FPDU
: FTAM Protocol Data Unit (PDU)

FPE
: Floating Point Engine

FPGA
: Field Programmable Gate Array

FPI
: Flux Changes per Inch (HDD)

FPI
: Functional Process Improvement

FPLA
: Field Programmable Logic Array

FPLMTS

Future Public Land Mobile Telecommunications System (IN, Mobile Systems)

FPM

Fast Page Mode [DRAM] (RAM, DRAM)

FPM

Fast Page Mode [ram] (RAM, DRAM, IC)

FPMDRAM

Fasr Page Mode DRAM (RAM, DRAM, IC), "FPM-DRAM"

FPNW

File and Print service for NetWare (MS, Windows NT, DSMN)

FPODA

Fixed Priority Orientated Demand Assignment (MAC, PODA)

FPP

Fast-Parallel-Port

FPP

Floating Point Processor

FPP

FORTRAN Pre-Processor (FORTRAN)

FPR

Floating Point Register (FPU)

FPS

Fast Packet Switching (X.25, Datex-J)

FPS

Frames Per Second

FPU

Floating Point Unit (CPU)

FQDN

Fully Qualified DOMAIN Name (Internet)

FRAD

Frame Relay Access Device

FRAM

Ferroelectric Random Access Memory (RAM, IC)

FRD

Functional Requirements Description

FRF

Floatingpoint Register File (DEC, Alpha, CPU)

FRICC

Federal Research Internet Coordinating Committee org.

FRL
: Frame Representation Language (AI)

FRMR
: FRaMe Reject (HDLC, LAPB, SDLC)

FROM
: Factory Read Only Memory (ROM)

FRR
: Functional Recovery Routine

FRS
: Flexible Route Selection

FRS
: Frame Relay Service (ATM, UNI)

FS
: File System (LVM)

FS
: Frame Status (FDDI, Token Ring)

FSAG
: Free Software Association of Germany org.

FSB
: For Small Business (Novell)

FSCK
: File System Consistency checK (Unix)

FSF
: Free Software Foundation org., GNU

FSK
: Frequency Shift Keying (DFUE)

FSM
: Finite State Machine (TTCN, ...)

FSMGDOS
: ??? Graphics Device Operating System (OS, Atari)

FSP
: File Slurping Protocol

FSS
: Fast System Switch (Unix)

FST
: Flat Square Technology (Hitachi)

FSTN

Film SuperTwisted Nematic (LCD)

FSTP
: Foiled Shielded Twisted Pair [cable] (STP, TP), "F/STP"

FSU
: Free Software Union org.

FT1
: Fractional T1 (ISDN, T1, DS-0, DS-1)

FTA
: Floptical Technology Association (3M u.a.)

FTAM
: File Transfer, Access and Management (GOSIP, ISO 8571/8572)

FTC
: Federal Trade Commission org., USA

FTE
: [DCOM] For The Enterprise (DCOM, SAG)

FTN
: Fido Technology Network (FidoNet)

FTOS
: File Transfer Open Systems

FTOS
: File Transfer OSI Support

FTP
: File Transfer Protocol (Internet, RFC 959)

FTP
: Foiled Twisted Pair [cable] (UTP, TP)

FTPD
: File Transfer Protocol DAEMON (FTP)

FTR
: Full Text Retrieval

FTS
: Fidonet Technical Standard

FTS2000
: Federal Telecommunications Services - 2000 (USA), "FTS-2000"

FTSC
: Federal Telecommunications Standards Committee org., USA

FTSC
: Fidonet Technical Standard Conference conference

FTZ
: FernmeldeTechnische Zentralamt / Zulassung Telekom, ZZF, predecessor

FUA
: Flow Update Acknowledge [message] (LFAP)

FUB
: Freie Universitaet Berlin org.

FUBAB
: Fouled / Fucked Up Beyond All Belief slang, Usenet, IRC

FUBAR
: Fouled / Fucked Up Beyond All Recognition / Repair slang, Usenet, IRC

FUN
: Flow Update Notification [message] (LFAP)

FUNI
: Frame User Network Interface

FUEV
: Fernmeldeanlagen-UEberwachungsVerordnung

FWIW
: For What It's Worth slang, Usenet, IRC

FWT
: Fast Wavelet Transformation

FXU
: FiXed point Unit (POWER, CPU)

FYI
: For Your Information slang, Usenet, IRC

FZA
: ??? [compression]

FZI
: ForschungsZentrum Informatik Org., Uni Karlsruhe, Germany

Go to the previous, next section.

Go to the previous, next section.

TOC

-- G ---

GA
: General Availability (OS/2)

GABRIEL
: GAteway and BRIdge to Europe's national Libraries

GAE
: Generic Application Environment

GAEB
: Gemeinsamer Ausschuss fur Elektronik im Bauwesen org.

GAIA
: GUI Application Interoperability Architecture (OSF)

GAIN
: German Advanced Integrated Network (IBM)

GAK
: Governmental Accessed Keys cryptography

GAL
: Generic Array celL / Logic

GAN
: Global Area Network

GAN
: GrenzAktenNachweis (INPOL)

GAP
: Generic Access Profile DECT, Europe

GAP
: Generic Address Parameter

GART
: Graphics Address Remapping Table (AGP)

GASH
: Group Admin SHell (Unix, Shell)

GATOR
: GATeway OrientierungsRatgeber (Internet)

GAWK
: GNU AWK (GNU, AWK)

TOC

GB
: GigaByte

GBIP
: General Purpose Interface Bus

GBPS
: GigaBits Per Second

GCA
: General Communications Architecture (Ingres)

GCAD
: Geographical Computer Aided Design System

GCATT
: Georgia Center for Advanced Telecommunications Technology org., USA

GCC
: GNU C Compiler (GNU)

GCCS
: Global Command and Control System DISA, mil.

GCID
: Global Call IDentifier

GCIDIE
: Global Call IDentifier- Information Element (GCID), "GCID-IE"

GCL
: GNU Common LISP (GNU, LISP)

GCL
: Graphics Command Language

GCOS
: General Comprehensive Operating System (Honeywell, OS)

GCR
: Gray Component Replacement (RGB, CYMK, DTP)

GCR
: Group Coded Recording

GCRA
: Generic Cell Rate Algorithm (UNI, ATM)

GCS
: Group Control System (IBM, VME)

GDA
: Global Directory Agent (DCE)

GDAP

Government Document Application Profile

GDB

GNU DeBugger (GNU)

GDBM

GNU Data Base Manager (DB, GNU)

GDD

Gesellschaft fuer Datenschutz und Datensicherung org.

GDDM

Graphical Data Display Manager (IBM)

GDI

Graphical Device / Display Interface

GDMI

Generic Definition of Management Information

GDMO

Guidelines for the Definition of Managed Objects (OSI)

GDOS

Graphics Device Operating System (OS, Atari)

GDOUG

Greater Detroit OS/2 User Group org., User group, OS/2, USA

GDS

Generalized Data Stream (IBM, APPC)

GDS

Global Directory Service (DCE)

GDT

Global Descriptor Table

GDTRC

Global Descriptor Table Register Cache (GDT, Intel, CPU)

GECOS

General Electric Comprehensive Operating Supervisor / System OS, GCOS, predecessor

GEI

Gesellschaft fuer Elektronische Informationsverbeitung [mbh] manufacturer, Aachen

GEIST

German Encyclopedic Internet Service Terminal org.

GEM

Graphics Environment Manager (DR)

GENIE

General Electric Network for Information Exchange network

GEOS
: Graphic Environment Operating System

GESIP
: GESellschaft fuer Informatik in der Pharmazie [e.v.] org.

GETS
: Government Emergency Telecommunications System (USA)

GFC
: Generic Flow Control (ATM)

GFC
: Going For Coffee slang, Usenet, IRC

GFLOPS
: Giga FLoating-point Operations Per Second (CPU)

GFP
: Global Functional Plane (IN)

GGP
: Gateway to Gateway Protocol (RFC 823)

GGS
: GueteGemeinschaft Software [e.v.] org.

GHOST
: Goal Hierarchy and Objectives Structuring Technique (TUB)

GHTSTN
: Guest Host Technique SuperTwisted Nematic (LCD), "GHT-STN"

GI
: Gesellschaft fuer Informatik org.

GID
: Group ID

GIDAS
: Grafisch-Interaktives DatenAnalyseSystem

GIGO
: Garbage In Garbage Out

GIL
: Gesellschaft fuer Informatik in der Land-, forst- und ernaehrungswirtschaft [e.v.] org.

GIM
: Gesellschaft fuer informationstechnik und InformationsManagement [mbh] (ISC, Bremen)

GIMP
: General / GNU Image Manipulation Program (Linux, Graphik, GNU)

GINA

TOC

Generische INteraktive Anwendung (GMD)

GIOP

General Inter-ORB Protocol (OMG, CORBA, ORB, IIOP)

GIPS

Giga Instructions Per Second (CPU)

GIRL

Generalized Information Retrieval Language

GIS

Geographic Information System

GIT

GNU Interactive Tools (GNU)

GITS

Government Information Technology Service (USA)

GKDC

Group Key Distribution Centre (KDC, CBT)

GKMP

Group Key Management Protocol

GKS

Graphic Kernel System

GLDV

Gesellschaft fuer Linguistische DatenVerarbeitung org.

GLM

Generalized Markup Language

GLOCOM

[center for] GLObal COMmunications org., Japan

GM

General Midi [standard]

GMA

Gesellschaft fuer Mess- und Automatisierungstechnik org., VDI

GMD

Gesellschaft fuer Mathematik und Datenverarbeitung org., St. Augustin

GMDS

deutsche Gesellschaft fuer Medizinische Dokumentation, informatik und Statistik [e.v.] org.

GME

Gesellschaft MikroElektronik org., VDE, VDI

GMI

Generic Management Information (OSI)

GML
: General Markup Language

GMOEOR
: Gesellschaft ??? org.

GMR
: Giant Magneto-Resistive [heads] (HDD, IBM, Toshiba)

GMSK
: Gaussian Minimum Shift Keying

GMT
: Greenwich Mean Time [+0000] (TZ, UTC, UK)

GNA
: Global Network Academy (Internet)

GNAT
: GNU Ada Translator (GNU)

GNI
: GayNet International network

GNMP
: Government Network Management Profile (USA)

GNN
: Global Network Navigator (Internet, WWW)

GNOME
: GNU Network Object Model Environment (GNU)

GNU
: GNU's Not Unix (rekursiv!, Unix, GNU)

GNX400
: Gateway Network eXchange 400 (Proteon, SNA, SDLC), "GNX 400"

GOCA
: Graphic Object Content Architecture (IBM, MO:DCA)

GOD
: Global OutDial

GOD
: Grundsaetze ordnungsmaessiger Datenverarbeitung, "GoD"

GOSIP
: Government Open Systems Interconnections Profile (USA, UK)

GPC
: General-Purpose Computation

GPC

GNU Pascal Compiler (GNU)

GPC

Graphics Performance Characterization [committee] org., HP, IBM, DEC, SGI, Sun,...

GPCI

Graphics Processor Command Interface

GPF

General Protection Fault (Windows)

GPI

Graphics Programming Interface

GPIB

General-Purpose Interface Bus

GPIO

General Purpose Input Output

GPL

General Public Licence (GNU)

GPL

Graphics Programming Language

GPM

Gesellschaft fuer ProjektManagement [e.v.] org.

GPP

[SCSI-3] Generic Packetized Protocol (SAM)

GPR

General Purpose Register (CPU)

GPRS

General Packet Radio System (ETSI, GSM, TCP/IP)

GPS

Global Positioning System

GPSI

Graphics Processor Software Interface

GPV

General Public Virus

GRADD

GRaphics Adapter Device Driver (OS/2, IBM)

GRASP

Graphic Animation System for Professionals

GRC

Generic Reference Configuration

GRE
: Generic Routing Encapsulation (RFC 1701)

GREP
: Global Regular Expression Print (Unix)

GRIC
: Global Research Internet Connection org., ISP

GRUB
: GRand Unified Bootloader (GNU)

GS
: Gepruefte Sicherheit

GSDS
: Genealogy Software Distribution System

GSL
: Graphics Software Labs (AT&T)

GSM
: Global System for Mobile communications

GSM
: Groupe Special Mobile org. , CEPT

GSMP
: General Switch Management Protocol (DEC, Sprint)

GSNW
: Gateway Service for NetWare (MS, Windows NT)

GSS
: Generic Security Service (IETF, GSS-API)

GSS
: Group Support System

GSSAPI
: Generic Security Service API (RFC 2078, API), "GSS-API"

GST
: Guam Standard Time [+1000] (TZ)

GSX
: Graphics System eXtension

GTDM
: Group Time Division Multiplexing [protocol]

GTF
: Generalized Timing Format (VESA)

GTK

[GNU] GUI ToolKit (GNU, GUI)

GTL
: Gunning Tranceiver Logic+ [bus] (Intel, SMP), "GTL+"

GTLD
: Gerneric Top Level DOMAIN (Internet, IAHC), "gTLD"

GUI
: Graphical User Interface (UI)

GUID
: Globally Unique IDentifier (COM)

GUIDE
: Graphical User Interface Design Editor (Sun)

GUILE
: GNU's Ubiquitous Intelligent Language for Extension (GNU)

GUS
: Guide to the Use of Standards (SPAG)

GUUG
: German Unix User Group org., User group, Unix

GV
: GebaeudeVerteiler cable, EN 50 173

GVBA
: Global Village Business Association org., USA

GVV
: German Volunteer Votetakers (Usenet, DANA)

GWAI
: German Workshop on Artificial Intelligence conference, AI

GWBASIC
: Graphics and Windows Beginner's All-purpose Symbolic Instruction Code (BASIC, MS-DOS), "GW-Basic"

GWDG
: Gesellschaft fuer Wissenschaftliche Datenverarbeitung Goettingen [mbH] org.

GWS
: Gesellschaft fuer Wirtschafts- und Sozialkybernetik org., Uni Marburg

GWS
: Graphics WorkShop

Go to the previous, next section.

Go to the previous, next section.

-- H ---

H4P
: High Performance Parallel Processing Project (ICI)

HACMP
: High Availability Clustered Multi Processing (IBM, RS/6000, Bull), "HA/CMP"

HAL
: Hard Array Logic

HAL
: Hardware Abstraction Layer Windows NT

HAL
: Heuristically programmed ALgorithmic computer (2001)

HALPC
: Houston Area League of PC-Users, "HAL-PC"

HAMPS
: Host AUTODIN Message Processing System AUTODIN, mil.

HANFS
: Highly Available Network File System

HAP
: Host Access Protocol (RFC 907/1221)

HAS
: High Availability Subsystem (Bull)

HAT
: Hashed Address Table

HBA
: Host Bus Adapter (SCSI)

HBCI
: HomeBanking Computer Interface Internet, banking

HCI
: Human Computer Interaction BCS, conference

HCL
: Host Control Links

HCSDS
: High Capacity Satellite Digital Service

HCSS
: High Capacity Storage System (Novell, Netware)

HCT
: Hamburger ComputerTage fair

HCT
: High-speed CMOS [logic] with TTT-compatible [logic] levels (IC, MOS)

HCTDS
: High-Capacity Terrestrial Digital Service

HD
: High Density

HDB
: HoneyDanBer [standard] (Unix)

HDB3
: High Density Bipolar 3

HDBMS
: Hierachical DataBase Management System (DBMS, DB)

HDC
: Hard Disk Controller

HDCD
: High Density Compact Disk (CD, Sony, Phillips)

HDD
: Hard Disk Drive

HDF
: Hierarchical Data Format (NCSA)

HDH
: HDLC Distant Host

HDI
: HOOPS Device Interface (VAGI)

HDL
: Hardware Description Language (ASIC)

HDLC
: High-level Data Link Control (ISO)

HDM
: Hardware Device Module (I2O)

HDN
: Hochgeschwindigkeits-DatenNetz

HDR

High availability Data Replication (Informix, DB)

HDR

High Dynamic Range

HDSL

High data / bit rate Digital Subscriber Line (BELLCORE, AT&T, DSL)

HDSL

High-level Data Specification Language

HDX

Half DupleX

HEA

Hyundai Electronics of America manufacturer

HEC

Header Error Check

HEC

Header Error Control (ATM)

HECTOR

HEterogeneous Computer TOgetheR IBM, Uni Karlsruhe, Germany

HEL

Hardware Emulation Layer

HEL

Header Extension Length

HEMT

High Electron Mobility Transistor

HEPNET

High Energy Physics NETwork network

HERMES

Heuristic Emergency Response Management Expert System (XPS)

HERODE

Handling the Electronic Representation of mixed text - image Office Documents based on ECMA standard 101 (ESPRIT, SNI, TITN, ECMA, ODA)

HES

Home Electronic System (SNI)

HFD

Hauptanschluss fuer Direktruf / Datenleitung (Telekom), "HfD"

HFDS

Highly Functional Distributed System (MTRON)

HFS

Hierachical File System (Apple, CD)

HFT

High Function Terminal (AIX, IBM)

HGA

Hercules Graphics Adapter

HGC

Hercules Graphics Card

HHI

Heinrich Hertz Institut org., Deutschland

HI

Hochschulverband Informationswissenschaften org.

HIC

Human Interaction Component

HIC

Hybrid Integrated Circuit

HID

Human Interface Device (MS)

HIFD

HIgh capacity Floppy Disk (Sony, Fuji), "HiFD"

HIFI

Hypertext Interface For Information (ESPRIT)

HIFIVE

HIgh Fidelity Interactive Visual Environment (DEC), "Hi-FIVE"

HIGGENS

Human Interface Graphical GENeration System (Uni Colorado)

HINT

Hierachical INTegration [benchmark]

HIPPI

HIgh Performance Parallel Interface

HIPPI

High Performance Peripheral Interface

HIRD

HURD of Interface Representing Depth (GNU, HURD)

HISP

HIgh SPeed channel connector (Cray, I/O)

HLL

High Level Language

HLLAPI

High Level Language Application Programming Interface (IBM, API)

HLPI

Higher Layer Protocol Identifier

HLR

Home Location Register (LR, GSM)

HLS

Hue, Luminance, Saturation color system, DTP

HMA

High Memory Area

HMAC

keyed-Hashing for Message AuthentiCation cryptography, RFC 2104

HMC

Highspeed Memory Controller (Apple)

HMD

Head Mounted Display (VR)

HMI

Human-Machine Interface

HMMP

HyperMedia Management Protocol (MS, Intel, Cisco, WWW, HMMS)

HMMS

HyperMedia Management Schema (MS, Intel, Cisco, WWW)

HMOS

High performance Metal Oxide Semiconductor (IC)

HMP

Host Monitoring Protocol (RFC 869)

HMUX

Hybrid MUltipleXer (FDDI), "H-MUX"

HNF

Heinz Nixdorf Museumsforum org., Paderborn

HNI

Heinz Nixdorf Institut org., ZIT

HOL

Head Of Line

HOOPS

Hierachical Object Orientated Picture System (Ithaca, Autodesk, OOP)

HP
: Hewlett Packard manufacturer

HPBIDS
: Hewlett Packard Broadband Internet Delivery System (HP, Internet), "HP BIDS"

HPC
: High Performance Computing

HPCC
: High Performance Computing and Communication

HPCS
: Hifh Performance Communication Server (Tobit, NLM, Netware)

HPF
: High Performance FORTAN

HPFS
: High Performance FileSystem (OS/2)

HPGL
: Hewlett Packard Graphics Language (CAD, CAM, HP)

HPIB
: Hewlett Packard Interface Bus

HPM
: Hyper Page Mode

HPMI
: ??? [Schnittstelle] (CAD), "HP-MI"

HPOFS
: High Performance Optical File System (IBM, OS/2, MOD)

HPPA
: Hewlett Packard Precision Architecture (HP, RISC)

HPPCL
: Hewlett Packard Printer Control Language

HPR
: High Priority Request (VUMA)

HPTS
: High Performance Transaction Systems

HPUX
: Hewlett Packard / UniX (Unix), "HP/UX"

HPVEE
: Hewlett Packard ??? (HP), "HP VEE"

HPVUE

Hewlett Packard - Visual User Environment (HP, GUI), "HP-VUE"

HRC
: Hybrid Ring Control (FDDI)

HS
: High Speed (MODEM)

HSA
: High Speed Access

HSB
: Hue Saturation Brightness (DTP)

HSC
: Hierarchical Storage Controller

HSFS
: High Sierra File System

HSI
: High Speed Interface

HSI
: Hue, Saturation, Intensity (DTP)

HSIS
: ??? High Speed Interface / SNA, "HSI/S"

HSLAN
: High Speed Local Area Network, "HS-LAN"

HSLFX
: ??? [hardware desription language] (HDL, NTT, LSI), "HSL-FX"

HSLN
: High Speed Local Network

HSM
: Hardware Specific Module (ODI)

HSM
: Hierarchical Storage Management

HSM
: Hierarchisches SpeicherManagement

HSRP
: Hot Standby Router Protocol (Cisco, IOS)

HSSDS
: High-Speed Switched Digital Service

HSSI
: High Speed Serial Interface

HSSN
: HighSpeed Switching Network (Cray, ATM)

HST
: Hawaiian Standard Time [-1030] (TZ)

HST
: High Soft Tech manufacturer, ISDN

HST
: High Speed Technology (US Robotics)

HT
: Horizontal Tab

HTDM
: Hybrid Time Division Multiplexing [protocol]

HTF
: Hyper-g Text Format (Hyper-G)

HTH
: Hope This Helps slang

HTML
: HyperText Markup Language (Internet, WWW, SGML, RFC 1866/1942)

HTTP
: HyperText Transfer Protocol (WWW, RFC 2068)

HTTPD
: HyperText Transfer Protocol DAEMON (WWW, HTTP)

HTTPNG
: HyperText Transfer Protocol - Next Generation (WWW), "HTTP-NG"

HUGO
: Holland User Group for OS/2 Org., User group, OS/2, Netherlands

HURD
: HIRD of Unix-Replacing DAEMONs (GNU, HIRD)

HVC
: Hue-Value-Chroma [model] (Tektronic, DTP)

HVD
: High Voltage Differential [technology] (SCSI)

HYTEA
: HYperText Environment for Authoring (ESPRIT)

HYTIME
: Hypermedia/Time-based structuring language (SGML, ISO, IEC, IS 10744), "HyTime"

Go to the previous, next section.

TOC

Go to the previous, next section.

-- I ---

I2O
: Intelligent Input/Output (Intel)

I2O
: Intelligent Input/Output [bus]

I4DL
: Interface, Inheritance, Implementation, Installation Definition Language (DME, OSF)

IA
: Identification and Authentication, "I & A"

IAA
: Initial Address Acknowledgment

IAAIL
: International Association for Artificial Intelligence and Law org., AI

IAB
: Internet Activities / Architecture Board (RFC 1160/1601, Internet)

IAC
: InterApplication Communication (Apple)

IAC
: International Association for Cybernetics org.

IAE
: ISDN AnschlussEinheit (ISDN, Telekom)

IAF
: Internet Address Finder (WWW)

IAHC
: Internet Ad Hoc Committee org., Internet

IAI
: Institut fuer Angewandte Informatik org.

IAK
: Internet explorer Administrator Kit (MS, Internet)

IAK2
: Internet Access Kit / 2 (IBM), "IAK/2"

IAM
: Initial Address Message

IANA
: Internet Assigned Numbers Authority (Internet)

IANW
: International Academic Networkshops conference

IAO
: [fraunhofer] Institut fuer Arbeitswirtschaft und Organisation org., Stuttgart

IAOG
: International ADMD Operators Group org., ADMD, X.400

IAP
: Internet Access Provider (Internet)

IAPR
: International Association on Pattern Recognition org.

IAR
: Initial Address Reject

IAS
: Interactive Application Systen (DEC)

IAT
: Information Access Technology (Provider)

IAUG
: International AIX User Group org., Unix, AIX, User group

IBASES
: Intel Baseline AGP System Evaluation Suite (AGP, Intel)

IBCN
: Integrated Broadband Communication Network

IBEX
: International Business EXchange network

IBFI
: Internationales Begegnungs- und Forschungszentrum fuer Informatik org., GI

IBFN
: Integriertes BreitbandFernmeldeNetz

IBM
: International Business Machines manufacturer

IBN
: Integrated Business Network

IC
: Integrated Circuit

ICA

Independent Computing Architecture MS, Windows NT

ICA

Information Connection Architecture (SGI)

ICA

Integrated Chameleon Architecture (SGML)

ICA

Integrated Communications Architecture

ICA

Intelligent Communication Adapter (Banyan, VINES)

ICA

International Communications Association org.

ICADD

International Committee for Accessible Document Design org.

ICALP

International Colloquium on Automata, Languages and Programming EATCS, conference

ICAN

Individual Customer Access Network

ICAP

Internet Calendar Access Protocol (TCP/IP, Lotus)

ICC

Intelligent Communications Control

ICC

International Color Committee org.

ICC

International Conference on Communications conference

ICC

Internode Communication Channel (SMP, Scalis)

ICCAD

International Conference on Computer Aided Design IEEE, conference

ICCC

InterClient Communication Convention (X-windows)

ICCC

International Conference on Computer Comunication conference

ICCC

International Council on Computer Comunication org.

ICCCM

InterClient Communication Convention Manual (X-Windows)

ICCD

International Conference on Computer Design IEEE, conference

ICCI

International Conference on Computing and Information conference

ICCIMA

International Conference on Computational Intelligence and Multimedia Applications AI, conference

ICCS

Integrated Communications Cabling System (SNI)

ICD

Installable Client Driver (OpenGL)

ICD

International Code Designator (ATM)

ICDA

Integrated Cached Disk-Arrays

ICDB

Integrated Communications DataBase (DB)

ICDCS

International Conference on Distributed Computing Systems (INRIA, IEEE-CS)

ICE

In-Circuit-Emulation

ICF

International Cryptographic Framework HP, cryptography

ICH

Intelligent Connectzion Handling (BIG)

ICI

Industrial Computing Initiative (LLNL, LANL, Cray, ..., HPC, MPP)

ICIA

International Communications Industries Association org.

ICL

International Computers Limited manufacturer, UK, USA, Japan, Fujitsu

ICM

Image Color Matching (MS, Windows 95, API)

ICMC

International Computer Music Conference fair

ICMP

Internet Control Message Protocol [version 4] (TCP/IP, RFC 792, Internet)

ICMPV6
: Internet Control Message Protocol Version 6 (Internet, IPV6, RFC 1885), "ICMPv6"

ICMX
: ??? [transport level interface standard]

ICONZ
: Internet Company Of New Zealand org., Usenet

ICOT
: Institute for new generation COmputer Technology org., Japan, FGCS

ICP
: Independent Content Provider (MSN)

ICP
: Intelligent Computer Peripherals manufacturer

ICP
: Interface Change Proposal

ICP
: Internet Caching Protocol (Internet)

ICR
: Intelligent Character Recognition

ICS
: IBM Cabling System

ICS
: International Classification for Standards (ISO)

ICSC
: Intelligent Channel/Storage Control, "IC/SC"

ICSI
: International Computer Science Institute org., USA

ICSS
: Internet Connection Secure Server (IBM, WWW)

ICU
: Instruction Cache Unit (CPU, POWER)

ICU
: Interactive Chart Utility (IBM, GDDM)

ICU
: ISA Configuration Utility (BIOS, PNP, ISA)

ICURYY
: I see you are too wise slang

ID
: IDentification

IDAC
: Infobus Data Access Component (Lotus, Java)

IDAPI
: Integrated Database API DB, IBM, Novell, Borland, predecessor, BDE, API

IDAS
: Interchange DAta Structure

IDB
: Integrated DataBase (DB)

IDBEF
: Integrated DataBase Extract Format (IDB, DB)

IDBTF
: Integrated DataBase Transaction Format (IDB, DB)

IDC
: Inter-Device Communication (IBM, OS/2)

IDC
: International Data Corporation

IDCT
: Inverse Discrete Cosinus Transformation

IDDD
: International Direct Distance Dialing

IDE
: Integrated Development Environment

IDE
: Integrated Drive Electronics (HDD)

IDEA
: International Data Encryption Algorithm cryptography

IDEA
: Internet Design, Engineering, and Analysis notes (IETF)

IDEF
: Integrated [CAM] DEFinition (CAM)

IDES
: International Demonstration and Education System (R/3, SAP)

IDG
: International Data Group org.

IDI

Initial DOMAIN Identifier (NSAP, IDP)

IDIS
: Integrated Dealer and Importer System (MBAG)

IDL
: Interactive Data Language (Research Systems)

IDL
: Interface Definition Language (DCE, CORBA)

IDLE
: International Date Line, East [+1200] (TZ)

IDLW
: International Date Line, West [-1200] (TZ)

IDN
: Integrated Digital Network

IDOC
: Intermediate DOCuments (ALE, SAP)

IDOMENEUS
: Information and Data on Open MEdia for NEtworks of USers WWW, Uni Hamburg, Germany

IDP
: Initial DOMAIN Part (NSAP)

IDP
: Internet Datagram Protocol (XNS)

IDRC
: ??? Redundancy Check (Datenkompression)

IDS
: Information Delivery System

IDS
: Inter Data Systems [gmbh] manufacturer

IDS
: Internal Directory System

IDSS
: Interoperability Decision Support System

IDT
: Integrated Device Technology manufacturer

IDT
: Interrupt Descriptor Table

IDTRC
: Interrupt Descriptor Table Register Cache (IDT, CPU, Intel)

IDU

Interface Data Unit

IDUPGSSAPI

Independant Data Unit Protection Generic Security Service API (GSS, API), "IDUP-GSS-API"

IDV

Individuelle DatenVerarbeitung

IDVC

Integrated Data/Voice Channel

IE

Information Element (ATM, ISDN)

IE

Information Engineering

IEAAIE

[international conference on] Industrial and Engineering Applications of Artificial Intelligence and Expert systems conference, ISAI, ..., AI, "IEA/AIE"

IEC

Inter-Exchange Carrier (AT&T, MCI, LEC)

IEC

International Electrotechnical Commission org., Schweiz

IECC

Informix Enterprise Command Center (Informix)

IEE

Institute of Electrical Engineers org., UK

IEEE

Institute of Electrical and Electronic Engineers org., USA

IEEECS

Institute of Electrical and Electronic Engineers - Computer Society org., IEEE, "IEEE-CS"

IEEJ

??? org.

IEF

Information Engineering Facility

IEICE

??? org.

IEMMC

Internet EMail Marketing Council Internet, org., Spam

IEMSI

Interactive Electronic Mail Standard Identification

IEN
: Internet Engineering Notes

IEN
: Internet Experiment Notebook predecessor, RFC

IEPG
: Internet Engineering and Planning Group (Internet, RFC 1690)

IEPRC
: International Electronic Publishing Research Centre org., PIRA

IES
: Intelligent Emulation Switching (Epson)

IESE
: [fraunhofer] Institut fuer ??? Software Engineering org.

IESG
: Internet Engineering Steering Group (IETF)

IETF
: Internet Engineering Task Force org., IAB, Internet, RFC 1603

IETFWG
: Internet Engineering Task Force - Working Group (RFC 1603, IETF), "IETF-WG"

IETM
: Interactive Electronic Technical Manual

IETS
: Interim European Telecommunications Standards (GSM)

IEW
: Information Engineering Workbench (IBM)

IF
: Information Flow (IN)

IFAC
: International Federation for Automatic Control org., OEsterreich

IFAD
: Institute of Applied Computer Science Org., Denmark

IFC
: Internet Foundation Classes (ONE, Netscape, Java)

IFCM
: Independent Flow Control Messages (SSP)

IFD
: Information Flow Diagram (IRM)

IFE
: Internet Forum Europe fair

IFIOM
: Intelligent FDDI Input Output Module (SMC, ES/1, FDDI)

IFIP
: International Federation of Information Processing societies org.

IFMP
: Ipsilon Flow Management Protocol (IP)

IFNA
: International FidoNet Association org., Fido

IFP
: Instruction Fetch Pipeline (Motorola, CPU)

IFR
: Interleaved Frame Recording (Video)

IFS
: ICOT Free Software (ICOT)

IFS
: Installable File System (OS/2, DOS)

IFS
: Intelligent Fax System (Ricoh)

IFS
: Internal Field Seperator (Unix)

IGC
: Institute for Global Communications (USA)

IGD
: [fraunhofer] Institut fuer Graphische Datenverarbeitung Org., Darmstadt, Germany

IGES
: Initial Graphics Exchange Specification (ANSI, USA, CIM, CAD)

IGMP
: Internet Group Management / Multicast Protocol (RFC 1112, IP)

IGN
: IBM Global Network network

IGOSS
: Industry/Government Open System Specification

IGP
: Interior Gateway Protocol

IGRP

??? Protocol (IPX)

IHV
: Independent Hardware Vendor

IIA
: Information Industry Association org.

IIASA
: International Institute for Applied Systems Analysis org., OEsterreich

IICM
: Institute for Information processing and Computer supported new Media org., OEstereich

IID
: Interface IDentifier (COM)

IIIR
: Integration of Internet Information Ressources (IETF)

IIIS
: International Institute of Informatics and Systemics org., USA

IIL
: Integrated Injection Logic

IIN
: Integrated Information Network

IIOP
: Internet Inter-ORB Protocol (CORBA, Internet, ONE, GIOP, TCP/IP)

IIP
: Interoperability Improvement Panel

IIRC
: If I Remember/Recall Correctly Usenet, IRC, slang

IIS
: [fraunhofer] Institut fuer Integrierte Schaltungen org.

IIS
: Intelligence Information System mil.

IIS
: Internet Information Server (MS)

IISP
: Interim Inter-Switch Signaling Protocol (P-NNI, ATM)

IIT
: Integrated Information Technology

IITF
: Information Infrastructure Task Force org., USA

IJCAI

International Joint Committee / Conference on Artificial Intelligence conference, Org., AI

IKE

IBM Kiosk for Education (IBM)

IKP

Internet Keyed Payment [protocols] (IBM, WWW), "iKP"

ILFA

Integrierte Logische Funktionen fuer Anwendungen IWBS, Uni Karlsruhe, Uni Duisburg, Germany

ILLIGAL

ILLinois Genetic Algorithm Laboratory org., Uni Illinois, USA, "IlliGAL"

ILLINET

ILlinois LIbrary NETwork network, USA

ILMI

Interim Link / Local Management Interface (ATM)

ILPNET

[european] Inductive Logic Programming scientifc NETwork

ILS

Intelligente LernSysteme

IM

Information Management

IM

Interface Module

IMA

Interactive Multimedia Association org.

IMA

International MIDI Association org.

IMAC

Isochronous Media Access Control (FDDI), "I-MAC"

IMACS

International ??? org.

IMAIL

Intelligent MAIL

IMAL

Integrated Media Architecture Laboratory Bell, org., USA

IMAP

Internet Message Access Protocol (RFC 2060)

TOC

IMB
: Intel Media Bench

IMC
: ??? org., "IM&C"

IMC
: Information Management Concept

IMCO
: In My Considered Opinion slang, Usenet, IRC

IMDB
: Internet Movie DataBase (Internet, WWW, DB)

IME
: Input Method Editor (MS, Windows)

IMEI
: International Mobile Equipment Identity (MS, GSM)

IMG
: IMplementation Guide (R/3, SAP)

IMHO
: In My Humble Opinion slang, Usenet, IRC

IMIA
: International Medical Informatics Association

IML
: Initial Machine Load

IMMD
: Institut fuer Mathematische Maschinen und Datenverarbeitung Org., Uni Erlangen, Germany

IMNSCO
: In My Not So Considered Opinion slang, Usenet, IRC

IMNSHO
: In My Not So Humble / Honest Opinion slang, Usenet, IRC

IMO
: In My Opinion slang, Usenet, IRC

IMP
: Interface Message Processors (ARPANET, MILNET)

IMPACT
: Information Market Policiy ACTions (ECHO)

IMR
: Information Management Representitive (IM)

IMR
: Internet Monthly Report (Internet)

IMR
: Interrupt Mask Register

IMS
: [fraunhofer] Institut fuer Mikroelektronische Schaltungen und systeme Org., Dresden, Germany

IMS
: Information Management System (IBM)

IMS
: Integrated Management System

IMS
: Integrated Measurement Systems [inc] manufacturer

IMS
: Intelligent Manufacturing System (MITI)

IMS
: International Meta Systems manufacturer

IMSESA
: Information Management System/Enterprise Systems Architecture (IBM), "IMS/ESA"

IMSI
: International Mobile Subscriber Identity (MM, GSM)

IMSO
: Integrated Micro Systems Operation (Intel)

IMSP
: nternet Message Support Protocol (Internet)

IMSVS
: Information Management System/Virtual Storage, "IMS/VS"

IMTC
: International Multimedia Teleconferencing Consortium org., MS, Intel, Apple, IBM, SNI, AT&T, ...

IN
: Individual Network [e.v.] org., ISP

IN
: Intelligent Network

INA
: Intelligent Network Architecture (IN)

INAP
: Intelligent Network Application Protocol (IN)

INCM
: Intelligent Network Conceptual Model (IN)

INDC
: [international conference on] Information Networks and Data Communication IFIP, conference

INET
: International NETworking conference conference, "iNET"

INGRES
: INteractive Graphic REtrieval System

INL
: Inter Node Link

INMS
: Integrated Network Management System

INN
: Inter Node Network

INNC
: International Neural Network Conference INNS, conference

INNS
: International Neural Network Society org., AI

INOC
: Internet Network Operations Center

INPOL
: INformationssystem der POLizei

INRIA
: Institut National de Recherche en Informatique et Automatique Org., France

INSITS
: INternational Symposium on IT Standardization conference, IFIP, GI

INST
: INformation Standards and technology Standardization

INTAP
: Interoperability Technology Association for Information Processing org.

INTEL
: INtegriertes TEileLogistiksystem (MBAG)

INTERNIC
: Inter Network Information Center org., Internet, "InterNIC"

INX
: INformation eXchange

INXS
: INternet eXchange Service (ECRC)

IO
: Input/Output, "I/O"

IOC
: Initial Operational Capability

IOC
: Input / Output Controller

IOC
: Input/Output Cluster (Cray, I/O)

IOCCC
: International Obfuscated C Code Contest

ION
: Internetworking Over NBMA (LIS, LAG, ATM, NBMA)

IONL
: Internal Organization of the Network Layer (OSI)

IOOC
: Integrated Optics and Optical fiber Communication conference

IOP
: Input / Output Processor

IOPL
: Input Output Privilege Level

IOS
: Internetworking Operating System (Cisco, OS)

IOS
: InterOrganizational Systems

IOTE
: Initial Operational Test and Evaluation, "IOT & E"

IOUG
: International Oracle Users Group org., DB, Oracle, User group

IP
: Information Provider

IP
: Instruction Pointer [register] CPU, Intel, assembler

IP
: Intellectual Property

IP

Intelligent Peripheral (IN)

IP

Internet Protocol [version 4] (RFC 791)

IPA

[fraunhofer] Institut fuer Produktionstechnik und Automation org., Stuttgart

IPA

Information Processing promotion Agency org., MITI, Japan

IPC

Internet Privacy Coalition (Internet)

IPC

InterProcess Communications [protocol]

IPCE

InterProcess Communication Environment

IPCP

[PPP] Internet Protocol Control Protocol (PPP, RFC 1332)

IPCR

International Conference on Pattern Recognition conference, IAPR

IPCS

Informatique et Calcul Parallele de Strasbourg Org., France, HPC

IPCS

Interactive Problem Control System

IPD

Intelligent Printer Data

IPDS

IBM Personal Dictation System IBM, voice processing

IPDS

Intelligent Printer Data Stream (IBM, CCS)

IPDU

Internet Protocol Data Unit (PDU)

IPDVMRP

IP Distance Vector Multicast Routing Protocol (RFC 1075, IP), "IP-DVMRP"

IPE

Integrated Programming Environment

IPES

Improved Proposed Encryption Standard (IDEA)

IPFC

Information Presentation Facility Compiler (IBM, OS/2)

TOC

IPG
: International Programmers Guild org.

IPG
: InterPacket Gap (IPX, IP (???))

IPI
: Intelligent Peripheral Interface [protocol]

IPL
: Initial Program Load

IPM
: Impulses Per Minute

IPM
: InterPersonal Messaging

IPM
: Interruptions Per Minute

IPMS
: InterPersonal Messaging System

IPN
: Info Pool Network (ISP)

IPNG
: Internet Protocol Next Generation (IP, RFC 1550/1752), "IPng"

IPOC
: Interim Policy Oversight Committee (Internet, TLD)

IPP
: Internet Presence Provider (Internet)

IPS
: Information Processing Standards

IPSE
: Integrated Programming Support Environment (CASE)

IPSEC
: Internet Protocol SECurity org., Internet, IETF, DES

IPSJ
: Information Processing Society of Japan org., Japan

IPSP
: Internet Protocol Security Protocol

IPTC
: International Press Telecommunications Council org.

IPTD

[microsoft] Internet Platform and Tools Division (MS)

IPU
: Intelligent Processing Unit

IPU
: ISDN-datex-P-Umsetzer

IPV6
: Internet Protocol Version 6 (IP, RFC 1883/1884), IPv6

IPVR
: Institut fuer Parallele und Verteilte Rechensysteme org., Uni Stuttgart

IPW
: Incremental Packet Writing (CD-R)

IPX
: Internet Packet eXchange (Novell, Netware)

IPXCP
: [PPP] Internetwork Packet eXchange Control Protocol (PPP, RFC 1552)

IPXODI
: IPX Open Datalink Interface (Novell, Netware)

IPXSPX
: IPX / Sequenced Packet eXchange (Novell, Netware), "IPX/SPX"

IPXWAN
: Internet Packet eXchange over various WAN media (Novell, IPX, WAN, RFC 1634)

IR
: Interface Repository (ORB)

IR
: Internet Registry

IRC
: InformationsRessourcen Controlling (IM)

IRC
: Internet Relay Chat [protocol] (RFC 1459, Internet)

IRCS
: Institute for Research in Cognitive Science org., Uni Pennsylvania, USA

IRDA
: InfraRed Data Association Org., manufacturer, "IrDA"

IRDS
: Information Resource Dictionary System (ANSI, ISO, IM)

IRF
: Integer Register File (DEC, Alpha, CPU)

IRIS
: Institute for Robotics and Intelligent Systems org., Canada

IRL
: In Real Life slang, Usenet, IRC

IRLMPTP
: InfraRed ??? (IRDA), "IrMLP/TP"

IRM
: Information Resources Management

IRQ
: Interrupt ReQuest

IRR
: Interrupt Request Register

IRRP
: Inter-DOMAIN Routing Protocol (ISO 10747)

IRSG
: Internet Research Steering Group org., Internet

IRTF
: Internet Research Task Force (IAB, Internet)

IRTP
: Internet Reliable Transaction Protocol (RFC 938)

IRV
: International Reference Version [character set] (ISO, IS 646)

IS
: Intermediate System (ATM)

IS
: International Standard (ISO)

IS
: International Supplement (ODT)

ISA
: Industry Standard Architecture (PC)

ISA
: Instruction Set Architecture (CPU)

ISA
: Integrated Systems Architecture (ESPRIT, ANSA)

ISA
: Interactive Services Association org., USA

ISADS

International Symposium on Autonomous Decentralized Systems IEEE-CS, conference

ISAGAT

Industry Standard Architecture - Guaranteed Access Time (ISA, DMA, PCI)

ISAKMP

Internet Security Association and Key Management Protocol

ISAM

Index-Sequential Access Mode (DB)

ISAPI

Internet Server Application Programmer's Interface (MS, C/S, WWW, API)

ISBN

International Standard Book Number

ISC

Intelligent Screen Cognition

ISC

Internet Service Center (ISP)

ISC

Internet Software Consortium org., Internet

ISCA

International Society for Computers and their Applications org., AI

ISD

Interface Summary Design

ISDN

Integrated Services Digital Network (ITU)

ISDNUP

ISDN User Part (ISDN, GSM), "ISDN-UP"

ISDV

Integrated Software Development and Verification

ISEE

Information System Engineering Environment (Westmount, CASE)

ISEE

Integrated Software Engineering Environment [reference model] (NIST, ECMA, CASE)

ISFUG

Integrated Software Federal User Group org., User group

ISH

Information SuperHighway

ISI

[fraunhofer] Institut fuer Systemtechnik und Innovationsforschung org.

TOC

ISI
: Information Sciences Institute org., USA

ISI
: Information Society Initiative (UK)

ISIS
: Intermediate System to Intermediate System [protocol / routing] (OSI, RFC 1195), "IS-IS"

ISIT
: [fraunhofer] Institut fuer SiliziumTechnologie Org., Itzehoe, Germany

ISKO
: International Society for Knowledge Organisation org.

ISLM
: Integrated Services Line Module

ISLN
: Integrated Services Local Network

ISLU
: Integrated Services Line Unit

ISN
: Information Systems Network

ISN
: Initial Segment / Sequence Number (TCP/IP)

ISN
: Integrated Systems Network

ISN
: Internet Shopping Network network, WWW

ISO
: International Standards Organisation org.

ISOC
: Internet SOCiety org., Internet

ISODE
: ISO Development Environment (OSI)

ISONET
: ISO information NETwork ISO, network

ISP
: Image Synthesis Processor (IC, Graphik, TSP)

ISP
: Integrated System Peripheral control (Wyse)

ISP

Interactive String Processor (BS2000)

ISP

International Standardized Profiles (ISO)

ISP

Internet Service Provider (Internet)

ISPA

Internet Service Provider Austria org., ISP

ISPA

ISDN PAcket [driver] (ISDN)

ISPC

Internet Service Providers' Consortium ISP, IPP, org.

ISPF

Interactive System Productivity Facility (IBM)

ISPN

International Standard Program Number

ISPO

Information Society Project Office [of the european comission] Org., Europe, ECHO

ISPS

??? [hardware desription language] (HDL)

ISQL

Interactive Structured Query Language

ISR

Interrupt Service Register

ISR

Interrupt Service Routine

ISRC

??? [Kodierung]

ISSAC

International Symposium on Symbolic and Algebraic Computation ACM, conference

ISSCC

International Solid State Circuits Conference conference

ISSN

Integrated Special Services Network

ISSN

International Standard Serial Number

ISSRE

International Symposium on Software Reliability Engineering IEEE-CS, conference

ISST
: [fraunhofer] Institut fuer Software- und SystemTechnik Org., Dortmund, Germany

IST
: Immerse System Technology (VR)

IST
: Indian Standard Time [+0530] (TZ)

IST
: Initial System Test

ISTO
: Information Science and Technology Office org., DARPA

ISUP
: Integrated Services User Part (ISDN)

ISV
: Independent Software Vendor

ISV
: Independent Solution Vendor (DEC)

ISV
: Information Service Vendor

IT
: Information Technology

IT
: Iran Time [+0300] (TZ)

ITA
: Interim Type Approval

ITAA
: Information Technology Association of America org., USA

ITE
: InformationsTechnische Einrichtungen

ITG
: InformationsTechnische Gesellschaft org., VDE

ITLB
: Instruction Tanslation Look-aside Buffer (CPU)

ITM
: Information Technology Management

ITMS
: Immediate check Truth Maintenance System (AI)

ITOT

ISO Transport service on TCP/IP (ISO, TCP/IP, RFC 2126)

ITR

Internet Talk Radio (Internet)

ITRC

Information Technology Research Centre org., Canada

ITRON

Industrial TRON (TRON)

ITS

Incompatible Time-sharing System (DEC)

ITS

International Telecommunications Society org.

ITSDN

Integrated Tactical/Strategic Data Network mil., USA, Netzwerk

ITSEC

[european] Information Technology Security Evaluation Criteria Europe

ITSEM

Information Technology Security Evaluation Manual / Methodology (ITSEC)

ITSP

Internet Telephony Service Provider (Internet, CAT)

ITSPO

Information Technology Standards Program Office org., USA

ITT

International Telegraph and Telephone org.

ITTI

Information Technology Training Initiative org., UK

ITTM

??? [institute of telecommunications and information technology] org., Malaysia

ITU

Information Transport Utility

ITU

International Telecommunications Union org.

ITUR

International Telecommunications Union - Radio sector (ITU), "ITU-R"

ITUSA

Information Technology Users Standards Association org., USA

ITUT

ITU Telecommunications [standards group] Org., successorr, CCITT, "ITU-T"

ITUTSS
: International Telecommunications Union Telecommunication Standards Sector (ITU)

IU
: Integer Unit (CPU)

IUK
: Informations- und Kommunikationstechnik, "IuK"

IUKDG
: Informations- Und KommunikationsDiensteGesetz

IUMA
: Internet Underground Music Archive (WWW)

IUP
: Interim Update Package (3COM)

IUT
: Implementation Under Test

IUTSUT
: Implementation Under Test - System Under Test, "IUT-SUT"

IV
: InformationsVerarbeitung

IV
: Initialisation Vector cryptography

IVA
: International Voice Association org.

IVDENIC
: InteressenVerbund DEutsches Network Information Center org., Internet, "IV-DENIC"

IVDLAN
: Integrated Voice and Data Local Area Network

IVR
: Interactive Voice Response (CTI)

IVS
: IBM Verkabelungs-System

IVS
: Interactive Visualization Systems

IVTS
: International Video Teleconferencing Service

IWBS
: Institut fuer WissensBasierte Systeme org., IBM

IWC

Inside Wire Cable

IWF

InterWorking Function (IN)

IWS

Intelligent Work-Station

IWT

International Workshop on Telematics conference, IEEE-CS, SEE

IWU

InterWorking Unit (IN)

IWV

ImpulsWahlVerfahren (MODEM)

IXC

IntereXchange Carrier (FCC, LATA)

IZM

[fraunhofer] Institut fuer Zuverlaessigkeit und Mikrointegration Org., Berlin, Germany

Go to the previous, next section.

Go to the previous, next section.

-- J ---

JACM
: Journal of the Association for Computing Machinery (ACM)

JADE
: Jasmine Application Development Environment (Jasmine, DB, CA)

JANET
: Joint Academic NETwork network, UK

JAPH
: Just Another PERL Hacker (PERL)

JAR
: Java Archive (Java)

JARS
: Java Applet Rating Service (Java, WWW)

JAVAOS
: Java Operating System (Java), "JavaOS"

JBIG
: Joint Bi-level Image expert Group org., JTC1

JCALS
: Joint Computer-Aided Logistics System (CAL)

JCB
: Job Control Block (BS2000)

JCE
: Java Cryptographic Extension (Java)

JCF
: JESSI Common Framework

JCL
: Job Control Language (IBM, OS/360)

JCSS
: ??? org.

JDBC
: Java standard DataBase Connectivity (DB, Java, Sun, Borland, ODBC, API)

JDK
: Java Development Kit (Sun, Java)

JEDEC

???

JEIDA

Japan Electronics Industry Development Association org., Japan

JES

Job Entry Subsystem (IBM)

JFC

Java Foundation Classes (Java, API)

JFIF

JPEG File Interchange Format

JFS

Journaled File System (LVM)

JFT

Job File Table (DOS)

JICST

Japan Information Center of Science and Technology org., Japan

JIEO

Joint Interoperability Engineering Organization Org., DISA, mil., USA

JIF

JPEG Interchange Format (JPEG)

JIPDEC

Japan Information Processing DEvelopment Center org., Japan

JIPS

JANET Internet Protocol Service

JIS

Japanese Institute of Standards org., Japan

JISC

Japanese Industrial Standards Committee org., Japan

JIT

Just In Time [compiler]

JITC

Joint Interoperability Technology Center Org., DISA, mil.

JMSC

Japanese MIDI Standard Committee org., Japan

JMSWG

Joint Multi-TADIL Standards Working Group Org., mil., TADIL

JNDI

Java Naming and Directory Interface (JavaSoft, API)

JNSS

??? org.

JODS

Jasmine Object Database Server (Jasmine, DB, CA)

JOLT

Java OnLine Transactions (Java, Bea, OLTP)

JOOP

Journal of Object Orientated Programming (OOP)

JOSS

Joint Object Services Submission

JOTS

Joint Operational Tactical System mil.

JOVE

Jonathan's Own Version of EMACS

JPEG

Joint Photographics Expert Group org., JTC1, RFC 1521, JPEG

JPLDIS

Jet Propulsion Laboratory Display Information System

JSA

Japanese Standards Association org., Japan

JSAI

??? org.

JSAN

Joint Staff Automation of the Nineties mil.

JSD

Jackson System Development

JSME

??? org.

JSP

Jackson Structured Programming

JSP

Java Server Pages (Java, Sun)

JSPE

??? org.

JSSA

Japan Society for System Audits org., JIPDEC, Japan

JST
: Japan Standard Time [+0900] (TZ)

JT
: Java Time [+0730] (TZ)

JTA
: Joint Technical Architecture JIEO, mil.

JTAG
: Joint Test Action Group org., Philips, IC

JTB
: Jump Trace Buffer (CPU)

JTBP
: Job To Be Processed [block] (BS2000)

JTBPX
: Job To Be Processed [block] eXtension (BS2000)

JTC1
: Joint Technical Committee 1 (ISO, IEC, IT)

JTM
: Job Transfer and Manipulation (ISO 8831/32)

JTMS
: Justification based Truth Maintenance System (AI)

JTSSG
: Joint Telecommunications Standards Steering Group mil.

JUGHEAD
: Jonzy's Universal Gopher Hierachy Excavation And Display

JUNET
: Japan Unix NETwork network, Unix

JURIS
: JURistisches InformationsSystem

JV
: JobVariablen (BS2000)

JVIDS
: Joint Visually Integrated Display System mil.

JVM
: Java Virtual Machine (Java)

JWICS
: Joint Worldwide Intelligence Communications System mil.

TOC

Go to the previous, next section.

Go to the previous, next section.

-- K ---

KAN
: KriminalAktenNachweis (INPOL)

KARL
: ??? [hardware desription language] (HDL)

KB
: KiloByte

KBPS
: KiloBits Per Second

KBS
: Knowledge-Based System

KCGI
: Kyocera ??? (Kyocera)

KCGL
: KyoCera Graphic Language (Kyocera)

KCMS
: Kodak Color Management System (Kodak, DTP)

KDC
: Key Distribution Center

KDE
: K Destop Environment (Linux, KDE)

KDM
: K Display Manager (KDE)

KDT
: Keyboard Display Terminal

KERMIT
: Kl-10 Error-free Reciprocal Micro Interconnect over Tty lines

KES
: Key Escrow System cryptography

KFM
: K File Manager (KDE)

KGRZ
: Kommunales GebietsRechenzentrum Giessen org.

KHG
: Kernel Hacking Guide (Linux)

KI
: Kuenstliche Intelligenz

KIF
: Konferenz der Informatik Fachschaften conference

KIPS
: Kilo Instructions Per Second

KIR
: Kyocera Image Refinement (Kyocera)

KISS
: Keep It Simple Stupid slang

KIT
: Kernel software for Intelligent Terminals (T-Online, Telekom, BTX)

KK
: Konnectivity Koordination (DE-NIC, DOMAIN)

KLICK
: Karlsruher LIChtleiter-Kommunikationsnetz Uni Karlsruhe, Germany

KLIPS
: Kilo Logical Inferences Per Second (AI, KI, XPS)

KLT
: Karhunen Loeve Transformation

KNN
: Kuenstliche Neuronale Netze neural nets

KNOOM
: KNowledge Orientated Office Model

KORE
: ???

KPCMS
: Kodak Precision Color Management System (Kodak, DTP)

KPDL
: Kyocera ??? (Kyocera)

KPDL
: Kyocera Page Description Language (Kyocera)

KPI
: Kernel Programming Interface (Unix, API)

KPM

Kyocera PrintMonitor (Kyocera)

KPOP

Kerberized Post Office Protocol (POP3)

KPT

Kai's PowerTools (DTP)

KQML

Knowledge Query Manipulation Language (AI)

KR

[brian] Kernighan & [dennis] Ritchie [c standard], "K&R"

KR

Knowledge Representation (AI)

KSA

Kalman Saffran Associates manufacturer

KSA

KommunikationsStrukturAnalyse (OA, TUB)

KSDS

Key Sequenced Data Set (VSAM)

KSH

Key Strokes per Hour

KSH

Korn SHell (Unix, Shell)

KSLNRC

Knowledge Systems Laboratory of the National Research Council org., Canada, AI

KSPM

KeyStrokes Per Minute

KSR

Keyboard Send Receive

KV

KabelVerzweiger

KV

Karnaugh Veitch diagram

KWM

K Window Manager (KDE)

Go to the previous, next section.

Go to the previous, next section.

-- L ---

L2F
: Layer 2 Forwarding [protocol] (Shiva, Cisco, IP, IPX, LLC)

L2TP
: Layer 2 Tunneling Protocol (PPP, VPN, IP)

LA
: Limited Availability (IBM, OS/2)

LA
: Location Area (MSC, GSM)

LAC
: Lotus Authorized Consultants (Lotus)

LADT
: Local Access Data Transport

LAEC
: Lotus Authorized Education Center (Lotus)

LAG
: Logical Address Group (ION)

LAM
: Local Area Multicomputer (Parallel Computing)

LAN
: Local Area Network

LANE
: Local Area Network Emulation [over ATM specification] (ATM)

LANL
: Los Alamos National Laboratory org., USA

LAP
: Link Access Procedure / Protocol (CCITT, X.25)

LAPB
: Link Access Procedure - Balanced (CCITT, LAP, X.25)

LAPD
: Link Access Procedure on the D channel (X.21, ISDN, TA)

LAPIC
: Local Advanced Programmable Interrupt Controller (PIC)

TOC

LAPM
: Link Access Procedure for Modem

LAPS
: LAN Adapter and Protocol Support

LASER
: Light Amplification by Stimulated Emission of Radiation

LASS
: Local Area Signaling Service

LAT
: Local Area Transport (DEC)

LATA
: Local Access Transport Area

LATM
: Local Asynchronous Transfer Mode

LAVAS
: LAger- und VersandAbwicklungsSystem (MBAG)

LAVC
: Local Area VAX Cluster (VAX, DEC)

LAW
: Local Authority Workstation

LAWN
: Local Area Wireless Network

LBA
: Logical Block Addressing (EIDE)

LBM
: Local Bus Master

LBN
: Logical Block Number (LBA)

LBT
: Local Bus Targets

LBX
: Low-Band with X

LC
: Loopback Capability (UNI, ATM, OAM)

LCA
: Logic Cell Array

LCC

TOC

Local Control Center

LCCM

LAN Client Control Manager (IBM, LAN)

LCD

Liquid-Crystal Display

LCD

LISP Code Directory (EMACS)

LCD

Loss of Cell Delineation (UNI, ATM)

LCGI

Local Common Graphics Interface (CGI, WWW)

LCM

LEAF Creation Method EES, cryptography

LCN

Logical Channel Numbers

LCP

[PPP] Link Control Protocol (PPP, RFC 1570)

LCR

Least Cost Routing

LCS

Laboratory for Computer Science (MIT)

LCS

Liquid Crystal Shutter

LCS

Lotus Communication Server

LCSS

Liquid Crystal Stereoscopic Shutter [display]

LCT

Last Compliance Time (GCRA)

LCU

LAN CID Utility (CID, IBM)

LCV

Line Coding Violation [error event] (DS1/E1, DS3/E3)

LD

LAN Destination (ATM)

LDAP

Lightweight Directory Access Protocol (RFC 1777, X.500, DS)

LDAPAPI
: Lightweight Directory Access Protocol Application Program Interface (LDAP, RFC 1823, API), "LDAP API"

LDCM
: LANDesk Client Manager

LDID
: Logical Disk IDentifier (MS)

LDP
: Linux Document Projecta (Linux)

LDP
: Loader Debugger Protocol (RFC 909)

LDR
: Light Detect Resistor

LDT
: Local Descriptor Table (CPU, Intel)

LDTR
: Load Descriptor Table Register CPU, Intel, assembler

LDTRC
: Local Descriptor Table Register Cache (LDT, Intel, CPU)

LDVA
: LaenderDatenVerareitungsAnlage

LE
: LAN Emulation (LANE, ATM)

LEAF
: Law Enforcement Access Field EES, cryptography

LEARP
: LAN Emulation Address Resolution Protocol (LANE, ARP, ATM), "LE-ARP"

LEAS
: LATA Equal Access System

LEC
: LAN Emulation Client (LANE, ATM)

LEC
: Layered Error Correction (CD)

LEC
: Local Exchange Carrier (FCC, LATA, IEC)

LECID
: LAN Emulation Client IDentifier (LANE, ATM, LEC)

LECS
: Local area network Emulation Configuration Server (ATM, LANE, LEC)

LED
: Light-Emitting Diode

LEL
: Link, Embed and Launch (UNIX)

LEN
: Low Entry Networking (IBM, SNA, PU)

LEO
: Link Everything Online (WWW, TUM)

LES
: LAN Emulation Server (LANE, ATM)

LES
: Line Errored Seconds (DS1/E1, DS3/E3)

LF
: Line Feed (ASCII)

LF
: Login Facility (DCE)

LFA
: Link Field Address (Forth)

LFA
: Local Feature Analysis

LFAP
: Leight weight Flow Admission Protocol (Cabletron, RFC 2124)

LFB
: Linear Frame Buffer (CPU, VESA)

LFN
: Long File Names

LFO
: Low Frequency Oscillator

LFS
: Loopback File System

LFSR
: Linear Feedback Shift Register

LGX
: Linux/GNU/X [distribution] (Yggdrasil, Linux, GNU)

LIC
: Licensed Internal Code

LIF
: Low Insertation Force (IC)

LIFD
: Last In First Drop

LIFE
: Laboratory for International Fuzzy Engineering [research] (MITI)

LIFE
: Logistics Interface For manufacturing Environment

LIFO
: Last In First Out

LIJP
: Leaf Initiated Join Parameter

LILO
: LInux [boot] LOader (Linux)

LIM
: Lotus - Intel - Microsoft manufacturer

LIMDOW
: Laser / Light Intensity Modulation, Direct OverWrite (MOD)

LIMEMS
: Lotus - Intel - Microsoft Expamded Memory Specification (Lotus, Intel, MS, EMS), "LIM EMS"

LIP
: Large Internet Packet

LIPS
: Logical Inferences Per Second (AI, KI, XPS)

LIS
: Logical IP Subnet (RFC 1577, ION)

LISA
: Linux Installation & System Administration (Linux, LST)

LISP
: LISt Processor (LISP)

LISP
: Lots of Isolated Silly Parentheses LISP, slang

LISUAF
: [thueringer] LandesInformationsSystem Umwelt, Agrar und Forst (UIS), "LIS-UAF"

LITA

Library and Information Technology Association org., USA

LIU

Line Interface Unit

LIV

Link Integrity Verification

LIVID

Language Identification and Voice IDentification

LIW

Long Instruction Word (CPU)

LKI

Labor fuer Kuenstliche Intelligenz Org., KI, Hamburg, Germany

LLAP

Localtalk Link Access Protocol (AppleTalk, LAP)

LLATMI

Lower Layer ATM Interface (ATM)

LLB

Local Location Broker (NCS)

LLC

Logical Link Control (IEEE 802.2, ISO, OSI)

LLCSNAP

Logical Link Control/SubNetwork Access Protocol (LLC, SNAP), "LLC/SNAP"

LLN

Line Link Network

LLNL

Lawrence Livermore National Laboratory org., USA

LLP

Link Level Protocol (BTX)

LMDS

Local Multi-point Distribution System

LME

Layer Management Entity (OSI)

LMFAO

Laughing My Fucking Ass Off Usenet, IRC, slang

LMI

Layer Management Interface (ATM)

LMS

Library Maintenance System (BS2000)

LMU
: LAN Manager for Unix (Unix), "LM/U"

LMX
: LAN Manager for uniX (LAN, Unix, MS), "LM/X"

LNN
: Lotus Notes Network (Lotus)

LOC
: LAN Operations Center (LAN)

LOC
: Lines Of Code

LOC
: Loss of Cell delineation (UNI, ATM)

LOCIS
: Library Of Congress Information System (Internet)

LOCT
: Layered Open Crypto Toolkit RSA, cryptography

LOF
: Loss of Frame (UNI, ATM, DS3/E3)

LOL
: Laughing Out Loud slang, Usenet, IRC

LOOPS
: LISP Object Oriented Programming System (Xerox, OOP)

LOP
: Loss of Pointer (UNI)

LOS
: Local Operating System

LOS
: Loss of Signal (UNI, ATM)

LOSP
: LOw SPeed channel connector (Cray, I/O)

LOVIS
: LagerOrtVerwaltungs- und InformationsSystem (MBAG)

LP
: Line Printer (Unix)

LP
: Linear Programming

LP

Logical Partition (LVM)

LPB

[S3] Local Peripheral Bus

LPC

Linear Predictive Coding voice processing

LPD

Line Printer DAEMON

LPDA

Link Problem Determination Aid

LPDP

Line Printer DAEMON Protocol (RFC 1179)

LPDU

Link Protocol Data Unit

LPEX

Live Parsing eXtensible Editor (IBM, OS/2)

LPF

League for Programming Freedom org.

LPI

Lines Per Inch

LPP

Licensed Program Product (IBM)

LPR

Low Priority Request (VUMA)

LPS

Lines Per Second

LPT

Line PrinTer

LPX

??? [motherboard]

LQ

Letter Quality

LQM

Link Quality Monitoring (PPP)

LR

Location Register (LA, GSM)

LRC

Longitudinal Redundancy Check

LRPC
: Lightweight Remote Procedure Calls (OLE)

LRS
: Line Repeater Station

LRU
: Last Recently Used

LS
: LAN Server (IBM)

LSA
: Link State Advertisement (OSPF)

LSAPI
: Licensed Services Application Program Interface (MS, API)

LSB
: Least Significant Bit

LSE
: Local Subscriber Environment

LSF
: Load Sharing Facility

LSI
: Large Scale Integration

LSL
: Link Support Layer (ODI)

LSM
: Linux Software Map (Linux)

LSN
: Logical Sector Numbers (OS-9)

LSR
: Leaf Setup Request

LST
: Linux Support Team [distribution] (Linux)

LSU
: LAN Service Unit (LAN)

LT
: Logical Terminal (IBM)

LT
: Lower Tester (ISO 9646-1)

LTC

TOC

Line Termination Coordinator

LTC

Longitudinal Time Code (Video)

LTE

Line Terminating Equipment (SONET)

LTID

Logical Terminal IDentifier (IBM)

LU

Logical Unit (NAU)

LU62

Logical Unit 6.2 (IBM), "LU6.2"

LUG

Local Users Group

LUN

Logical Unit Number (SCSI)

LUNA

Leuchtendatei fuer UnfallfluchtNAchforschungen (INPOL)

LUNI

LANE User Network Interface (LANE, ATM)

LUT

Look-Up Table (RAMDAC)

LV

Logical Volume (LVM)

LVD

Low Voltage Differential [technology] (SCSI, Symbois Logic)

LVDS

Low Voltage Differential Signal

LVM

Logical Volume Manager (AIX, HP/UX, OSF/1, HDD)

LVN

LandesVerwaltungsNetz (Baden-Wuerttemberg)

LVTTL

Low Voltage Transistor Transistor Level (IC)

LW

Living Worlds (VRML)

LWC

Last Working Configuration (ESCD, PNP, BIOS)

LWL

LichtWellenLeiter cable

LWP

Light Weight Process (Sun, OS)

LX

Linear eXecutable (OS/2)

LZS

Lempel-Ziv-Stac [compression]

LZSDCP

[PPP] Lempel-Ziv-Stac - Data Compression Protocol (PPP, RFC 1967), "LZS-DCP"

LZW

Lempel-Ziv-Welch [compression]

Go to the previous, next section.

Go to the previous, next section.

-- M ---

MAC
: Mandatory Access Control (MLS)

MAC
: Media Access Control (ISO, OSI)

MAC
: Message Authentication Code SSL, SRT, cryptography

MACH
: Multilayer ACtuator Head

MACOS
: MACintosh Operating System (Apple, OS), "MacOS"

MACS
: Manufacturing Application Control System (SNI)

MACS
: Modem Access Control System (MODEM, DES), "M.A.C.S."

MACSBUG
: Motorola Advanced Computer Symbolic deBUGger (Motorola)

MAD
: Memory Address Driver strength (BIOS)

MAD
: Message Address Directory

MAD
: Militaerischer AbschirmDienst mil., Org.

MADE
: Multimedia Application Development Environment (CWI)

MAE
: Macintosh Application Environment (Apple, Sun, HPUX)

MAINSAIL
: MAchine INdependent SAIL (SAIL)

MAJOUR
: Modular Application for JOURnals (EWS, SGML)

MAN
: Metropolitan Area Network

MAP
: Maintenance Analysis Procedure (IBM)

MAP
: Manufacturing Automation Protocol (General Motors)

MAP
: Mobile Application Part (MSC, GSM)

MAP27
: Mobile Access Protocol [for MPT 1327] (MPT 1327), "MAP 27"

MAPASE
: Mobile Application Part - Application Service Elements (MAP, MSC, GSM), "MAP-ASE"

MAPI
: Messaging Application Program Interface (MS, WOSA, API)

MAPTOP
: Manufacturing Automation Protocol/Technical Office Protocol, "MAP/TOP"

MAR
: Microprogram Address Register

MAS
: Multi Agent Systems (AI)

MAS90
: Mittelstands-Anwendungs-System 90 (IBM)

MASE
: Message Administration Service Element

MASM
: Microsoft ASseMbler MS, assembler

MAU
: Medium Access Unit

MAU
: Medium Attachment Unit (IEEE 802.3, Transceiver)

MAU
: Multistation Access Unit (Token Ring, Hub)

MAUS
: Muensters Apple User Service BBS, network, "M.A.U.S."

MAW
: Microsoft At Work

MAWI
: MAterialWIrtschaft

MAX

TOC

MAssively parallel uniX (Cray, OS, MIMD, MPP, Unix)

MAX

Media Access Exchange (Ascend)

MB

MailBox

MB

MegaByte

MBAG

Mercedes-Benz AktienGesellschaft (Anwender)

MBCMS

Mercedes-Benz-Computer-Mikrofilm-System (MBAG), "MB-CMS"

MBPS

MegaBits Per Second

MBS

Maximum Burst Size

MBUS

Module BUS [standard] (Sun, SPARC), "MBus"

MCA

Micro Channel Architecture (IBM, PS/2)

MCA

Mission Critical Applications

MCAV

Modified Constant Angular Velocity

MCB

Memory Control Block (DOS, TPA)

MCBF

Mean Cycle Between Failure

MCC

Manchester Code Converter

MCC

Manchester Computing Centre (Linux)

MCC

Mitteldeutsches Communication Center (Leipzig)

MCD

Mini Client Driver MS, Windows NT

MCD

Multimedia Cartridge Drive (Nomai)

MCEB
: Military Communications-Electronics Board Org., mil., USA

MCF
: Meta Content Framework / Format (Apple, WWW)

MCGA
: Multi Color Graphics Adapter (IBM, PS/2)

MCI
: Measurement Layer Interface (UMA)

MCI
: Media Control Interface

MCI
: Microwave Communications Incorporated

MCL
: Media Communication Lab org., Uni Boston, USA

MCM
: MultiChip Module (CPU)

MCN
: Metropolitan Campus Network

MCNS
: ??? [cable modem standard] org.

MCP
: Master Control Program

MCP
: Multiport / Multiprotocol Communication Processor

MCPAS
: Master Control Program/Advanced System, "MCP/AS"

MCPC
: Multi Channel Per Carrier

MCPS
: Microsoft Certified Product Specialist (MS)

MCR
: Minimum Cell Rate (UNI, ATM, PCR, ABR)

MCS
: Material Control System

MCS
: Message Conversion System

MCS

MultiCast Server

MCS

Multichannel Communications System (Mac)

MCS

Multivender Customer Services (DEC)

MCSD

Microsoft Certified Solution Developer (MS)

MCSE

Microsoft Certified System Engineer (MS)

MCSMP

Message Conversion System Message Processor (MCS)

MCSS

Mitac Computer Security System (Mitac)

MCTD

Mean Cell Transfer Delay (UNI, ATM)

MCU

Management & Cascade Unit (Hub)

MCU

Micro Control Unit

MCU

Multipoint Control Unit

MCVA

Modified Constant Angular Velocity (Optical Disk)

MD

Make Directory (DOS, OS/2)

MD

Management DOMAIN

MD

Mini Disk (Sony)

MD2

Message Digest [algorithm] 2 (RFC 1115/1319)

MD4

Message Digest [algorithm] 4 (RFC 1320)

MD5

Message Digest [algorithm] 5 (RFC 1321)

MDA

Medicated Digest Authentication (HTTP)

MDA
: Monochrom Display Adapter (IBM, PC)

MDBMS
: Multidimensional DataBase Management System (DBMS, DB)

MDBS
: Micro Data Base System (DB)

MDC
: Message conversion system directory Component (MCS)

MDE
: ??? DatenErfassung

MDI
: Medium Dependent Interface ethernet

MDI
: Mobile Data Initiative org., GSM

MDI
: Multiple Document Interface (MS, Windows)

MDI
: Multiplex Device Interface

MDL
: Microstation Development Language (CAD)

MDRAM
: Multibank Dynamic Random Access Memory (RAM, DRAM, IC)

MDRC
: Manufacturing Design Rule Checker

MDS
: Manufacturing Design System

MDSE
: Message Delivery Service Element

MDT
: Mechanical DeskTop (SGI)

MDT
: Message Distribution Terminal

MDT
: Mittlere DatenTechnik

MDT
: Mountain Daylight Time [-0600] (TZ, MST, USA)

ME

Mapping Entity

MEG

Mega Evil Grin slang, Usenet, IRC

MEGA

MessdatenErfassung und Graphische Auswertung

MEL

Maya Embedded Language

MEMS

Micro-ElectroMechanical Systems

MERS

Most Economic Route Selection

MES

Manufacturing Execution System

MES

Menue-EntwicklungsSystem (Sinix, SNI)

MESCH

Multi-WAIS Engine for Searching Commercial Hosts (WAIS, WWW)

MESH

Macintosh Enhanced SCSI Hardware (Apple, SCSI)

MESI

Modified - Exclusive - Shared - Invalid (SMP)

MESZ

MittelEuropaeische SommerZeit [+0200] (TZ)

MET

Memory Enhancement Technology (HP), "MEt"

MET

Middle European Time [+0100] (TZ, CET, METDST, MEZ)

METDST

Middle European Time Daylight Saving Time [+0200] (TZ, MET, MEZ)

MEZ

MittelEuropaeische [sommer] Zeit [+0200] (TZ)

MF

Multi Frequency

MFAST

Mwave Folded Array Signal Transform [DSP] (IBM, DSP)

MFC

Microsoft Foundation Classes

TOC

MFC
: Music Feature Card (IBM, Yamaha, MIDI)

MFENET
: Magnetic Fusion Energy NETwork

MFG
: Mit Freundlichen Gruessen [= best wishes]

MFG
: Multi-Function Gateway

MFI
: MainFrame Interactive (IBM)

MFII
: Multi-Functional [keyboard] II (IBM)

MFIP
: Multi-Function Interoperability Processor

MFKS
: MultiFunktionales KonferenzSystem

MFLOPS
: Million FLoating-point Operations Per Second (CPU)

MFM
: Modified Frequency Modulation

MFP
: MultiFunction Product

MFS
: Macintosh File System (Apple)

MFS
: Message Format Service (IBM, IMS)

MFS
: Mobile File Sync (IBM)

MFV
: MehrFrequenzwahlVerfahren

MGA
: Matrox Graphics Adapter (Matrox)

MGA
: Monochrome Graphics Adapter

MGI
: Multi-Function Interpreter

MGM

Memory Grant Manager (Informix, DB)

MH

Mobile Host (MHP)

MH

Modified Huffman (Fax)

MHDL

MIMIC Hardware Description Language

MHEG

Multimedia and Hypermedia information coding Expert Group (JTC1, ISO)

MHP

[columbia] Mobile Host Protocol

MHPCC

Maui High Performance Computing Center org., USA

MHS

Message Handling System (GOSIP, X.400, Novell, SPX, IPX)

MHTML

MIME [e-mail encapsulation of aggregate documents, such as] HTML (MIME, HTML, RFC 2110)

MIAW

Movie In A Window

MIB

Management Information Base (OSI, SNMP)

MIC

Media Interface Connector (FDDI, PMD)

MIC

Memory in Cassette (Seagate, EEPROM)

MIC

Message Integrity Check / Code

MIC

Multiple Interface Connection (Kyocera)

MICA

MODEM ISDN Channel Integration (Telebit, MODEM, ISDN)

MICE

Modular Integrated Communications Environment

MICO

MICO Is CORBA (ORB, CORBA, Uni Frankfurt)

MID

Message IDentifier (ATM)

MIDI

Musical Instruments Digital Interface

MIDL

Microsoft Interface Definition Language (NT)

MIF

Module Interconnection Facility (Proteus)

MIG

Mach Interface Generator (Mach)

MII

Media Independent Interface

MIL

Matrox Imaging Library

MILNET

MILitary NETwork USA, mil., Netzwerk

MILSTAR

Military Strategic TActical Relay mil., USA

MILSTD

Military STandarD mil., USA, "MIL-STD"

MIMD

Multiple Instruction [stream], Multiple Data [stream]

MIME

Multipurpose Internet Mail Extensions (RFC 2045/2046/2047/2048/2049, IETF)

MIMOLA

Machine Independent MicrOprogramming LAnguage (HDL)

MIN

Multistage Interconnection Networks

MINISTREL

Models for INformatIon STorage and REtrievaL (OA, BIS, ESPRIT)

MINT

Mint is Not TOS (Atari), "MiNT"

MINT

Multimedia-kommunikation aif Integrierten Netzen und Terminals

MINX

Multimedia Information Network eXchange

MIO

Memory Input/Output (Motorola)

MIO

Modular Input/Output (HP)

MIPS

Microprocessor without Interlocked Piped Stages

MIPS

Million Instructions Per Second (CPU)

MIR

Maximum Information Rate

MIR

Micro-Instruction Register

MIS

Management Information System

MISS

Mecklenburg Internet Service System (ISP)

MISSI

Multilevel Information System Security Initiative org.

MISX

Metered Services Information eXchange (Internet)

MIT

Management Information Tree

MIT

Massachusetts Institute of Technology org., USA

MJ

Modular Jack

MKS

Mortice Kern Systems manufacturer

ML

Mail List

ML

Meta Language

MLA

Mail List Agent

MLC

MultiLevel Cell (IC, PCMCIA, Flash, Intel)

MLE

MultiLine Entry field (IBM, OS/2)

MLI
: Measurement Layer Interface (UMA)

MLI
: Multiple Link Interface (ODI)

MLIA
: [symposium on] Machine Learning in Information Access (AAAI, AI, conference)

MLID
: Multiple Link Interface Driver (ODI, LAN)

MLM
: Mailing List Manager [software]

MLMA
: Multi-Level Multi-Access (MAC)

MLP
: MultiLink Point-to-Point Protocol (PPP)

MLR
: Multi-channel Linear Recording [technology] (Tandberg)

MLS
: MultiLevel Secure [operating systems / platforms]

MLT
: Master Lower Tester (ISO 9646-3, TTCN)

MLT3
: MuLTilevel-3 [encoding] (Schneider & Koch, FDDI), "MLT-3"

MM
: Media Manager (Novell, Netware, SMS)

MM
: Military Message mil.

MM
: Mobile Management (RR, CM, GSM)

MMA
: Microcomputer Managers Association org., USA

MMA
: MIDI Manufacturer Association org., IMA

MMC
: MIDI Machine Control

MMC
: MultiMedia Comands (SAM)

MMCD

MultiMedia Compact Disk (Sony, Philips, CD)

MMCDE

MultiMedia Compact Disk - Erasable (Sony, Philips, CD), "MMCD-E"

MMCF

MultiMedia Communications Forum org.

MMDO

Magnetic Modulation Direct Overwrite

MMDS

Multi-channel, Multi-point Distribution System

MMF

Make Money Fast (Usenet, ECP, EMP)

MMF

Memory Mapped File (Windows 95, Windows NT)

MMF

Multimode Fiberoptic cable

MMFS

Manufacturing Message Format Standard (MAP)

MMHS

Military Message Handling System mil., MHS

MML

Maker Markup Language (FrameMaker)

MMPM2

MultiMedia Presentation Manager /2 (IBM, OS/2), "MMPM/2"

MMR

Modified Modified Read (Fax)

MMS

Manufacturing Message Specifications / Standard (MAP, EIA, RS-511, ISO, DIS 9506)

MMS

Maximum Message Size

MMS

Module Making System (VMS)

MMS

Multilevel Mail Server

MMU

Mass Memory Unit

MMU

Memory Management Unit

MMVF
: MultiMedia Video File [disk] (NEC, DVD)

MMWM
: MultiMedia Window Manager (X-Windows)

MMX
: MultiMedia eXtensions (Intel, CPU)

MNG
: Multiple-image Network Graphics [format]

MNOS
: Metal Nitride Oxide Semiconductor (IC)

MNP
: Microcom Networking Protocol

MO
: Managment Object (OSI)

MOCS
: Microsoft Official Curriculum Seminar (MS, ATEC, MCSD)

MOD
: Magneto-Optical Disk (OD)

MODACOM
: MObile DAta COMmunication

MODCA
: Mixed Object Document Content Architecture (IBM), "MO:DCA"

MODEM
: MOdulator DEModulator

MOE
: Measure of Effectiveness

MOLAP
: Multidimensional OnLine Analytical Processing (OLAP)

MOLP
: Microsoft Open Licence Pak (MS)

MOM
: Message Orientated Middleware (IBM)

MONALISA
: MOdelling NAturaL Images for Synthesis and Animation, "MONA LISA"

MONET
: high data rate MObile interNET network

MOO

MUD Object Orientated (Internet, OOP, MUD)

MOP
: Maintenance Operations Protocol (DEC)

MOP
: Meta Object Protocol (CLOS)

MOP
: Multiple Original Prints

MOPS
: Million Operations Per Second

MOS
: Metal Oxide Semiconductor (IC)

MOSES
: Major Open Systems Environment Standards

MOSFET
: Metal Oxide Semiconductor Field Effect Transistor (IC)

MOSP
: Microsoft Online Services Partnership (MSN)

MOSPF
: Multicast [extensions to] Open Shortest Path First [routing] (IP, OSPF, RFC 1584)

MOST
: Mobile Open Systems Technologies (UK, Uni Lancaster)

MOT
: Means of Test

MOTIS
: Message-Orientated Text Interchange System (ISO 10021, JTC1)

MOTSS
: More Of The Sameold Sameold (Usenet)

MOU
: Memorandum Of Understanding (IPOC, TLD, Internet)

MOVE
: Microsoft Overlay Virtual Environment (MS)

MOVI
: [projekt] MObile VIsualisierung (DFG)

MOWORM
: Magneto-Optical Write Once Read Many, "MO-WORM"

MP
: [PPP] Multilink Protocol (MPPP, RFC 1990)

MP
: Multi Processor

MP
: MultiProtocol

MP3
: MPEG audio layer 3 (MPEG, audio)

MPC
: MPOA Client (MPOA, ATM)

MPC
: Multimedia Personal Computer

MPD
: Message Preparation Directory

MPDA
: MultiPlattform DiskArray

MPDU
: Message Protocol Data Unit (PDU)

MPEG
: Motion Picture Expert Group org., ISO 11172-1, JTC1

MPEXL
: ??? (HP), "MPE/XL"

MPI
: Message Passing Interface (SMP)

MPI
: Multiprozessor Interconnect Bus

MPIF
: Message Passing Interface Forum manufacturer

MPM
: Metra Potential Method

MPM2
: MultiMedia Presentation Manager/2 (OS/2, IBM), "MPM/2"

MPML
: Main Profile @ Main Level (MPEG), "MP@ML"

MPOA
: Multi-Protocol Over ATM (ATM)

MPOW
: Multiple Purpose Operator Workstation

MPP

Massive Parallel Processor / Processing

MPP

Message Posting Protocol (RFC 1204)

MPP

Message Processing Program

MPPC

Microsoft Point-to-Point Compression (MS, RFC 2118)

MPPD

Multi Purpose Peripheral Device

MPPP

Multilink Point-to-Point Protocol (PPP, RFC 1990, MP, Ascend)

MPR

[National Board for Metrology and Testing] org., Schweden

MPS

Megabytes Per Second

MPS

MPOA Server (MPOA, ATM)

MPS

MultiPage Signal (Fax)

MPS

MultiProzessor Spezifikation (SMP)

MPSX

Mathematical Programming System eXtended (IBM)

MPT

Ministery of Post and Telecommunications org., Japan

MPT

MultiPort Transceiver

MPT1327

???

MPTA

Multi Protocol Transport Architecture (IBM, SNA)

MPTN

Multi Protocol Transport Networking (IBM)

MPTS

Multi Protocol Transport Services (IBM, MPTN)

MPTS2

Multi Protocol Transport Services / 2 (IBM, OS/2), "MPTN/2"

MPU
: MicroProcessor Unit

MPU401
: MIDI Processing Unit 401 (Roland, MIDI), "MPU 401"

MPW
: Macintosh Programmers Workshop (Apple)

MQI
: Message Queueing Interface (IBM)

MQUIPS
: Million QUality Improvements Per Second

MR
: Magneto - Resistive (HDD)

MR
: MODEM Ready (MODEM)

MR
: Modified Read (Fax)

MRA
: Multi Resolutions Analysis

MRB
: Method Request Broker (OMG)

MRBC
: Multiple Resolution Bitmap Compiler (MS, Windows)

MRCF
: Microsoft Realtime Compression Format (MS)

MRCI
: Microsoft Realtime Compression Interface (MS)

MRCS
: Multi-Rate Circuit Switching

MRJ
: Macintosh / MacOS Runtime for Java (Java, Apple)

MRNET
: Minnesota Regional NETwork network, USA

MRO
: Mandatory Router Option [flag] (CATNIP)

MROC
: Multicommand Required Operational Capability

MROM

??? Read Only Memory

MRP
: Material Requirement Planning (PPS, CIM)

MRS
: Media Recognition System (DAT)

MRSE
: Message Retrieval Service Element

MRTG
: Multi Router Traffic Grapher

MRU
: Maximum Receive Unit [size] (SLIP, PPP)

MRU
: Most Recently Used

MRX
: MagnetoResistive eXtended [technology] (HDD)

MS
: Message Store

MS
: Meta Signaling (ATM, ???)

MS
: MicroSoft manufacturer

MS
: Mobile Station (GSM)

MSA
: Management Service Architecture

MSAP
: Management Service Access Point (OSI, OSI/RM, SAP)

MSAP
: Mini Slotted Alternating Priorities (MAC)

MSB
: Most Significant Bit

MSC
: MicroSoft C

MSC
: Mobile services Switching Center (PLMN, GSM)

MSCDEX
: MicroSoft Compact Disk EXtensions (CD)

TOC

MSCHAP
: MicroSoft Challenge Handshake Authentication Protocol (MS), "MS-CHAP"

MSCM
: Multiple Slots on Continuation Mechanism (DQDB)

MSCP
: Mass Storage Control Protocol

MSD
: Most Significant Digit

MSDN
: Macintosh Software Distribution Network (FidoNet, Apple)

MSDN
: MicroSoft Developer Network (Internet, MS)

MSDN
: MicroSoft Developer Network (MS)

MSDOS
: MicroSoft Disk Operating System (MS, OS)

MSE
: MicroSoft Exchange server (MS)

MSFIS
: MicroSoft Fax Information Service (MS)

MSFP
: Management Service Focal Point (IBM)

MSI
: Medium Scale Integration

MSI
: Micro-Star International manufacturer, Taiwan

MSIE
: MicroSoft Internet Explorer (MS, WWW)

MSISDN
: Mobile Station ISDN Number (PLMN, GSM), "MS-ISDN"

MSL
: Maximum Segment Lifetime (TCP/IP)

MSL
: Microsoft Software Library (Internet, MS)

MSL
: Mirrored Server Link

MSM

Media Support Module (ODI)

MSN

Microsoft Support Network (Internet, MS)

MSN

Monitoring cell Sequence Number (UNI, ATM)

MSN

Multiple Subscriber Number (Euro-, ISDN)

MSNF

Multiple Systems Networking Facility (IBM)

MSP

Media Signal Processor (Samsung, MMX, ARM)

MSP

Message Security Protocol

MSP

Message Send Protocol (RFC 1159)

MSPI

M&T Software Partner International [gmbh]

MSR

Machine Specific Register

MSR

Mess-, Steuer- und Regelungssysteme

MSR

Mobile Support Router (MHP)

MSR

Model Specific Registers (Intel, Pentium)

MSRN

Mobile Station Roaming Number (MM, HLR, MS-ISDN, GSM)

MSRPC

MicroSoft Remote Procedure Call (MS, DCOM), "MS-RPC"

MSS

MIMOLA Software System (MIMOLA)

MSSE

Message Submission Service Element

MST

Mountain Standard Time [-0700] (TZ, MDT, USA)

MSVC

Meta-Signaling Virtual Channel (ATM)

MSVC
: MicroSoft Visual C++ (MS)

MSW
: Machine Status Word

MT
: Machine [assisted] Translation

MT
: Mannesmann Tally [gmbh] manufacturer

MT
: Message Transfer / Type

MT
: Mobile Termination (GSM)

MT
: Moluccas Time [+0830] (TZ)

MTA
: Message Transfer Agent (MTS, OSI, X.400)

MTBF
: Mean Time Between Failure

MTBRP
: Mean-Time-Between-Parts-Replacement

MTC
: Master Test Component (ISO 9646-3, TTCN)

MTC
: MIDI Time Code (MIDI)

MTD
: Memory Technology Drivers

MTDA
: Mean Time Data Availability

MTE
: MuTating Engine (Viren), "MtE"

MTEC
: Motorola Training and Education Center org., Chicago

MTF
: Message Text Formats

MTF
: Microsoft Tape Format (MS)

MTF

Modulation Transfer Function

MTI

MIPS Technologies Inc. manufacturer, SGI

MTP

Message Transfer Part (ISDN, GSM)

MTRR

Memory Type Range Register (CPU, Pentium Pro)

MTS

Message Transfer System (MHS)

MTTF

Mean Time To Failure

MTTFF

Mean Time To First Failure

MTTR

Mean Time to Recovery / Repair / React

MTTR

Mean Time Trouble Repair

MTU

Maximum Transmission Unit (SLIP, PPP)

MTU

Message Transfer Unit

MUA

Mail User Agent

MUBIS

MUltimediales BueroInformationsSystem (OA, BIS, TU Braunschweig), "MuBIS"c

MUCK

Multi-User Chat Kingdom (MUD)

MUD

Multi-User Dungeon (MUD)

MUGD

MUMPS User Group Deutschland MUMPS, user group, "MUG-D"

MUI

Management User Interface (OMF, UI)

MUI

Multimedia User Interface (SGI, UI)

MULDEM

MULtiplexer-DEMultiplexer

MULE
: MULtilingual Enhancement of GNU EMACS (EMACS, GNU)

MULTICS
: MULTiplexed Information and Computing Service (OS)

MULTOS
: MULtimedia Office Server (OA, BIS, ESPRIT)

MUMPS
: Massachusetts general hospital Multi-Programming System

MUPAD
: Multi Processing Algebra Data [tool] Uni Paderborn, Germany, CAS, "MuPAD"

MUSH
: Multi-User Shared Hallucination (MUD)

MUSICAM
: Masking-pattern adapted Universal Subband Integrated Coding And Multiplexing MPEG, digital audio

MUT
: Master Upper Tester (ISO 9646-3, TTCN)

MUTOS
: MultiUser-/multitasking Operating System GDR, OS

MUEW
: eW MonopolUEbertragungsWeg, "MueW"

MUX
: MUltipleXer

MVA
: Multidomain Vertical Alignment [technology] (LCD, Fujitsu)

MVC
: Model View Controller (Smalltalk, GUI)

MVI
: Motion Video Instructions (DEC, CPU, Alpha)

MVIF
: Multi-Vendor Interacting Forum org., IN

MVP
: Most Valuable Professional [bonus program] (MS)

MVRCA
: Magnalink Variable Resource Compression Algorithm (PPP, RFC 1975)

MVS
: Multiple Virtual Storage (IBM, OS)

MVS
: Multiple Virtual System (OS)

MVSESA
: Multiple Virtual Storage/Extended System Architecture (IBM), "MVS/ESA"

MVSESASP
: Multiple Virtual Storage/Extended System Architecture System Product (IBM), "MVS/ESA SP"

MVSSP
: Multiple Virtual Storage/System Product, "MVS/SP"

MVSTSO
: Multiple Virtual Storage/Time Sharing Option (IBM), "MVS/TSO"

MVSXA
: Multiple Virtual Storage/eXtended Architecture, "MVS/XA"

MWS
: Management WorkStation

MWS
: Matsushita White Skipping (Fax)

MX
: Mail eXchange (Unix)

MXB
: Multimedia eXension Board (SNI)

MXC
: Multimedia eXtension Connector

MZ
: Mark Zbikowski (MS-DOS, MCB)

Go to the previous, next section.

Go to the previous, next section.

-- N ---

NAARS
: National Automated Accounting Research System (USA)

NAB
: National Association of Broadcasters org., USA

NAB
: Netware Asynchronous Board (Novell, Netware)

NAC
: Network Administration Center

NAC
: Networks and Communications

NAC
: Null Attached Concentrator (FDDI)

NACLP
: North American Conference on Logic Programming (USA)

NACM
: Networks And Communications Marketing

NACS
: National Advisory Committee on Semiconductors org., USA

NACS
: Netware Asynchronous Communication Server (Novell, Netware)

NACS
: Network Access Control System (Netware, DES)

NACSIS
: National Academic Center for Science Information Systems org., USA

NACT
: Neural Adaptive Control Technology [project]

NAD
: Network Access Device

NADB
: National Archeological DataBase (DB, USA)

NAEC
: Novell Authorized Education Center (Novell, Netware)

NAG
: National Algorithms Group [ltd] UK, org., "NAg"

NAG
: Network Architecture Group org.

NAI
: Netzwerk Arbeitswelt Informatik manufacturer

NAL
: Netware Application Launcher (Novell, Netware, NAM)

NALIS
: Nevada Academic Libraries Information System

NAM
: Netware Application Manager (Novell, Netware, NAL)

NAMS
: National Association of Multimedia Shareware org., USA

NAP
: Network Access Point (IN)

NAPLPS
: North American Presentation Level Protocol Syntax (BBS)

NAPT
: ???

NARP
: Non-Broadcast Multiple Access Address Resolution Protocol (RFC 1735)

NAS
: Netware Access Services (Novell, Netware)

NAS
: Network Application Services (DEC)

NAS
: Network Application Support

NASI
: Netware Asynchronous Service Interface (Novell, Netware)

NAT
: [IP] Network Address Translator (RFC 1631, IP)

NATOA
: National Association of Telecommunications Officers & Advisors org., USA

NAU
: Network Addressable Unit (IBM, SNA, OSI)

NAU

TOC

Network Attachment Unit GigaB, IP-router

NAUN

Nearest Active Upstream Neighbour (MAC)

NAVNET

NAVy NETwork mil., USA, Netzwerk

NB

Nota bene slang, Usenet, IRC

NBCD

Natural Binary Coded Decimal

NBE

Not-Below-or-Equal

NBFCE

NETBIOS Frames Control Program (NETBIOS)

NBFCP

[PPP] NETBIOS Frames Control Protocol (PPP, NETBIOS, RFC 2097)

NBFM

Narrow Band Frequency Modulation

NBMA

Non-Broadcast Multiple Access (UNI, ATM)

NBP

Name Binding Protocol (AppleTalk)

NBS

National Bureau of Standards org., Vorlaeufer, NIST

NBT

NETBIOS on TCPIP (MS)

NC

Network Channel / Connect / Control

NC

Network Co-ordinator (FidoNet)

NC

Network Computer [reference profile] (Apple, IBM, Netscape, Oracle, Sun, Internet)

NC

Norton Commander (Symantec)

NC

Numerical Control

NCA

Network Communications Adapter

TOC

NCA
: Network Computing Architecture (Oracle)

NCA
: Network Control Analysis

NCA
: Novell Certification Alliance (Novell, Netware)

NCAIR
: National Center for Automated Information Research org., USA

NCB
: Network Control Block (LAN)

NCC
: Network Control / Coordination Center

NCC
: Network Control Computer

NCCF
: Network Communications Control Facility (IBM)

NCCS
: NASA Center for Computational Sciences org., NASA

NCD
: Network Computing Devices manufacturer

NCD
: Norton Change Directory (DOS)

NCE
: Nomadic Computing Environment (Tadpole)

NCGA
: National Computer Graphics Association org., USA

NCHPC
: National Consortium for High Performance Computing org., HPC, USA

NCI
: Network Channel Interface

NCI
: Non Coded Information

NCIA
: Native Client Interface Architecture (IOS)

NCIP
: Novell Certified Internet Professional (Novell, WWW, CNA)

NCMOS

N-channel [Silicon Gate Reversed] CMOS

NCN

Nixdorf Communications Network

NCOHPCC

National Coordination Office for High Performance Computing and Communications org., USA, HPC, "NCO/HPCC"

NCOS

Network Computer Operating System (OS, Oracle, Internet)

NCP

Netware Core Protocol (Novell, IPX)

NCP

Network Control Processor

NCP

Network Control Program (BBN, ARPANET)

NCP

Network Control Program / Point (IBM)

NCP

Non-Carbon Paper

NCP

Not Copy-Protected

NCPE

Netware Core Protocol Extension (NCP, Netware, IPX)

NCPIE

National Council on Patient Information and Education org., USA

NCR

National Cash Registers manufacturer, AT&T

NCS

National Communications System (USA)

NCS

Network Computing System (HP, Apollo)

NCS

Network Control System

NCSA

National Center for Supercomputing Applications org., USA

NCSC

National Computer Security Center org., USA

NCSC

TOC

North Carolina Supercomputing Center org., USA

NCSI

Network Communications Service Interface (NMP)

NCSL

National Computer Systems Laboratory NIST, org., USA

NCSL

National Conference of Standards Laboratories org., USA

NCSNDR

Network Computing System Network Data Representation (HP, Apollo), "NCS NDR"

NCSS

Non Commentary Sources Statements (LOC)

NCSS

Number Crunching Statistical System

NCT

Network Control and Timing

NCTE

Network Channel-Terminating Equipment

NCTL

National Computer and Telecommunications Laboratory org., USA

NDA

Network Delivery Access

NDA

Non-Disclosure Agreement

NDA

Norddeutsche DatenAutobahn network

NDBMS

Network DataBase Management System (DB)

NDBS

Non-standard DataBase System (DB)

NDC

National Destination Code (MS-ISDN, GSM)

NDC

Network Data Collection

NDC

Node Data Controller (Zenith)

NDCC

Network Data Collection Center

NDDL

Neutral Data Definition Language (DDL)

NDE

NeWS Development Environment

NDI

Network Distributed ISDN [for windows NT] (AVM, ISDN, Windows NT)

NDIS

Network Driver Interface Specification (3COM, MS)

NDL

Network Database Language (DB, 4GL)

NDMP

Network Data Management Protocol

NDMS

Netware Distributed Management Services (Novell, Netware)

NDP

Numeric Data Processor

NDPA

Network Problem Determination Application

NDR

Network Data Representation (NCS, DCE)

NDR

Network Data Representation service (DCE/RPC)

NDR

Non-Destructive Read

NDRO

Non-Destructive ReadOut

NDS

Netware Directory Services (Novell, Netware)

NDS

Network Data System

NDT

Net Data Throughput

NDT

Newfoundland Daylight Time (TZ, NFT)

NDT

Non-Destructive Testing

TOC

NDU
: Network Device Utility

NE
: Network Element

NEA
: ??? [protocol stack on OSI transport layer]

NEAR
: National Electronic Accounting and Reporting

NEARNET
: New England Academic and Research NETwork USA, network, "NEARnet"

NEAT
: New Enhanced Advanced Technology (AT)

NEAT
: Novell Easy Administration Tool (Novell, Netware)

NEC
: National Electrical Code (USA)

NEC
: Nippon Electronic Corporation manufacturer

NED
: NASA Extragalactic Database (DB)

NEDO
: New Energy and industrial technology Development Organization org., Japan

NEFS
: Network Extensible File System, "NeFS"

NEII
: National Engineering Information Initiative org., USA

NEM
: Nothing Else Matters slang, Usenet, IRC

NEP
: Never-Ending Program

NEREN
: NEbraska Research and Education Network USA, network

NES
: News Electronic Service

NESPINN
: NEurocomputer fuer Spikende Neuronale Netze (TUB)

NEST

Netware Embedded Systems Technology (Novell, Netware)

NET

Network Entity Title

NETBEUI

NETBIOS Extended User Interface (UI)

NETBIOS

NETwork Basic Input Output System (IBM, RFC 1001/1002), "NetBIOS"

NETBLT

NETwork BLock Transfer (IP)

NETCDF

NETwork Common Data Format, "NetCDF"

NETDA

NETwork Design and Analysis

NETPARS

NETwork Performing Analysis Reporting System

NETSS

National Electronic Telecommunication Surveillance System (USA)

NEWS

Netware Early Warning System (Novell, Netware)

NEWS

Networked Extensible Windowing System (Sun), "NeWS"

NEWT

NeWS Terminal, "NeWT"

NEXT

NEw eXtended Technology, "NeXT"

NFA

Name Field Address (Forth)

NFA

Non-determistic Finite-state Automation

NFAS

Non Facilities-Associated Signaling (ISDN, PRI)

NFR

Near Field Recording [technology]

NFS

Network Facilities Services

NFS

Network File System (Sun, Unix, RFC 1813)

NFSP
: Netware File Service Protocol (Novell, Netware)

NFT
: Network File Transfer (DNA, DEC)

NFT
: NewFoundland Time [-0330] (TZ, NDT)

NGDD
: New Generation Desktop Design (Mitsubishi)

NGE
: Not-Greater-or-Equal

NGI
: ??? [Niederlaendische Informatikgesellschaft] Org., Netherlands

NGS
: Non-Government Standard (USA)

NGSB
: Non-Government Standards Body

NHRP
: [NBMA] Next Hop Resolution Protocol (X.25, ATM, NBMA, IETF)

NHSE
: National HPCC Software Exchange (USA)

NI
: Network Interconnect / Interface

NI
: Normenausschuss Informationsverarbeitungssysteme org., DIN

NIA
: Network Information Access

NIAS
: Netware Internet Access Server (Novell, Netware)

NIC
: Network Information Center Internet, org.

NIC
: Network Interface Card / Controller

NIC
: Numeric Intensive Computing

NICD
: NIckel CaDmium [batterie], "NiCd"

NICE

TOC

Network Information and Control Exchange (DECNET)

NICOL

Network Information Center On-Line

NICOLAS

Network Information Center On-Line Aid System

NID

Namespace IDentifier (URN)

NIDR

Network Information Discover and Retrieval

NIDX

Network Intrusion Detection eXpert system (BELLCORE, XPS)

NIE

Newton Internet Enabler (Apple, PDA)

NIFTP

Network Independent File Transfer Program (FTP)

NIHCL

National Institute of Health [c++] Class Library (PD)

NII

National Information Infrastructure [program] USA, org.

NIIT

National Information Infrastructure Testbed (ISH, USA)

NIMH

NIckel Metal Hybrid [batterie], "NiMH"

NIMT

National Institute for Management Technology org., Irland

NIPT

[international symposium on] New Information Processing Technologies conference, MITI

NIR

Network Information Registry / Retrieval

NIS

Network Information Service (NSF)

NIS

Network Information System (Unix)

NISC

Network Information and Support Center

NISDN

Narrowband Integrated-Services Digital Network (ISDN), "N-ISDN"

NISI
: Network Information Services Infrastructure

NISO
: National Information Standards Organization org., USA

NISS
: National Information on Software and Services (USA)

NISSPAC
: NISS Public Access Collections (NISS)

NIST
: National Institute of Standards and Technology org., USA

NISYP
: Network Information System / Yellow Pages, "NIS/yp"

NITF
: National Imagery Transmission Format

NITOL
: Norway-net with IT for Open Learning network

NITS
: Network Independent Transport Service

NIU
: Network Interface Unit

NIU
: North american ISDN Users (USA, ISDN)

NIUF
: North american ISDN Users Forum USA, user group, ISDN

NJC
: Nordic Journal of Computing Finland

NJE
: Network Job Entry (BITNET, RSCS)

NJE
: Network Job Entry

NKS
: Network Knowledge Server

NKSR
: Non-Kernel Security Related (Unix)

NL
: Network Layer (ISO, OSI)

NL

New Line (ASCII)

NL

Number Lines (Unix)

NLDM

Network Logical Data Manager (IBM)

NLE

Not-Less-or-Equal

NLM

Netware Loadable Module (Novell, Netware)

NLP

Natural Language Processing

NLP

Non-Linear Programming

NLP

Normal Link Pulse ethernet, LAN

NLPID

Network Layer Protocol IDentifier (ATM, OSI)

NLQ

Near Letter Quality

NLRI

Network Layer Reachability Information

NLS

Native Language Support (HP)

NLS

Native Language System (OSF)

NLS

Network License Server

NLSP

Netware Link Services Protocol (Novell, Netware, IPX)

NLSP

Network Layer Security Protocol (ISO, IEC, ISO/IEC 11557)

NM

Network Management

NMA

Network Management Architecture (SNA)

NMAA

National Multimedia Association of America org., USA

NMC
: Network Management Center

NMCP
: Network Management Protocol

NMCS
: National Military Command System mil., USA

NMEA
: National Marine Electronics Association [protocol] org., USA, GPS

NMF
: Network Management Forum

NMI
: Non-Maskable Interrupt

NML
: National Media Laboratory org., USA

NML
: Network Management Layer (TMN)

NMOS
: Negative-channel Metal-Oxide Semiconductor (IC)

NMP
: Network MODEM Program

NMPF
: Network Management Productivity Facility (IBM)

NMS
: Network Management Station / System (Novell, Netware)

NMS
: Network Monitoring Station

NMSU
: New Mexico State University org.

NMT
: Nordic Mobile Telephone (Mobile Systems)

NMVT
: Network Management Vector Transport

NN
: Neural Network

NNI
: Network Node Interface (ATM)

NNI

Network to Network Interface

NNS

Netware Named Services (Novel, Netware)

NNSC

NSF Network Service Center org., NSF

NNTP

Network News Transfer Protocol (Internet, RFC 977, Usenet)

NNX

Network Numbering eXchange

NOA

Net On Air (Internet)

NOC

Network Operations Center

NOCS

Network Operations Center System

NODIS

NSSDC Online Data and Information Service (NSSDC)

NOI

Node Operator Interface

NOMA

National Online Media Association org., USA

NORGEN

Network Operations Report GENerator

NORMA

NO Remote Memory Access (OSF/1, Multi-Server)

NOS

Network Operating System

NOTIS

Network Operator Trouble Information System

NP

Network Performance

NPA

Network Performance Analyzer

NPA

Network Printer Alliance (IEEE 1284, IBM, Lexmark, Xerox)

NPA

Network Professional Association org., USA

NPAC
: Northeast Parallel Architectures Center org., USA, HPC

NPC
: Network Parameter Control

NPDA
: Network Problem Determination Application (IBM)

NPH
: No Parse Headers (HTML)

NPI
: Network Printer Interface

NPL
: Non-Procedural Language

NPM
: Network Performance Monitor

NPMS
: Named Pipes / Mail Slots (MS)

NPSI
: NCP Packet Switching Interface (IBM, NCP)

NPSI
: Network Protocol Service Interface

NPSS
: NASA Packet Switch System (NASA)

NPTN
: National Public Telecomputing Network network, USA

NPV
: Net Present Value

NQS
: Network Queuing System

NRC
: National Research Council org., USA

NRCLSE
: National Resource for Computers in Life Science Education (USA)

NREN
: National Research and Education Network USA, network

NRFD
: Not Ready for Data

NRM

Network Resource Management

NRN

National Research Network USA, network

NRN

No Reply Necessary slang, Usenet

NROFF

New Run-OFF (Unix)

NRS

Name Registration System

NRS

Novell Replication Services (Novell, Netware)

NRT

Non-Requesting Terminal

NRTVBR

Non-RealTime Variable Bit Rate (VBR, ATM), "nrt-VBR"

NRZ

Non-Return-to-Zero [recording]

NRZI

Non-Return-to-Zero, Invert to ones [encoding]

NS

Name Server (DNS, Unix)

NS

National Standard

NSA

National Security Agency org., USA

NSAI

National Standards Authority of Ireland org., Irland

NSAP

Network Service Access Point (OSI, OSI/RM, SAP)

NSAPA

Network Service Access Point Address (OSI, NSAP)

NSAPI

Netscape Server Application Programmer's Interface (Netscape, WWW, C/S, API)

NSB

National Science Board USA, org.

NSC

National Security Council org., USA

NSC
: National SemiConductor manufacturer

NSC
: Network Service Center

NSCA
: National institute for Supercomputing Applications

NSD
: National Security Directive (USA)

NSE
: Network Support Encyclopedia (Novell)

NSEC
: Network Switching Engineering Center

NSERC
: Natural Sciences and Research Council org., USA

NSF
: National Science Foundation org., USA

NSF
: Norges StandardiseringsForbund org., Norwegen

NSFIP
: NextStep Fuer IntelProzessoren, "NSfIP"

NSFNET
: National Science Foundation NETwork network, USA, Internet, "NSFnet"

NSG
: Network Services Group

NSI
: Name Service Independent (DCE/RPC, CDS)

NSI
: NASA Science Internet NASA, network

NSIS
: National Schengen Information System SIS, Europe

NSN
: NASA Science Network USA, network

NSP
: Native Signal Processing (Intel, CPU)

NSP
: Network Service Point

NSP

Network Service Provider

NSP

Network Services Protocol (DNA)

NSPMP

Network Switching Performance Measurement Plan

NSR

Non-Source Routed

NSRD

National Software Reuse Directory (USA)

NSS

Namespace Specific String (URN)

NSS

Nodal Switching Subsystem (NSFNET)

NSSA

Not So Stubby Area (OSPF, RFC 1587)

NSSDC

National Space Science Data Center org., USA

NST

Newfoundland Standard Time (TZ)

NST

North Sumatra Time [+0630] (TZ)

NSTAN

NebenSTellenANlagen (Telekom, CBX), "NStAn"

NSTC

National Science and Technology Council org., USA

NSTL

National Software Testing Lab org., USA

NSTS

National Secure Telephone System

NT

Netzwerk Terminator, Network Terminator (ISDN)

NT

New Technology (MS, OS)

NT

Nome Time [-1100] (TZ)

NT1

Network Termination [unit] 1 (ISDN)

NTAS
: NT Advanced Server MS, Windows NT

NTC
: National Telecommunications Conference conference, USA

NTE
: Network Terminal Equipment

NTFS
: [windows] New Technology File System

NTIA
: National Telecommunications and Information Administration / Agency org., USA

NTK
: Need-To-Know slang, Usenet, IRC

NTK
: Newton ToolKit (Apple)

NTN
: National Telecommunications Network

NTO
: Network Terminal Option

NTP
: Network Time Protocol (Internet, RFC 1119)

NTS
: Network Technical Support

NTS
: Network Test System

NTS2
: Network Transport Services /2 (IBM), "NTS/2"

NTSA
: Netware Telephony Services Architecture

NTSC
: National Television Standards Committee org., USA

NTSC
: Never The Same Color slang

NTT
: Nippon Telephone & Telegraph org., Japan

NUA
: Network User Address

NUI

Network User Identification (Datex-P)

NUISSH

Natur- und UmweltInformationsSystem Schleswig-Holstein (UIS), "NUIS-SH"

NUMA

Non Uniform Memory Access (SMP)

NURB

Non Uniform Rational B-spline

NURBS

Non-Uniform Rational B-Spline (CAD, Animation)

NUS

National University of Singapore org.

NUTEK

[National Board for Industrial and Technical Development] org., Schweden

NVDM2

NetView Distribution Manager /2 (OS/2, IBM), "NVDM/2"

NVN

National Videotex Network USA, network

NVP

Network Voice Protocol

NVRAM

Non-Volatile Random Access Memory (RAM, IC)

NVS

NachrichtenVermittlungsSystem (INPOL)

NVT

Network Virtual Terminal (Telnet, Internet)

NWCS

Netware Workstation Compatible Service (Netware, Windows NT)

NWIP

NetWare Internet Protocol (Novell, Netware)

NWNET

NorthWestern states NETwork network, USA, "NWNet"

NYSERNET

New York State Education and Research NETwork network, USA, "NYSERNet"

NZCS

New Zealand Computer Society org., Neuseeland

NZT

New Zealand Time [+1130] (TZ)

TOC

NZUSUGI

New Zealand Unix System User Group, Inc. org., Unix, User group

Go to the previous, next section.

Go to the previous, next section.

TOC

-- O ---

OA
: Object Adapter (ORB, IDL)

OA
: Office Automation

OA
: Open Access (SPI, DB)

OAA
: Open Arcade Architecture (Intel)

OACIS
: Oregon Advanced Computing InStitute org.

OAD
: Open Architecture Driver (Iomega)

OADG
: Open Architecture Development Group org.

OAG
: Online Airline Guide (CIS)

OAG
: Open Application Group org., USA

OAI
: Open Applications Interface

OAK
: Object Application Kernel Java, predecessor, Sun

OAM
: Operations, Administration and Management [cell] (ATM)

OARNET
: Ohio Academic Resources NETwork network, USA, "OARnet"

OAS
: Office Automation System

OASF
: Office Automation System Facilities (OA)

OASIS
: Online Application System Interactive Software

OASIS
: Open And Secure Information Systems (Eureka)

OASYS
: Office Automation SYStem

OATS
: Office Automation Technology and Services (OA)

OAW
: Optically Assisted Winchester (HDD, Seagate)

OBAK
: OBjektAbbildungskatalog (ATKIS)

OBD
: Online Bugs Database (Sun, DB)

OBEX
: OBject EXchange (WordPerfect)

OBS
: Online Book Store (Internet, WWW)

OBST
: OBject management system of STONE (STONE)

OC
: Object Class (OOP)

OC1
: Optical Carrier level 1 [51,48 Mbps] (SONET), "OC-1"

OC12
: Optical Carrier level 12 [622,08 Mbps] (SONET), "OC-12"

OC24
: Optical Carrier level 24 [1244,16 Mbps] (SONET), "OC-24"

OC3
: Optical Carrier level 3 [155,52 Mbps] (SONET), "OC-3"

OC48
: Optical Carrier level 48 [2488,32 Mbps] (SONET), "OC-48"

OCC
: Operations Control Center

OCC
: Other Common Carrier

OCCA
: Open Cooperative Computing Architecture

OCD

Out-of-Cell Delineation (UNI)

OCE

Open Collaborative Environment (Apple)

OCF

Open Computing Facility

OCL

Operation Control Language

OCL

OS/2 inside Class Library (OS/2)

OCLC

Online Computer Library Center (Internet, USA)

OCO

Object Code Only (OOP)

OCR

Optical Character Recognition

OCS

Object Compatibility Standard (Motorola)

OCS

Open Cabling System

OCX

OLE Control eXtentions (MS)

OD

Optical Disk

ODA

Office Document Architecture [protocol] (RFC 1197, ISO 8613, JTC1, ECMA)

ODA

Open Document Architecture (CCITT T.410)

ODAPI

Open Database Application Programming Interface (Borland, DB, API), "Odapi"

ODBC

Open DataBase Connectivity (WOSA, DB, API)

ODBMS

Object orientated Database Management System (DBMS, DB)

ODC

Optical Disc Corporation manufacturer

ODD

Operator Distance-Dialing

ODE

Object Database and Environment (AT&T, DB)

ODE

Online Data Entry

ODETTE

Organization for Data Exchange by Tele-Transmission in Europe Org., Europe

ODF

Opendoc Development Framework (Apple, OpenDoc)

ODF

Opendoc Part Framework (OpenDoc)

ODI

Open Data link Interface (Novell)

ODI

Open Device Interconnect

ODIA

Object ??? (Apple, ALOE)

ODIF

Office Document Interchange Format (ISO 8613, ASN.1, ODA)

ODIN

Optimale Datenmodelle und algorithmen fuer Ingenieur- und Naturwissenschaften [auf hochleistungsrechnern] Uni Karlsruhe, Germany, SNI

ODINSUP

Open Data link Interface - Network driver interface specification SUPport (ODI, NDIS)

ODISS

Optical Digital Image Storage System

ODK

Office Develoment Kit

ODL

Open Document Language (ODA, SGML)

ODM

Object Database Manager (AIX, IBM)

ODMA

Open Document Management API (API)

ODMG

Object Database Management Group org., DB

ODP

Open Distributed Processing (ISO)

ODP
: OverDrive Processor (Intel)

ODPR
: OverDrive Processor Replacement (Intel)

ODR
: Ontrack Data Recovery (Ontrack, Software, Netware)

ODR
: Optimized Dynamic Routing (SNI)

ODS
: Offenes Deutsches Schulnetz network

ODS
: Office Dialog System (OA)

ODS
: Open Data Services

ODS
: Optical Data Systems [inc.] manufacturer

ODS
: Overhead Data Stream

ODSI
: Open Directory Service Interfaces (MS, Windows NT)

ODT
: Online Debugging Technique

ODT
: Open DeskTop (SCO, GUI)

OECOS
: Organizational Engineering for Communications and Organizational Systems (SNI, OA)

OEM
: Original Equipment Manufacturer

OEP
: Operand Execution Pipelines (Motorola, CPU)

OES
: Olivetti EntsorgungsService (Olivetti)

OF
: Overflow Flag assembler

OFA
: Optimal Flexible Architecture (Oracle)

OFC
: Open Financial Connectivity MS, banking

OFFIS
: Oldenburg Forschungsinstitut Fuer Informatikwerkzeuge und -Systeme org., Uni Oldenburg

OFS
: Object File System

OFS
: Output Field Separator (AWK)

OFTA
: OFfice of the Telecommunications Authority org., Hongkong

OFX
: Open Financial eXchange banking, Intuit, MS, Checkfree

OGRAN
: OpenGate - Router Access Node ??? (RND), "OG-RAN"

OH
: Off-Hook (MODEM)

OIA
: Operator Information Area (IBM)

OID
: Object IDentifier (OSI)

OIDL
: Object Interface Definition Language

OII
: Operations-Intelligence Interface mil.

OIL
: Operator Identification Language (ELI)

OIW
: Open Information Warehouse (SAP, R/3)

OIW
: OSI Implementors Workshop (OSI)

OLAP
: OnLine Analytical Processing (DB)

OLE
: Object Linking and Embedding

OLEDS
: Object Linking and Embedding Directory Services (ODSI, MS), "OLE DS"

OLEO

Open Linking and Embedding of Objects (OLE, OpenDoc)

OLI

Originating Line Information

OLIS

Oxford Library Information System

OLIT

OpenLook Interface Toolkit (OpenLook)

OLIVR

OnLine Interactive Virtual Reality, "OLiVR"

OLLI

OnLine Library Information

OLM

OnLine Monitor

OLMC

Output Logic Macro Cell (GAL)

OLPS

OnLine Programming System

OLRT

OnLine Real-Time

OLS

Online Library System

OLTEP

OnLine Test Executive Program

OLTM

Optical Line Terminating Multiplexer, "O-LTM"

OLTP

OnLine Transaction Processing

OLWM

OpenLook Window Manager (OpenLook)

OM1

Open MPEG consortium 1 org., MPEG, "OM-1"

OMA

Object Management Architecture (OMG)

OMC

Operation and Maintenance Center (PLMN, GSM)

OME

Open Messaging Environment (Novell)

OMF
: Object Management Framework (DME)

OMF
: Object Module Format (IBM, MS)

OMF
: Open Media Framework org.

OMFI
: Open Media Framework Interchange (OMF)

OMG
: Object Management Group IBM, HP, DEC, Tandem, Sun, org.

OMI
: Open Messaging Interface

OMISTN
: Optical Mode Interface SuperTwisted Nematic (LCD), "OMI-STN"

OMN
: Open Network Management

OMR
: Optical Mark Reader (Fax)

OMR
: Optical Mark Recognition

OMS
: Object Management System

OMS
: Open Music System (MIDI, Opcode Systems)

OMT
: Object Management Technique (Westmount, CASE)

OMT
: Object Modelling Technique (CASE)

ON
: OEstereichisches Normungsinstitut org., Wien, OEsterreich

ONA
: Open Network Architecture

ONAC
: Operations Network Administration Center

ONAL
: Off Network Access Line

ONC

TOC

Open Network Computing (Sun)

ONCXDR

Open Network Computing eXternal Data Representation, "ONC XDR"

ONE

Open Network Environment (Netscape)

ONI

Operator Number Identification

ONMS

Open-Network Management System

OO

Object Orientated

OOA

Object Orientated [system] Analysis (OOP)

OOAD

Object Orientated Analysis and Design (OOP)

OOCTG

Object Orientated COBOL Task Group CODASYL, org., OOP

OOD

Object-Oriented Design

OODB

Object Oriented Data Base (OOP, DB)

OODBMS

Object Orientated Database Management System (DBMS, DB)

OODL

Object Orientated Dynamic Language (OOP)

OOF

Office of The Future (OA)

OOF

Out of Frame (DS3/E3)

OOL

Object-Oriented Language (OOP)

OOL

Open Objects Library (OS/2)

OOM

Object-Oriented Methodology (OOP)

OOP

Object Orientated Programming

OOPL
: Object-Oriented Programming Language (OOP)

OOPS
: Object Oriented Program Support (OOP)

OOPS
: Object-Oriented Programming System (OOP)

OOPSLA
: [conference on] Object Orientated Programming Systems, Languages and Applications ACM, OOP, conference

OOPSTAD
: Object Orientated Programming for SmallTalk Application Development association org., OOP

OOS
: Object-Oriented Systems

OOS
: Out-Of-Sequence

OOSA
: Object Orientated System Analysis (OOP)

OOSH
: Object-Oriented SHell (OOP, Shell)

OOT
: Object-Oriented Technology (OOP)

OPA
: Open Publishing Architecture

OPAC
: Online Public Access Catalogue (Internet)

OPC
: OpenGL Performance Characterization [project group] OpenGL, org.

OPC
: Overall Performance Category

OPCR
: Original Program Clock Reference

OPDU
: Operation Protocol Data Units

OPEN
: Open Protocol Enhanced Networks

OPENHCI
: Open Host Controller Interface (USB), "OpenHCI"

OPI
: Open Prepress Interface (DTP)

OPM
: Operations-Per-Minute

OPS
: Operations Per Second (CPU)

OPS
: Oracle Parallel Server (Oracle)

OPSEC
: Open Platform for Secure Enterprise Connectivity Org., manufacturer

OPT
: Open Protocol Technology

OQL
: Object Query Language (ODMG)

OR
: Originator / Recipient (X.400, MOTIS), "O/R"

ORAIS
: Opportunities and Risks of Artificial Intelligent Systems conference, GI

ORB
: Object Request Broker (OMA, OMG, CORBA)

ORCA
: Online Resource Control Aid

ORDBMS
: Object Relational Database Management System (DBMS, DB)

ORM
: Optical Remote Module

OROM
: Optical Read Only Memory, "O-ROM"

ORS
: Output Record Separator (AWK)

ORT
: Ongoing Reliability Test

OS
: Operating System

OS2
: Operating System /2 (IBM, OS), "OS/2"

OS360
: Operating System /360 (IBM, OS), "OS/360"

OS400
: Operating System /400 (IBM, OS), "OS/400"

OS9
: Operating System - 9 (Microware, OS), "OS-9"

OSA
: Object System Adaptor (SOM)

OSA
: Office Systems Administrator (OA)

OSA
: Open Scripting Architecture (Apple, OS/2)

OSAC
: Operator Services Assistance Center

OSAK
: Open Systems Application Kernel

OSAM
: Overflow Sequential Access Method

OSAR
: Optical Storage and Retrieval

OSC
: Ohio Supercomputer Center org., USA

OSCAR
: OnScreen Configuration & Activity Reporting (Apex)

OSCRL
: Operating System Command Response Language

OSD
: OnScreen Display

OSD
: Open Software Description standard (MS, Marimba, XML)

OSDS
: Operating System for Distributed Switching

OSE
: Open Software / System Environment

OSEIA
: Open System Environment profile for Imminent Acquisitions (OSE), "OSE/IA"

OSF

Open Software Foundation HP, DEC, IBM, org.

OSF

Operations System Function (IN)

OSF

Oppose Sun Forever [aka Open Software Foundation] Slang, org.

OSF1

Open Software Foundation [system] /1 (OSF, OS, DEC), "OSF/1"

OSI

Open Systems Interconnection (ISO 9646-1)

OSICS

OSI Communication Systems (OSI)

OSID

Origination Signaling IDentifier

OSINLCP

[PPP] OSI Network Layer Control Protocol (OSI, PPP, RFC 1377)

OSIRM

Open Systems Interconnection/Reference Model (ISO), "OSI/RM"

OSKA

ObjektSchluesselKAtalog (ATKIS)

OSM

Operating system Service Module (I2O)

OSM

Optical Storage Manager (ARMS)

OSME

Open Systems Message Exchange

OSN

Office System Node (IBM, DIA, OA)

OSN

Open Systems Network

OSP

Optical Storage Processor

OSP

OrganisationStrukturPlan

OSPF

Open Shortest Path First [protocol / routing] (Internet, IGP, RFC 1583)

OSPM

Operating System directed Power Management

OSQL
: Object-Structured Query Language (DB, SQL, OODBMS)

OSR
: Optical Scanning Recognition

OSS
: Operator Service System

OSTA
: Optical Storage Technology Association org.

OSTC
: Open System Test Consortium org., OSTC

OSTP
: Office of Science and Technology Policy org., USA

OSVS2
: (IBM, MVS), "OS/VS2"

OSWL2
: OS-9 Windows / Level 2, "OSW/L2"

OSWS
: Operating System Work Station

OT
: Object Technology

OT
: Off Topic slang, Usenet

OT
: Open Transport (Apple)

OTA
: Office of Technology Assessment org., IBM, HP, AEG, AT&T, SNI, ...

OTDR
: Optical Time DOMAIN Reflectometer cable

OTOH
: On The Other Hand slang, Usenet, IRC

OTP
: One Time Password

OTP
: One Time Programmable [chip] (IC)

OTPPP
: Open Transport/Point-to-Point Protocol (Apple, OT, PPP), "OT/PPP"

OTPROM

One Time Programmable Read Only Memory

OTS

Office TeleSystem (OA)

OUCL

Oxford University Computing Laboratory

OUG

Occam User Group org., User group

OUI

Organization Unique Identifier (IEEE)

OUI

Organizational Unit Identifier

OUTWATS

OUTward Wide Area Telephone Service

OVID

Object, View, and Interaction Design

OWF

Object World Frankfurt fair

OWG

Optical WaveGuide (LWL)

OWL

Object Windows Library (Borland, API)

OWL

Open Windows Library (API)

OWORM

Optical Write Once Read Many, "O-WORM"

Go to the previous, next section.

Go to the previous, next section.

-- P ---

P2P
: Peer to Peer [net]

PA
: Precision Architecture (HP)

PABX
: Private Automatic Branch eXchange

PAC
: Privilege Attribute Certificate (DCE)

PACE
: Priority Access Control Enabled 3Com, ethernet

PACER
: Public Access to Court Electronic Records

PACS
: Picture Archiving and Communications Systems

PACS
: Public-Access Computing Systems

PACT
: Partnership in Advanced Computing Technologies org. UK

PACT
: Personal Air Communications Technology, "pACT"

PACT
: Production Analyses Control Technique

PACX
: Private Automatic Computer eXchange

PAD
: Packet Assembling Disassembling (CCITT, X.3, X.28, X.29, PSDN)

PAEB
: Pan American EDIFACT Board (UN/EDIFACT)

PAEB
: Pan American EDIFACT Board org., EDIFACT, "PA/EB"

PAI
: Process After Input (R/3)

PAL
: Paradox Application Language (Borland, DB)

PAL
: Phase Alternating Line

PAL
: Privileged Architecture Library (DEC, Alpha)

PAL
: Programmable Array Logic

PAM
: Paging Area Memory

PAM
: Pluggable Authentication Module (Linux, LISA)

PAM
: Primary Access Method (BS2000)

PAM
: Programmable Attribut Maps (DRAM, PCI)

PAM
: Pulse Amplification Modulation

PAML
: Publicly Accessible Mailing Lists (Internet, BBS)

PAN
: Personal Account Number

PANIX
: Public Access internet/uNIX [system] Unix, Internet, network

PAP
: Password Authentication Protocol (RFC 1334)

PAP
: Printer Access Protocol (AppleTalk)

PAP
: ProgrammAblaufPlan (DIN 66001)

PAPI
: PC voice API (API)

PAR
: Positive Acknowledgement with Retransmission [protocols]

PAR
: Project Authorization Request (IEEE)

PARC

Palo Alto Research Center org., USA, Xerox

PARLE

[conference on] Parallel ARchitecture and Languages Europe ECRC, conference

PARTS

Parts Assembly and Reuse Tool Set (Visual Smalltalk)

PAS

Publicly Available Specifications (ISO)

PAS2

Personal Application System /2 (IBM, DB2/2), "PAS/2"

PASC

Portable Application Standards Committee org., IEEE

PAT

Port and Adress Translation (IOS, Cisco, LAN, IP)

PAX

Pixel Adressing eXtensions (Intel, RISC, CPU)

PAX

Portable Archive eXchange (SCO, Unix)

PAX

Private Automatic eXchange

PB

Pipeline Burst [cache]

PBC

Peripheral Bus Computer

PBE

Prompt By Example

PBI

Pacific Bell Internet (ISP)

PBI

Phone Based Interface

PBM

Play By Mail [game]

PBO

Process Before Output (R/3)

PBX

Private Branch eXchange (Nebenstellenanlage)

PC

Personal Computer

TOC

PC
: Printed Circuit (IC)

PC
: Priority Control

PC
: Protocol Control

PC
: Punched Card

PC97
: Personal Computer 97 (MS), "PC 97"

PCA
: Policy Certification Authority (DFN, PGP)

PCA
: Power Calibration Area (CD)

PCAT
: Personal Computer - Advanced Technology, "PC-AT"

PCB
: Printed Circuit Board

PCB
: Process Control Block (BS2000)

PCB
: Protocol Control Block (TCP)

PCC
: Portable C Compiler

PCCA
: Personal Computer Communications Associations org., USA

PCD
: Photo CD (Kodak, CD)

PCD
: Process Control Device

PCDA
: Program Controlled Data Acquisition

PCDOS
: Personal Computer DOS (OS), "PC-DOS"

PCERT
: Purdue Computer Emergency Response Team

PCF

Portable Compiled Font

PCH

Paging CHannel (GSM, CCCH)

PCI

Peripheral Component Interconnect

PCI

Protocol Control Information (OSI, ETSI)

PCIA

Personal Communications Industry Association org., USA, SMS

PCIS

Portable Common Interface Set

PCIXF

Personal Computer Intergrated eXchange Format (OS/2), "PC/IXF"

PCL

Portable Common LOOPS (CLOS)

PCL

Printer Control Language (HP)

PCL

Programmable Command Language (CMU, DEC, TOPS)

PCM

Personal Computer Manufacturer

PCM

Physical Connection Management (FDDI, SMT)

PCM

Plug Compatible Machine

PCM

Pulse Coded Modulation

PCMCIA

People Can't Memorize Computer Industries Acronyms slang

PCMCIA

Personal Computer Memory Card International Association org.

PCMS

Project & Configuration Management System

PCMT

Personal Computer Message Terminal

PCN

Personal Communications Network

PCN
: Public Communications Network (Mobile Systems)

PCNFS
: Personal Computer Network File System

PCO
: Point of Control and Observation (ISO 9646-1, IUT)

PCPC
: Personal Computers Peripheral Corporation

PCR
: Peak Cell Rate (UNI, ATM, SCR)

PCR
: Processor Configuration Register (Motorola, CPU)

PCR
: Program Clock Reference

PCRW
: Phase Change ReWritable (DVD, EMCA), "PC-RW"

PCS
: Personal Communication System (Sony, Panasonic, Motorola)

PCS
: Personal Communications Services

PCS
: Programmable Communications Subsystem

PCS
: Punched Card System (PC)

PCSA
: Personal Computing Systems Architecture [= Pathworks] (DEC, NOS)

PCSD
: Printer Control Sequence Description

PCSI
: Pacific Communication Sciences, Inc. manufacturer

PCT
: Private Communication Technology (Internet, MS, Visa)

PCT
: Probe Control Table (FFST/2)

PCTCP
: Personal Computer/Transmission Control Protocol, "PC/TCP"

PCTE

TOC

Portable Common Tool Environment (CASE, ESPRIT)

PCTS

POSIX Compliance Test Suite (POSIX)

PCV

P-bit Coding Violation [error event] (DS3/E3)

PCV

Path Coding Violation [error event] (DS1/E1)

PCXT

Personal Computer - eXtended Technology, "PC-XT"

PD

Packetization Delay

PD

Public DOMAIN

PDA

Personal Digital Assistent

PDC

Power Disk Cartridge (ECMA)

PDC

Primary DOMAIN Controller MS, Windows NT

PDC

PROLOG Development Center Hersteller, Denmark, PROLOG

PDE

Partial Differential Equation

PDES

Product Data / Definition Exchange Standards (USA, STEP, CIM)

PDF

Portable Document Format (Adobe)

PDF

Program Development Facility (IBM)

PDH

Plesiochronous Digital Hierarchy (ATM)

PDI

Power and Data Interface

PDL

Page / Program Description / Design Language

PDL

PERL Data Language (PERL)

PDL
: Program Development Language

PDM
: Program Development Manager (IBM, ADT)

PDN
: Programmer's Distribution Network (Fido)

PDN
: Public Data Network (Mobile Systems)

PDO
: Portable Distributed Objects (NeXT)

PDOS
: Parallel and Distributed Operating System (OS, MIT)

PDP
: Parallel Distributed Processing (AI)

PDP
: Peripheral Data Processing

PDP
: Plasma Display Panel

PDP
: Programmable Data Processor (DEC)

PDQ
: Parallel Data Query (Informix, DB)

PDS
: Partitioned Data Set

PDS
: Portable Display Shell (Shell)

PDS
: Processor Direct Slot / Socket (Motorola)

PDS
: Professional Development System (MS)

PDS
: Public DOMAIN Software (PD)

PDSP
: Peripheral Data Storage Processor

PDSS
: Post Development and Software Support

PDT

Pacific Daylight Time [-0700] (TZ, PST, USA)

PDTR
: Proposed Draft Technical Report

PDU
: Packet Data Unit

PDU
: Product Development Unit

PDU
: Protocol Data Unit (OSI)

PDV
: Photorealistic Data Visualization

PDV
: ProzessDatenVerarbeitung

PE
: Phase Encoding

PE
: Portable eXecutable (Win32)

PE
: Processing Element (MPP)

PEA
: Pocket Ethernet Adapter LAN, ethernet

PEARL
: Process and Experiment Automation Realtime Language

PEC
: Program Execution Control (IBM, OS/2)

PEDS
: Planning, programming, budget, and execution Electronic Delivery System

PEEK
: Partners Early Experience Kit (Taligent)

PEL
: Picture ELement

PEM
: Privacy Enhanced Mail (PSRG, RFC 1421/1422/1423/1424)

PEN
: Public Education Network

PEP
: Packetized Ensemble Protocol (Telebit)

TOC

PER
: Packed Encoding Rules (ASN.1)

PER
: Program Event Recording

PERL
: Pathologically Eclectic Rubbish Lister slang

PERL
: Practical Extraction and Report Language (PERL)

PERM
: Pre-Embossed Rigid Magnetic (Sony, HDD)

PERMIS
: PERManentes InventurabwicklungsSystem (MBAG)

PEROM
: Programmable Erasable Read Only Memory (IC)

PERT
: Program Evaluation / Evolution and Review Technique

PES
: P-bit Errored Seconds (DS3/E3)

PES
: Photo-Electric Scanning (DTP)

PES
: Programmed Electrical Stimulation

PES
: Proposed Encryption Standard

PEST
: Parameter Estimation by Sequential Testing

PET
: Personal Electronic Translator

PET
: Print Enhancement Technology (Compaq)

PET
: Progressive Educational Technology

PEX
: PHIGS Extension for X (X-Windows, PHIGS)

PFA
: Parameter Field Address (Forth)

PFA

Predictive Failure Analysis (HDD)

PFC

Protocol Field Compression (PPP)

PFE

Page Fault Error (Windows, MS)

PFK

Program Function Key

PFM

Printer Font Metrics (Adobe)

PFPU

Processor Frame Power Unit

PFR

Portable Font Resource (Netscape)

PGA

Pin Grid Array (IC, CPU)

PGA

Professional Graphics Adapter (IBM)

PGC

Professional Graphics Controller

PGFNA

Prince George Free-Net Association org., USA

PGI

Parameter Group Identifier (SPDU)

PGML

Precision Graphics Markup Language (XML, IBM, Netscape, Sun, Adobe)

PGP

Pretty Good Privacy

PGS

Program Generation System

PHIGS

Programmer's Hierarchical Interactive Graphics System

PHIGSPLUS

PHIGS Plus Lumiere and Surfaces (PHIGS), "PHIGS-PLUS"

PHIPS

Professional High-Resolution Image Processing System (CA)

PHUN

Phreakers and Hackers Underground Network network

TOC

PHY
: PHYsical Layer Control (FDDI)

PI
: Placement and Interconnect [system] (VLSI, MIT)

PIA
: Peripheral Interface Adapter

PIA
: Plug-In Administrator

PIC
: Personal Intelligent Communicator

PIC
: Point in Call (IN)

PIC
: Primary Independent Carrier

PIC
: Priority Interrupt Controller (IC)

PIC
: Programmable Interrupt Controller

PICG
: PCTE Interface Control Group org., PCTE

PICS
: Platform for Internet Content Selection org., Internet

PICS
: Plug-in Inventory Control System

PICS
: Protocol Implementation Conformance Statements

PICSDCPR
: PICS Detailed Continuing Property Record, "PICS/DCPR"

PICU
: Priority Interrupt Control Unit

PID
: Process IDentification number (Unix)

PID
: Protocol IDentifier [governing connection types]

PIE
: Personal Interactive Electronics [division] (Apple)

PIG

TOC

Psion Interest Group (Psion)

PIIX

PCI IDE/ISA Xcelerator (Intel IC)

PIM

Parallel Inference Machine (FGCS, AI)

PIM

Personal Information Manager

PIM

Protocol Independent Multicast

PIMB

PCTE Interface Management Board org., PCTE

PIMF

Paar [kabel] In MetallFolie (VDE, STP), "PiMF"

PIMS

Project Information Management System

PIN

Personal Identification Number banking

PIN

Processor Independent Netware (Novell, HP, DEC, Apple, Sun)

PINE

Program for Internet News and Email / PINE Is No longer ELM

PINET

Physicians Information NETwork network, USA

PING

Packet InterNet Groper (ICMP, TCP/IP)

PIO

Parallel / Programmed Input/Output

PIOS

Personen, Institutionen, Objekte, Sachen (INPOL)

PIP

Packet Interface Port

PIP

Paper Impact Printing

PIP

Periphal Interchange Program

PIP

Peripheral Interchange Program

PIP
: Personal Information Processor

PIP
: Picture In Picture (Video)

PIP
: Plug-In Protokoll (ZOC)

PIPS
: Parallel Information Processing System (GNU)

PIPS
: Pattern Information Processing Systems

PIRA
: Printing Industries Research Association org.

PISA
: Portable InformationsSystem Architecture (DB, infodas, PISA)

PISABG
: PISA/Business Graphics (PISA), "PISA/BG"

PISADB
: PISA/Data Base (PISA, DB), "PISA/DB"

PISADD
: PISA/Data Dictionary (PISA), "PISA/DD"

PISADK
: PISA/ ??? (PISA), "PISA/DK"

PISAMP
: PISA/Menue Processor (PISA), "PISA/MP"

PISANT
: PISA/ ??? (PISA), "PISA/NT"

PISAPPS
: PISA/ProduktionsPlanung und -Steuerungsystem (PISA), "PISA/PPS"

PISAQL
: PISA/Query Language (PISA), "PISA/QL"

PISARG
: PISA/Report Generator (PISA), "PISA/RG"

PISW
: Process Interrupt Status Word CPU, assembler

PIT
: Programmable Interval Timer

PITA

TOC

Pain In The Anatomy / As slang, Usenet, IRC

PIU

Path Information Unit (IBM, SNA)

PIU

Plug-In Unit

PIXIT

Protocol Implementation eXtra Information for Testing

PKCS

Public Key Cryptography Standards org., USA, Verschluesselung

PKS

Polizeiliche KriminalStatistik (INPOL)

PL

Physical Layer (ISO, OSI)

PL1

Programming Language no. 1 (DEC), "PL/1"

PLA

Programmable Logic Array

PLATO

Programmed Logic for Automated Teaching Operations

PLB

Picture Level Benchmark (GPC)

PLBSI

Picture Level Benchmark Sample Implementation (GPC)

PLC

Programmable Logic Controller (IC)

PLCC

Plastic Leadless Chip Carrier

PLCP

Physical Layer Convergence Procedure / Protocol (UNI, DS-3)

PLD

Programmable Logic Device (IC)

PLDM

Power Line Disturbance Monitor

PLIP

Parallel Line Internet Protocol (IP), "PL/IP"

PLL

Phase Locked Loop

PLM
: Programming Language for Microcomputers

PLMN
: Public Land Mobile Network (Mobile Systems, GSM)

PLOU
: Physical Layer Overhead Unit (UNI), "PL-OU"

PLP
: Packet Layer Protocol (X.25)

PLP
: Party Line Protocol

PLP
: Presentation Level Protocol

PLS
: Personal Library Systems manufacturer

PLS
: PhysicaL Signaling

PLS
: Programmable Logic Sequencer

PLV
: Presentation Level Video (DVI, Intel)

PM
: Performance Management

PM
: Peripheral Module

PM
: Physical Medium

PM
: Presentation Manager (OS/2)

PMA
: Physical Medium Attachment

PMA
: Policy Management Architecture (Netmanage, Internet)

PMA
: Program Memory Area (CD)

PMAC
: Packet Media Access Control (FDDI), "P-MAC"

PMAC

Peripheral Module Access Controller

PMC

Pseudo-Machine Code

PMD

Physical layer, Medium Dependant [sub-layer] (FDDI, UNI)

PMDRAM

Page Mode Dynamic Random Access Memory (RAM, DRAM, IC), "PM-DRAM"

PME

Pattern Matching Engine

PMJI

Pardon Me for Jumping In slang

PMMU

Paged Memory Management Unit

PMOS

Positive-channel Metal Oxide Semiconductor (IC)

PMP

Point to MultiPoint (UNI)

PMP

Preventive Maintenance Package (AIX, IBM)

PMR

Poor Man's Routing

PMR

Problem Management Report (IBM)

PMS

Personal Mailing System

PMS

Processor Memory Switch

PMS

Project Management System

PMS

Public Message System

PMSP

Preliminary Message Security Protocol

PMU

PowerMeter Unit

PMW

Project Manager Workbench

PMX
: Packet MultipleXer (MUX)

PMX
: Presentation Manager for the X window system, "PM/X"

PNG
: Portable Network Graphics [format] (RFC 2083)

PNNI
: [phase 0] Public Network Node Interface [protocol] (ATM)

PNNI
: Private Network to Network Interface (ATM), "P-NNI"

PNP
: Plug 'N Play (PNP), "PnP"

PNPBIOS
: Plug and Play Basic Input Output System (PNP, BIOS), "PnP BIOS"

PNPN
: Positive Negative Positive Negative [devices]

PNS
: PeaceNet Sweden network

PNW
: Personal NetWare (Novell, Netware)

POACH
: PC-On-A-Chip (PC)

POC
: Proof of Concept

POCSAG
: Post Office Code Standard Advisory Group org., SMS

POD
: Plain Old Document [format]

PODA
: Piloting of ODA (ESPRIT, SNI, Bull, TITN, ICL, Olivetti, ODA)

PODA
: Priority Orientated Demand Assignment (MAC, TDM)

POE
: PowerOpen Association Apple, IBM, Motorola, ..., org.

POEM
: Portable Object-orientated Entity Manager (SGML)

POEMS

Platinum Open Enterprise Management System

POEP

Primary Operand Execution Pipeline (Motorola, CPU), "pOEP"

POF

Plastic Optical Fiber (OWG)

POGO

Programmer-Oriented Graphics Operation

POH

Path OverHead (SONET)

POI

Path Overhead Indicator (SONET)

POI

Point Of Information

POI

Point Of Interaction (IN)

POL

Problem / Procedure / Process Oriented Language

POOL

Parallel Object Orientated Language (DOOM, OOP)

POP

Package for Online Programming

POP

Point Of Presence (Internet, ISP)

POP3

Post Office Protocol 3 (Internet, RFC 1939)

POR

Point Of Return (IN)

POR

Power-On Reset (BIOS)

POS

Primary Operating System (OS)

POS

Programmable Object Select

POSI

Promoting conference for OSI Japan, Org., conference

POSIX

Portable Operating System for unIX (OS, IEEE 1003, ISO 9945, PASC, Unix)

POST
: Power-On Self-Test

POSYBL
: PrOgramming SYstem for distriButed appLications

POTS
: Plain Old Telephone System slang, Usenet, IRC

POV
: Persistence of Vision raytracing

POWER
: Power Optimization With Enhanced RISC [chip] (IBM, Apple, Motorola)

PP
: Physical Partion (LVM)

PP
: Physical Plane (IN)

PP
: Pre-Processor

PPA
: Parallel Port Adapter

PPC
: PowerPC (Apple)

PPC
: Program to Program Communication (Apple)

PPDS
: Personal Printer Data Stream

PPDU
: Presentation Protocol Data Unit (PDU, OSI, ISO 8823, OSI/RM)

PPE
: Packet Processing Engine (Hub)

PPGA
: Plastic Pin Grid Array (CPU)

PPI
: Parallel Port Interface

PPI
: Pixels Per Inch

PPI
: Programmable Peripheral Interface

PPID

Parent Process IDentification number (Unix)

PPL

Pcboard Programming Language (BBS)

PPL

Polymorphic Programming Language (Harvard, Xerox, PARC)

PPM

Pages Per Minute

PPM

Pulse Position Modulation

PPMS

Professional Productivity Management System

PPN

Project Programmer Number

PPN

Public Packet Switching

PPOP

Plain Paper Optimized Printing (Canon), "P-POP"

PPP

Point-to-Point Protocol (Internet, PPP, RFC 1171/1661)

PPS

Packets Per Second

PPS

Parallel Processing System

PPS

ProduktionsPlanung und -Steuerung

PPS

Public Packet Switching [network]

PPSN

Public Packet-Switched Network

PPSU

Personal Printer Spooling Utility

PPTP

Point-to.Point Tunneling Protocol (MS, Ascend, IP)

PPU

Peripheral Processing Unit

PQET

Print Quality Enhancement Technology (Lexmark)

PQFP
: Plastic Quad Flat Package (CPU)

PR
: Packet Radio

PR
: Pattern Recognition

PR
: Pentium Rating (AMD)

PRACSA
: Public Remote-Access Computer Standards Association org., USA

PRAM
: Parameter Random Access Memory (RAM, IC, Apple), "P-RAM"

PRCS
: Project Revision Control System (GNU)

PREP
: PowerPC REferenz Plattform (IBM, Apple), "PReP"

PREPNET
: Pennsylvania Research and Economic Partnership NETwork network, USA, "PREPnet"

PRF
: Problem Report Form (IBM)

PRI
: Primary Rate Interface (ISDN)

PRIDE
: PRofitable Information by DEsign (IRM)

PRISM
: Parallel Reduced Instruction Set Multiprocessing

PRISM
: PRogrammed Integrated System Maintenance

PRMD
: PRivate Management DOMAIN (X.400, MHS)

PRML
: Partial Response - Maximum Likelihood (HDD)

PRN
: Pseudo Random Noise [code] (GPS)

PRNG
: Pseudo Random Numer Generator

PRO

Precision RISC Organization org.

PROFIBUS

PROcess FIeld BUS

PROFS

PRofessional OFice System (IBM, VM)

PROLOG

PROgramming in LOGic

PROM

Partial Read Only Memory, "P-ROM"

PROM

Programmable Read Only Memory

PROMATS

PROgrammable Magnetic Tape System

PROPAL

PROgrammed PAL

PROS

??? [protocol]

PROTEL

PRocedure Oriented Type Enforcing Language

PRPQ

Programming Request for Price Quotation

PRS

Pattern-Recognition System (PR)

PRT

Portable Remote Terminal

PRT

Program Reference Table

PRTM

Printing Response Time Monitor

PS

PostScript (Adobe)

PS

Power Supply

PS

Privilege Service (DCE)

PS

Process Status (Unix)

PS2
: Personal System /2, "PS/2"

PSAP
: Presentation Service Access Point (OSI, OSI/RM, SAP)

PSB
: Program Specification Block

PSC
: Pittsburgh Supercomputing Center org., USA

PSCNET
: Pittsburgh Supercomputing Center NETwork network, USA, "PSCnet"

PSD
: Printer Sharing Device

PSD
: Programmer's Supplementary Documents (BSD, Unix)

PSDC
: Public Switched Digital Capability

PSDN
: Packet-Switching Data Network

PSDS
: Public-Switched Digital Service

PSE
: Packet Switch Exchange

PSE
: Port Switched Ethernet Bytex, VLAN, ethernet

PSES
: P-bit Severely Errored Seconds (DS3/E3)

PSF
: Permanent Swap File

PSF
: Print Services Facility (IBM)

PSI
: Performance Systems International, inc.

PSID
: Presentation Space IDentifier (IBM)

PSIPU
: PostScript - Intelligent Processing Unit (Canon, Adobe, CLC), "PS-IPU"

PSIU

Packet Switch Interface Unit

PSK

Phase Shift Keying (DFUE)

PSL

PD Service Lage (Haendler)

PSL

Public Software Library

PSM

Persistent Stored Modules (SQL)

PSM

Personal Software Marketing (IBM)

PSN

Packet / Public Switched Network

PSN

Packet Switch Node (ARPANET, MILNET)

PSOM

Persistency framework in SOMobjects (IBM, DSOM)

PSP

Personal Software Products (IBM)

PSP

Program Segment Prefix (DOS)

PSP

ProjektStrukturPlan

PSPDN

Packet Switched Public Data Network (IN)

PSR

Phase Shift Register

PSR

Program Support Representative

PSRG

Privacy and Security Research Group (IRTF)

PSRO

Professional Standards Review Organization

PSS

Packet Switch Services / Stream

PST

Pacific Standard Time [-0800] (TZ, PDT, USA)

PSTN
: Public Switched Telephone Network (IN)

PSU
: Portable Storage Unit

PSU
: Primary Sampling Unit

PSU
: Program Storage Unit

PSW
: Program Status Word

PT
: Payload Type (ATM)

PTAL
: Payment Transaction Application Layer Internet, banking

PTAT
: Private Trans-Atlantic Telecommunications

PTB
: Physikalisch-Technische Bundesanstalt org.

PTC
: Pacific Telecommunications Council org.

PTC
: Parallel Test Component (ISO 9646-3, TTCN)

PTD
: Parallel Transfer Disk / Drive

PTE
: Packet Transport Equipment

PTE
: Path Terminating Equipment (SONET)

PTF
: Program Temporary Fix (IBM)

PTI
: Payload Type Identifer (ATM, PT)

PTM
: Packet Transfer Mode (ATM)

PTO
: Patent and Trademark Office org., USA

PTOCA

Presentation Text Object Content Architecture (IBM, MO:DCA)

PTP
: Paper Tape Punch

PTR
: Paper Tape Reader

PTT
: Postes, Telegraphe et Telephone org., Schweiz

PTW
: Primary Translation Word

PTY
: Pseudo-Terminal driver

PU
: Physical Unit (NAU)

PU
: Power Unit

PU
: Processing Unit

PUA
: Profiling User Agent

PUB
: Physical Unit Block

PUC
: Peripheral Unit Controller

PUC
: Public Utilities Commission

PUCP
: Physical Unit Control Point (IBM, SNA)

PUFFT
: Purdue University Fast FORTRAN compiler (FORTRAN)

PUID
: Physical Unit IDentifier (IBM)

PUMA
: Power User's Macintosh Association org., Apple

PUMA
: Processor Upgradable Modular Architecture

PUMA
: Programmable Universal Micro Accelerator

PUMS
: Physical Unit Management Services

PUP
: PARC Universal packet Protocol

PUT
: Program Update Tape

PV
: Physical Volume (LVM, HDD)

PVC
: Permanent Virtual Circuit (SVC)

PVC
: Permanent Virtual Circuit / Channel / Connection (ATM)

PVCC
: Permanent Virtual Channel Connection (ATM)

PVCS
: Polytron Version Control System

PVD
: Primary Volume Descriptor (CD, IS 9660)

PVITL
: Program-Variation-In-The-Large (SCM)

PVITS
: Program-Variation-In-The-Small (SCM)

PVM
: Parallel Virtual Machine

PVN
: Private Virtual Network

PVP
: Packet Video Protocol

PVPC
: Permanent Virtual Path Connection (ATM)

PWB
: Printer Wire Board

PWB
: Programmers Work Bench (MS)

PWD
: Print Working Directory (Unix)

PWM

Pulse Width Modulation

PWN

Peacenet World News

PWSCS

Programmable WorkStation Communication Services

PWV

Pulse-Wave Velocity

PY

Person Years

PZDF

Phil Zimmermann Defense Fund (PGP)

PZE

???

Go to the previous, next section.

Go to the previous, next section.

-- Q ---

QA
: Quality Assurance

QA
: Question & Answers, "Q&A"

QAM
: Quadrature Amplitude Modulation

QAP
: Quality-Assurance Program (QA)

QAPI
: Quality-Assurance Program Inspections (QA)

QAPI
: Queue Application Program Interface (SAP, API)

QAPP
: Quality-Assurance Program Plan (QA)

QAS
: Quasi-Associated Signaling

QBASIC
: Quick Beginner's All-purpose Symbolic Instruction Code (BASIC, DOS)

QBBS
: Quick Bulletin Board System (BBS)

QBE
: Query By Example (DB)

QBF
: Query By Forms (DB)

QBIC
: Query By Image Content (IBM)

QCB
: Queue Control Block

QCDP
: Quality Color Dithering Modus

QCIF
: Quarter Common Interchange Format

TOC

QD
: Queueing Delay

QDL
: Quadri Data Layer [CD] (CD)

QDOS
: Quick and Dirty Operating System (OS)

QEMM
: Quarterdeck Expanded Memory Manager (EMM, Quarterdeck)

QF
: Query Filter (DB)

QFA
: Quick File Access

QFP
: Quad Flat Pack

QIC
: Quarter Inch Committee org.

QICSC
: Quarter Inch Cartridge Standards Committee (QIC)

QICW
: Quarter Inch Cartridge Wide (QIC)

QIS
: Quality Information System

QL
: Query Language

QLD
: Queuing Literature Database (DB)

QLI
: Query Language Interpreter

QLLC
: Qualified Logical Link Control [protocol] (IBM)

QMC
: Quine McCluskey approach

QMF
: Query Management Facility (IBM)

QMS
: Queue Management Service (Netware)

QOS

Quality Of Service (ATM, CLR, CTD, CDV), "QoS"

QPA

Quality Program Analysis

QPS

Quark Publishing System (DTP)

QPSK

Quadri Phase Shift Keying [modulation]

QPSX

Queue Packet and Synchronous circuit Exchange

QPU

Quick Pascal Units (MS)

QSAM

Queued Sequential Access Method

QTAM

Queued Terminal Access Method

QUEX

Query Update by EXample

QUICC

QUad Integrated Communications Controller (Motorola)

QUIP

QUad In-line Package

QUIPS

QUality Improvements Per Second

QXI

Queue eXecutive Interface

Go to the previous, next section.

Go to the previous, next section.

TOC

-- R ---

RA
: Remote Access (BBS)

RACE
: Research and technology development in Advanced Communications technologies in Europe Europe, Vorlaeufer, CORDIS

RACF
: Resource Access Control Facility (IBM)

RACH
: Random Access CHannel (GSM, CCCH)

RAD
: Rapid Access Disk

RAD
: Rapid Application Development [toolkit] (Delphi, Borland)

RAD
: Research And Development [data communications] manufacturer

RADIUS
: Remote Authentication Dial-In User Service (RFC 2138)

RADSL
: Rate Adaptive Digital Subscriber Line (ADSL)

RAI
: Remote Alarm Indication (DS3/E3)

RAID
: Redundant Arrays of Independent Disks

RAIR
: Random Access Information Retrieval

RAL
: ReichsAusschuss fuer Lieferbedingungen (IEC 12119)

RAL
: Rutherford Appleton Laboratory org., UK

RAM
: Random Access Memory (RAM, IC)

RAMDAC

Random Access Memory Digital to Analog Converter (RAM)

RAMIS

Rapid Access Management Information System

RAMP

Remote Access Maintenance Protocol

RAMS

Record Archival Management System

RAND

Rural Area Network Design

RAP

[internet] Route Access Protocol (RFC 1476, Internet)

RAP

Remote Access Point

RAPID

Resource And Performance Interactive Display system

RAPS

Real Application on Parallel Systems

RARE

Reseaux Associes pour la Recherche Europeene org.

RARP

Reverse Address Resolution Protocol (Internet, RFC 903)

RAS

Reliability, Availability and Serviceability (IBM)

RAS

Remote Access Software

RAS

Row Address Strobe (IC, DRAM)

RATP

Reliable Asynchronous Transfer Protocol (RFC 916)

RATS

Radio-Amateur Telecommunications Society org., USA

RAVE

Relational Advanced Visual Environment

RBA

Relative Byte Address

RBBS

Remote Bulletin Board System (BBS)

TOC

RBF
: Random Block Filemanager (OS-9)

RBH
: Remote Bridge Hub

RBMS
: Remote Bridge Management Software

RBOC
: Regional Bell Operating Company (USA)

RC
: Region Co-ordinator (FidoNet)

RC
: Release Candidate (MS)

RC
: Return Code

RC
: Routing Control

RC24
: Ron's Code 2/4 cryptography, "RC2/4"

RC5
: Rivest Cipher 5 ??? RFC 2040, cryptography

RCAS
: Reserve Component Automation System

RCC
: Radio Common Carrier

RCC
: Regional Control Center

RCC
: Remote Cluster Controller

RCC
: Routing Control Center

RCD
: Receiver-Carrier Detector

RCE
: Resident Command Extension (DOS)

RCF
: Reader's Comment Form (IBM)

RCF

Remote Call-Forwarding

RCGI
: Remote Common Gateway Interface (CGI, WWW, Novell)

RCL
: Runtime Control Library

RCM
: Remote Carrier Module

RCP
: Remote Copy Program

RCS
: Reloadable Control Storage

RCS
: Remote Connection Service (IBM, OS/2, LAN)

RCS
: Resource Construction Set

RCS
: Revision Control System (Unix, PD, CM, GNU)

RCTC
: Rewritable Consumer Time Code (Video)

RCTL
: Resistor-Capacitor-Transistor Logic

RD
: Receive Data (MODEM)

RD
: Recursive Design (CASE)

RD
: Remove Directory (DOS, OS/2)

RD
: Research and Development, "R&D"

RD
: Route Descriptor

RDA
: Remote Database Access (ISO, OSI)

RDB
: Receive Data Buffer

RDB
: Relational DataBase (DB)

RDB
: Rigid Disk Block (Amiga, Commodore)

RDBMS
: Relational DataBase Management System (DBMS, DB)

RDBMSMIB
: Relational DataBase Management System - Management Information Base (DB), "RDBMS-MIB"

RDC
: Remote Data Connector

RDD
: Random Digital Dial

RDD
: Replacable Database Driver (Clipper, CA-VO, DB)

RDE
: Received Data Enable

RDES
: Remote Data Entry System

RDF
: Rate Decrease Factor

RDF
: Record Definition Field

RDF
: Resource Description Framework (IBM, Netscape, MS, ..., WWW)

RDI
: Remote Defect Identification / Indicator (UNI)

RDL
: Relational Database Language (DB)

RDL
: Remote Digital Loopback

RDLAP
: Radio Data Link Access Protocol (MODACOM), "RD-LAP"

RDM
: Relational Data Modeler

RDM
: Reliably Delivered Message

RDN
: Relative Distinguished Name (X.500)

RDNCRC

Research Data Network Cooperative Research Centre Org., Australia

RDO

Remote Data Objects (DB)

RDP

Reliable Datagram Protocol

RDR

Request Data with Reply (Feldbus)

RDRAM

Rambus Dynamic Random Access Memory (RAM, IC)

RDRN

Rapidly Deployable Radio Networks (USA, Uni Kansas)

RDS

Radio Digital System

RDS

Rapid Development System (DB, Informix)

RDS

Remote Data Services

RDSN

Region Digital Switched Network

RDT

Radio Digital Terminal

RDT

Restricted Data Transmissions

RE

Research and Engineering, "R&E"

REACH

Research and Educational Applications of Computers in the Humanities

READ

Relative Element Address Designate cryptography

REGEX

REGular EXpressions (GREP, EMACS, ...)

REGIS

REmote Graphics Instruction Set

REM

Remote Equipment Module

REM

Ring Error Monitor

REMIS
: Real Estate Management Information System

REMOS
: Resources Management On-Line System

RES
: Remote Entry Services

RESTENA
: RESeau Teleinformatique de l'Education NAtionale et de la recherce network, Luxemburg

RET
: Report Engine Technology (CA, DB)

RET
: Resolution Enhancement Technology (HP), "REt"

RETM
: Rare Earth / Transition Metal (MO), "RE/TM"

REXX
: Restructured EXtended eXecutor [language] (IBM, SAA)

RF
: Radio Frequency (Mobile Systems)

RFA
: Request for Assistance (Internet)

RFC
: Remote Function Call (SAP, CPIC)

RFC
: Request For Comments (Internet)

RFD
: Ready-For-Data

RFD
: Report Fragmentation Done [bit] (CATNIP)

RFD
: Request For Discussion (Internet, Usenet), "RfD"

RFE
: Request For Enhancement (Internet)

RFI
: Radio Frequency Interference

RFI
: Request For Information (Internet)

RFMS

Remote File Management System

RFNM

Request For Next Message (IMP)

RFP

Request For Proposal (Internet)

RFS

Remote File Sharing (AT&T, Unix)

RFS

Remote File System

RFT

Request For Technology (OSF)

RFT

Revisable Form Text

RFTS

Remote File Transfer System (DDVS)

RGB

Rot Gruen Blau / Red Green Blue color system, DTP

RGP

Raster Graphics Processor

RGP

Remote Graphics Processor

RH

Request Header

RI

Ring Indicator (RS-232, MODEM)

RI

Routing Information

RIACS

Research Institute for Advanced Computer Science org.

RIDE

Research Issues in Data Engineering (IEEE-CS)

RIF

Routing Information Field (Token Ring)

RIFF

Resource Interchange File Format (MS, IBM, MM)

RIFF

Resource Interchange File Format (MS, MM)

RIG
: Related Interest Group

RII
: Routing Information Indicator

RIM
: Remote Installation and Maintenance

RIME
: Relaynet International Message Exchange

RIO
: Redistributed Internet Objects (S3, VRML)

RIP
: Raster Image Processor (DTP)

RIP
: Remote Imaging Protocol (BBS)

RIP
: Routing Information Protocol (BSD, IGP, RFC 1721, IP)

RIPE
: Reseaux IP Europeene Europenet

RIPL
: Remote Initial Program Load (IBM)

RIPNG
: Routing Information Protocol Next Generation (RIP, IPV6, RFC 2080), "RIPng"

RIPS
: Raster Image Processing Systems corporation manufacturer

RIS
: Remote Installation Service

RISC
: Reduced Instruction Set Code (CPU)

RISC
: Research Institute for Symbolic Computation org., OEsterreich

RISCOS
: RISC Operating System (MIPS, Acorn, OS), "RISC OS"

RISLU
: Remote Integrated Services Line Unit

RITECH
: Resolution Improvement TECHnology (Epson)

RITSEC

Regional Information Technology and Software Engineering Center

RJE

Remote Job Entry (IBM, Internet, RFC 407)

RJEF

Remote Job Entry Function

RJP

Remote Job Processing

RKRM

ROM Kernel Reference Manual (Amiga, Commodore)

RLE

Run-Length Encoded

RLIN

Research Libraries Information Network network

RLL

Run Length Limited [encoding]

RLN

Remote LAN Nodes (LAN)

RLP

Radio Link Protocol (GSM)

RLP

Resource Location Protocol (Internet, RFC 887)

RLS

Received Line Signal

RLSD

RLSD

Received Line Signal Detector

RLT

Remote Line Test

RLT

Remote Line Test

RLV

RingLeitungsVerteiler

RM

Resource Management

RMAS

Remote Memory Administration System

RMATS

Remote Maintenance Administration and Traffic

RMF

Resource Measurement Facility (IBM, MVS)

RMI

Remote Method Invocation (Java, API)

RMM

Ring Management Module (Token Ring)

RMON

Remote MONitoring

RMOS

Realtime Multitasking Opearting System (SNI, OS)

RMS

??? [bus]

RMS

Reason Maintenance System (AI)

RMS

Richard Matthew Stallman (FSF, EMACS)

RMT

Raw Magnetic Tape (Unix)

RMT

Rexxware Migration Tool (Simware, REXX)

RMT

Ring ManagemenT (FDDI, SMT)

RNA

Remote Network Access

RND

RAD Network Devices RAD, manufacturer

RNOC

Regional Network Operations Center

RNR

Receive Not Ready (LAPB)

RNUI

Remote Network User Identification (Datex-P)

RO

Read-Only (I/O)

ROAMS

Reusable Object Access and Management System

ROD
: Rewritable Optical Disk (OD)

ROLAP
: Relational OnLine Analytical Processing (OLAP, DB)

ROLC
: Routing Over Large Clouds

ROM
: Read Only Memory

ROMBIOS
: Read-Only Memory - Basic Input Output System (ROM, BIOS), "ROM-BIOS"

ROMC
: Required Operational Messaging Characteristics

ROP
: Remote Operations Service (IBM, OS/2)

ROS
: Read-Only Storage

ROS
: Resident Operating System (OS)

ROSE
: Remote Operations Service Element (OSI, RPC)

ROTFL
: Rolling On The Floor Laughing slang, Usenet, IRC

ROTFLBTC
: Rolling On The Floor Laughing and Biting The Carpet slang, Usenet, IRC

RP
: Relay Party (IRC)

RPC
: Remote Procedure Call (Sun, Xerox, OSF, ECMA, RFC 1831)

RPCL
: Remote Procedure Call Language (ONC, Sun)

RPE
: Remote Peripheral Equipment

RPELTP
: Regular Pulse Excitation with Long-Term Prediction [loop] (LPC), "RPE-LTP"

RPG
: Report Program Generator

RPL

Remote Program Load

RPL

Requested Privilege Level

RPL

Resident Programming Language

RPM

Radio Packet Modem (Motorola)

RPM

Redhat Package Manager (Linux)

RPM

Remote Port Module (Ascend)

RPN

Reverse Polish Notation

RPP

Relative Processor Performance (CPU, Cray)

RPS

Realtime Programming System

RQBE

Relational Query By Example (DB, QBE)

RR

Radio Resource management (MM, CM, GSM)

RRCM

Reservation Request Control Mechanism (DQDB)

RRDS

Relative Record Data Set (VSAM)

RRIP

Rock Ridge Interchange Protocol (CD, Unix)

RRZE

Regionales RechenZentrum Erlangen org.

RRZN

Regionales RechenZentrum fuer Niedersachsen org.

RS

Recommended Standard

RS

Registery Service (DCE)

RSA

Random Scheduling Algorithm [protocol]

TOC

RSA
: Reference System Architecture

RSA
: Reusable Software Assets

RSA
: Rivest, Shamir und Adleman cryptography, RSA

RSAC
: Recreational Software Advisory Council org.

RSAP
: Remote Service Access Point (SAP)

RSAT
: Reliability and System Architecture Testing

RSCS
: Remote Source Control System

RSCS
: Remote Spooling Communications Subsystem (IBM, VM, NJE)

RSE
: Removable Storage Elements

RSE
: Research and Systems Engineering

RSF
: Remote Support Facility

RSH
: Remote SHell (Unix, BSD, Shell)

RSH
: Restricted SHell (Unix, Shell)

RSI
: Repetitive Strain Injury

RSIS
: Relocateable Screen Interface Specification

RSLM
: Remote Subscriber Line Module

RSM
: Remote Switching Module

RSN
: Real Soon Now slang

RSPC

Reed Solomon Product Code (SDD), "RS-PC"

RSTS

Resource Sharing Time Sharing (DEC)

RSTSE

Resource System Time Sharing/Enhanced (DEC), "RSTS/E"

RSU

Remote Switching Unit

RSVP

Resource reSerVation Protocol (IP)

RSX11D

Resource Sharing eXecutive - 11D (DEC), "RSX-11D"

RSX11M

Resource Sharing eXecutive - 11M (DEC), "RSX-11M"

RSX3D

Realistic Sound eXperience - 3D (Intel, Audio, VRML), "RSX-3D"

RT

Register Transfer

RT

Remote Terminal

RT

Research and Technology, "R&T"

RT

Routing Type

RTA

Real-Time Accelerator

RTAM

Remote Teleprocessing Access Method

RTB

Read Tape Binary

RTBM

Real Time Bit Mapping

RTC

Real Time Clock

RTC

Real-Time Command

RTCS

Real Time Computer System

RTCU
: Real Time Control Unit

RTD
: Real Time Display

RTDHS
: Real Time Data Handling System

RTE
: Real Time Execution

RTE
: Remote Terminal Emulation

RTE
: Run Time Environment

RTF
: Rich Text Format

RTFF
: Read The Fucking FAQ slang, Usenet

RTFM
: Read The Fucking Manual slang, Usenet, IRC

RTG
: Routing Table Generator

RTI
: ReTurn from Interrupt

RTL
: Real Time Language

RTL
: Register Transfer Language (GCC)

RTL
: Resistor-Transistor Logic

RTL
: RunTime Library

RTM
: Registered Transfer Module

RTM
: Release To Market

RTM
: Remote Test Module

RTMP

Routing Table Maintenance Protocol (AppleTalk)

RTOSUH

Real Time Operating System - Universitaet Hannover (OS), "RTOS-UH"

RTP

Real Time Protocol (Internet, RFC 1889/1890)

RTS

Real Time System

RTS

Reliable Transfer Service (OSI)

RTS

Request To Send (MODEM, RS-232)

RTS

Residual Time Stamp

RTSE

Reliable Transfer Service Element (OSI)

RTSM

RealTime System Manager

RTSP

Real Time Streaming Protocol (TV, WWW)

RTT

Round-Trip Time

RTTI

Run-Time Type Identification (ANSI)

RTTY

Radio Tele TYpe

RTU

Real Time Unix (Unix)

RTV

Real Time Video

RTVBR

RealTime Variable Bit Rate (VBR, ATM), "rt-VBR"

RUA

Remote User Agent

RUAC

Remote User Access Centers

RUI

Reality User Interface

RUOW
: Remote Unit Of Work (DRDA, IBM), "RUoW"

RUS
: Rechenzentrum der Universitaet Stuttgart org., Uni Stuttgart

RVD
: Remote Virtual Disk

RW
: Read/Write (I/O), "R/W"

RWCP
: Real World Computing Partnership org., Japan

RZ
: RechenZentrum

RZ
: Return-to-Zero [recording]

RZG
: RechenZentrum Garching

Go to the previous, next section.

Go to the previous, next section.

-- S ---

SA
: Source [MAC] Address (SNA, Token Ring, ATM, FDDI, ...)

SA
: Storage Array

SA
: Structered Analysis / Strukturierte Analyse (CASE)

SA
: System Administrator

SA
: Systems Analyst

SAA
: Standard Application Architecture (IBM)

SAA
: Standards Association of Australia Org., Australia

SAAL
: Signalling ATM Adaptation Layer (ATM)

SABB
: Storage Array Building Block

SABM
: Set Asynchronous Balanced Mode (LABM, LAPB, HDLC)

SABME
: Set Asynchronous Balanced Mode Extension (SABM)

SAC
: Service / Special Area Code

SAC
: Single Attachment Concentrator (FDDI)

SAC
: Strict Avalanche Criterion cryptography

SACCH
: Slow Associated Control CHannel (GSM, DCCH)

SACT
: SPARC Application Conformance Test (SI, SPARC)

SAD
: Serial Analog Delay

SADF
: Semi-Automatic Document Feeder

SADT
: Structured Analysis and Design Techniques (SA)

SAF
: Service Access Facilities (Unix)

SAFTE
: SCSI Accessed Fault-Tolerant Enclose (SCSI, RAID, Intel, NStor), "SAF-TE"

SAG
: SQL Access Group Org., manufacturer, DB

SAGE
: Software Aided Group Environment (GSS, NUS)

SAHF
: Store AH Into Flags assembler

SAIL
: Stanford Artificial Intelligence Laboratory [language] (USA)

SAL
: Security Access List

SAL
: Semware Applications Language (Semware)

SAL
: Symbolic Assembly Language assembler

SALT
: Script Application Language for Telix

SALU
: Structured Assembly Language Utilities

SAM
: SCSI-3 Architecture Model

SAM
: Sequential Access Mode / Method

SAM
: Sort and Merge

SAM
: Symantec Anti-Virus For Macintosh (Apple)

SAM

TOC

System Activity Monitor

SANE

Standard Apple Numeric Environment (Apple)

SAO

Source Address Omitted [flag] (CATNIP)

SAP

Service Access Point (OSI)

SAP

Service Advertising Protocol (Novell, Netware, IPX)

SAP

Symbolic Assembler Programm (IBM, IBM 704)

SAP

SystemAnalyse und Programmentwicklung manufacturer, Vorlaeufer

SAP

Systems, Applications and Products [in data processing] [ag] manufacturer

SAR

Segmentation And Reassembly

SAR

Store Address Register

SARAH

Standard Automated Remote-to-AUTODIN Host AUTODIN, mil.

SARPDU

Segmentation And Reassembly Protocol Data Unit (ATM, PDU), "SAR PDU"

SART

Structered Analysis / Real Time (SA, CASE), "SA/RT"

SARTS

Switched Access Remote Test System

SAS

Session Active Screen (IBM)

SAS

Simulation Automation System

SAS

Single Attachment Station (FDDI)

SAS

Statistical Analysis System

SASE

Specific Application Service Element (ISO, OSI, CASE)

SASI
: Shugart Associates System Interface

SAST
: South Australia Standard Time [+0930] (TZ)

SAT
: Standard AUTODIN Terminal AUTODIN, mil.

SATAN
: Security Administrator Tool for Analyzing Networks (Internet)

SATCOM
: SATellite COMmunications

SATF
: Shared Access Transport Facility

SAU
: Secure Access Unit

SBA
: SideBand Address [port / bus] (AGP)

SBA
: Standards-Based Architectures

SBA
: Synchronous Bandwitdth Allocation (SMT, FDDI)

SBA
: System For Business Automation

SBC
: SCSI Block Commands (SAM)

SBC
: Single Board Computer

SBC
: Small Business Computer

SBCCS
: Single Byte Command Code Set [protocol]

SBCS
: Single Byte Character Set (ASCII, DBCS)

SBF
: Sequential Block Filemanager (OS-9)

SBH
: Secure Backbone Hub (Accton)

SBI

Storage Bus Interconnect

SBI

Synchronous Bus Interface

SBIS

Sustaining Base Information System

SBL

Super BASIC Language (BASIC)

SBLC

Sustaining Base Level Computer

SBMS

Southwestern Bell Mobile Service

SBP

SCSI-3 serial Bus Protocol (SAM)

SBS

Small Business Server

SBUS

Sun [i/o interface] BUS (Sun, SPARC), "SBus"

SC

SubCommittee (ISO, TC, IEC, ...)

SCA

Software Corporation of America manufacturer, USA

SCA

Synchronous Clock Adjustment

SCAF

Service Control Agent Function (IN)

SCAI

Scandinavian Conference on Artificial Intelligence conference, FAIS, AI

SCAM

SCSI Configured AutoMatically (SCSI)

SCAMC

Symposium on Computer Applications in Medical Care conference

SCAN

Switched-Circuit Automatic Network

SCB

Subsystem Control Block (OS/2, IBM)

SCC

SCSI Controller Commands (SAM)

SCC
: Serial Communication Controller

SCC
: Serial Controller-Chip (IC)

SCC
: Softarc Certified Consulter (SoftArc)

SCC
: Specialized Common Carrier

SCC
: Standards Council of Canada org., Canada

SCC
: Storage Connecting Circuit

SCCP
: Signaling Connection Control Part (MSC, GSM)

SCCS
: Source Code Control System (Unix, AT&T, CM)

SCCS
: Specialized Common Carrier Service

SCCS
: Switching Control Center System

SCD
: SPARC Compliance Definition (SI, SPARC)

SCDE
: Significant CALS Data Elements

SCDMS
: Society for Clinical Data Management Systems org., USA

SCE
: Structure Chart Editor

SCEF
: Service Creation Environment Function (IN)

SCEIO
: Societe Canadienne pour l'Etude de l'Intelligence par Ordinateur Org., Canada, AI

SCF
: Selective Call-Forwarding

SCF
: Sequential Character Filemanager (OS-9)

SCF

Service Control Function (IN)

SCG

Security Classification Guide

SCH

Synchronization CHannel (GSM)

SCI

Scalable Coherent Interface (ANSI)

SCIT

Semi-Conductor and Interconnect Technologies

SCL

System Control Language

SCM

Segment Control Module

SCM

Service Control Manager Windows NT

SCM

Small Core Memory

SCM

Software Configuration Management

SCMP

Stream Control Message Protocol (ST2)

SCN

Specifications Change Notice

SCNR

Sorry, Could Not Resist slang, Usenet

SCO

Santa Cruz Operation manufacturer, Unix

SCOOPS

SCheme Object Oriented Programming System (OOP)

SCOPE

SCalable Object Processing Environment (Creamware)

SCOPE

Simple COmmunications Programming Environment (DFUE)

SCP

Secondary Communications Processors

SCP

Service Control Point (OSI)

SCP
: Standard Configuration Profile (MODEM)

SCP
: System Control Program

SCPC
: Single Channel Per Carrier

SCR
: Selective Call Rejection

SCR
: Sustainable Cell Rate (UNI, ATM, VBR)

SCRI
: Supercomputer Computations Research Institute org., USA, HPC

SCS
: [systimax] Structured Cabling System (AT&T)

SCS
: Silicon Controlled Switch

SCS
: Singapore Computer Society org., Singapur

SCS
: Small Computer System

SCS
: SNA Character String (IBM)

SCS
: Switch Control Software (ForeRunner, ATM)

SCSA
: Signal Computing System Architecture

SCSA
: Sun Common SCSI Architecture (Sun)

SCSI
: Small Computer Systems Interface (SCSI)

SCT
: Subroutine Call Table

SCTS
: Secondary Clear to Send

SCU
: Selector Control Unit

SCU

System Control Unit (CPU, POWER)

SCX

Specialized Communications eXchange

SD

Starting Delimiter (FDDI, Token Ring)

SD

Structured Design (CASE)

SD

Super Density [disk] (CD, Toshiba, Time Warner)

SDA

Screen Design Aid (IBM, ADT)

SDA

Shared Data Architecture

SDA

Software Design Automation

SDA

Source Data Automation

SDA

Swappable Data Area (DOS)

SDA

System Display Architecture

SDAV

Systems Design Automation Vendor

SDC

Sample Data Collection

SDC

Software Development Control [system] (CMU, CM)

SDC

Software Distribution Center

SDCCH

Stand-alone Dedicated Control CHannel (GSM, DCCH)

SDCD

Secondary Data Carrier Detect

SDD

Software Design Document

SDD

Super Density Disk (Toshiba, Time Warner, ...)

SDDAS
: Southwest research Data Display and Analysis System

SDDI
: Shielded Distributed Data Interface (FDDI, STP)

SDE
: Software Development Environment

SDF
: Screen Definition Facility

SDF
: Standard Data Format

SDF
: System Dialog Facility (BS2000)

SDH
: Synchronous Digital Hierarchy (FDDI, ATM)

SDI
: Single Document Interface

SDI
: Source Data Information

SDI
: Standard Data Interface

SDI
: Standard Disk Interconnect

SDI
: Standard Disk Interface

SDI
: Standard Drive Interface

SDI
: Storage Device Interconnect

SDI
: Storage Device Interface (Novell, Netware, SMS)

SDI
: Super Data Interchange

SDIF
: SGML Document Interchange Format (SGML, ISO, IS 9069)

SDIF
: System Independent Data Format (Novell, SMS)

SDILINE

Selective Dissemination of Information onLINE

SDIMM

Single [RAS] Dual Inline Memory Module (DIMM, RAS), "S-DIMM"

SDIS

Switched Digital Integrated Service

SDK

Software Development Kit (MS)

SDL

Specification and Description Language (CCITT)

SDL

System Design Language

SDLC

Synchronous Data Link Control (IBM, SNA)

SDLGR

Specification and Description Language / Graphical Representation, "SDL/GR"

SDLLC

Synchronous Data Link Control Conversion (SNA, IBM)

SDLP

Standard Device Level Protocol

SDM

??? [benchmark] (DB, SPEC)

SDM

Short Data Message

SDM

Spatial Data Management

SDMA

Storage Device Migration Aid

SDMS

SCSI Device Management System (BIOS)

SDN

Software Defined Network

SDN

Software Distribution Net (FidoNet)

SDNS

Secure Data Network Service / System USA, mil.

SDOC

Selective Dynamic Overload Controls

TOC

SDP
: Service Delivery Point

SDP
: Software Development Plan

SDP
: Specialized Data Point (IN)

SDP
: Streaming Data Procedure

SDR
: Shared Data Research

SDR
: Signal-to-Distortion Ratio

SDR
: Store Data Register

SDR
: Streaming Data Request

SDR
: System Design Review

SDRAM
: Synchronous Dynamic Random Access Memory (RAM, DRAM, IC, Intel, Samsung)

SDRP
: Source Demand Routing Protocol

SDS
: Switched Data Service

SDS
: Synchronous Data Set

SDS
: Sysops Distribution System (BBS)

SDSAF
: Switched Digital Services Applications Forum org.

SDSC
: San Diego Supercomputer Center org., USA

SDSC
: Synchronous Data Set Controller

SDSCNET
: San Diego Supercomputer Center NETwork network, USA, "SDSCnet"

SDSL

Single line Digital Subscriber Line (DSL)

SDSMA

Slotted Digital Sense Multiple Access (MODACOM), "S-DSMA"

SDSN

Secure Data System Network

SDT

Software Development Tools

SDT

Source Data Terminal

SDT

Systems Development Tool

SDTP

[PPP] Serial Data Transport Protocol (PPP, RFC 1963)

SDU

Service Data Unit (OSI)

SDU

Software Distribution Utilities (IBM, HP)

SDV

Switched Digital Video (VOD)

SDX

Storage Data ACceleration (ATAPI, WD, CD, DVD)

SE

Service / Systems Engineer

SE

Software Engineering

SE

Standard / Special Edition (IBM, OS/2)

SE

Switching Element

SE

System Extension

SEA

Self-Extracting Archive

SEA

Society for Electronic Access org.

SEAC

Standards Eastern Automatic Computer

SEAL
: Simple and Efficient Adaptation Layer (ATM)

SEAP
: Service Element Access Point

SECAM
: SEquentiel Couleur Avec Memoire

SECB
: Severely Errored Cell Block (UNI)

SED
: Stream EDitor (Unix)

SEDAS
: Standardisiertes Einheitliches DatenAustauschSystem (EDI)

SEE
: Societe des Electriciens et Electroniciens Org., France

SEE
: Software Engineering Environments

SEE
: Systems Equipment Engineering

SEF
: Source Explicit Forwarding

SEFS
: Severely Errored Framing Seconds (DS1/E1, DS3/E3)

SEH
: Structured Exception Header

SEI
: Software Engineering Institute

SEL
: Software Engineering Laboratory

SEL
: Standard Elektrik Lorenz [ag] manufacturer

SELHPC
: South East London - High Performance Computing [centre] org., "SEL-HPC"

SEM
: Server Enhancement Module

SEM
: Standard Error of the Mean

SEPP

Secure Electronic Payment Protocol banking, IBM, Netscape, GTE

SEQUEL

Structured English QUEry Language IBM, DB, SQL, predecessor

SERM

Structured Entity Relationship Model (DB, ERM)

SERT

Security Emergency Response Team Org., Australia, Internet

SES

Security Enabling Services (IBM)

SES

Severely Errored Seconds (DS1/E1)

SESAM

Synergetische Erkennung mittels Standbild, Akustik und Motorik (IIS)

SET

Secure Electronic Transactions IBM, Visa, MS, IBM, Mastercard, Netscape, banking

SET

Software Engineering Technology

SET

Standard d'Echange et de Transfert AFNOR, France

SETA

Systems Engineering and Technical Assistance

SETL

SEt Theory Language (New York Uni.), "SetL"

SEU

Software-Entwicklungs-Umgebung (CASE)

SEU

Source Entry Utility (IBM, ADT)

SF

Service Feature (IN)

SF

Sign Flag assembler

SF

Standard Form

SF

Switching Fabric

SFBI

Shared Frame Buffer Interconnect

SFD
: Simple Formattable Document (CCITT, MHS, X.420)

SFD
: Start Frame Delimiter ethernet

SFD
: Symbolic File Directory

SFF
: Small Form Factor [committee] org.

SFMJI
: Sorry For My Jumping In slang, Usenet, IRC

SFPS
: Secure Fast Packet Switching (Cabletron)

SFQL
: Structured Full-text Query Language

SFS
: ??? [finland standards org.] Org., Finland

SFT
: System Fault Tolerance (Novell)

SFT
: System File Table (DOS)

SFTP
: Screened Foiled Twisted Pair [cable] (UTP, TP)

SFTP
: Simple File Transfer Protocol (RFC 913)

SFUG
: Security Features User's Guide

SG
: Signal Ground (MODEM)

SGA
: Shared Global Array (DEC, VMS)

SGC
: SCSI Graphic Commands (SAM)

SGI
: Silicon Graphics Incorporated manufacturer

SGM
: SeGmentation Message

SGML

TOC

Standard Generalized Markup Language (ISO 8879, JTC1, RFC 1874)

SGMLB

Standard Generalized Markup Language - Binary version (SGML), "SGML-B"

SGMP

Simple Gateway Monitoring Protocol (RFC 1028)

SGRAM

Synchronous Graphics Random Access Memory (DRAM, RAM)

SGTSI

Semi-Graphical Tree Structure Interface

SHA

Secure Hash Algorithm cryptography, NIST

SHA

Super High Aperture [LCD] (LCD)

SHED

Segmented Hypergraphics EDitor (MS, Windows, ADT)

SHF

Super-High Frequency

SHTSI

Somebody Had To Say It slang, Usenet

SHTTP

Secure HyperText Transfer Protocol (HTTP), "S-HTTP"

SHV

Standard High Volume [motherboard] (SMP, Intel)

SI

Schweizer Informatikergesellschaft org., Schweiz

SI

Source Index [register] CPU, Intel, assembler

SI

SPARC International Org., manufacturer

SI

System Information

SIA

Semiconductor Industry Association org., USA

SIAM

Society for Industrial and Applied Mathematics org., USA

SIB

Service Independent building Block (IN)

SIC
: Silicon Integrated Circuit

SIC
: Standard Industry Classification

SIC
: Subject Indicator Code

SICE
: ??? org.

SICS
: Swedish Institute of Computer Sciences org., Schweden

SID
: Signaling IDentifier

SID
: SWIFT Interface Device (SWIFT)

SID
: System IDentification

SIDF
: System Independent Data Format (Novell, SMS)

SIDM
: Serial Impact Dot Matrix [printer]

SIDR
: Service Independent Data Requester (Novell, Netware)

SIETEM
: Siemens TTCN Test Manager (TTCN, Tektronix, SNI)

SIF
: Significant Pel Field (MPEG)

SIFT
: ???

SIFTUFT
: Sender-Initiated File Transfer/Unsolicited File Transfer (RFC 1440), "SIFT/UFT"

SIG
: Special Interest Group

SIGART
: [ACM] Special Interest Group on ARTificial intelligence org., ACM, AI

SIGBIT
: Special Interest Group on Business Information Technology

SIGCAT

Special Interest Group for CD-ROM Applications Technology (CD)

SIGCPR

Special Interest Group on Computer Personal Research org., ACM

SIGCSE

Special Interest Group on Computer Science Education org., ACM

SIGDA

Special Interest Group on Design Automation org., ???

SIGG

SIGnaturGesetz (DFUE), "SigG"

SIGMA

Software Industrialized Generator and Maintenance Aids system (MITI)

SIGPLAN

Special Interest Group for Programming LANguages

SIGV

SIGnaturVerordnung (DFUE), "SigV"

SIGWEB

Special Interest Group on the world wide WEB org., UK

SII

Static Invocation Interface

SILS

Secure Interoperable LAN/MAN Standard (LAN, MAN)

SILS

Standard for Interoperable LAN Security (LAN)

SIM

Signal Interface Module

SIM

Subscriber Identity Module (GSM)

SIMD

Single Instruction [stream], Multiple Data [stream]

SIMDIS

SIMulation DISposition (MBAG)

SIMIS

SIcheres MIkroprozessor System (SNI)

SIMM

Single Inline Memory Module (IC)

SIMNET

SIMulation NETwork network

SIO
: Simultan Input Output (QMS)

SIP
: SCSI-3 Interlocked Protocol (SAM)

SIP
: Simulated Input Processor

SIP
: SMDS Interface Protocol (SMDS)

SIP
: Strategische InformationsPlanung (IM)

SIP
: Symbolic Input Program

SIPC
: Simply Interactive Personal Computer

SIPO
: Serial In Parallel Out

SIPP
: Simple Internet Protocol Plus (IP, IPV6, RFC 1710, Internet)

SIPP
: SImple Polygone Processor (Unix)

SIPP
: Single Inline Package Pin (IC)

SIPRNET
: Secret IP Router NETwork DISN, mil.

SIR
: Save Instruction Recognition

SIR
: Selective Information Retrieval

SIR
: Serial InfraRed (HP)

SIR
: Sicherheit im Rechenzentrum (TPS)

SIR
: Statistical Information Retrieval

SIR
: Substained Information Rate (SMDS)

SIRENE

Supplementary Information REquest at the National Entry SIS, Europe

SIS
: Schengen Information System Polizei, Europe

SIS
: Silicon Integrated Systems [corp.] manufacturer, Taiwan

SIS
: Software Information Services

SIS
: Stellen-Informations-Service (WWW)

SIS
: Strategic Information System

SISAL
: Streams and Iteration in a Single-Assignment Language

SISD
: Single Instruction [stream], Singke Data [stream]

SITD
: Still In The Dark slang

SIWPS
: Simple Internet White Pages Service (Internet)

SJF
: Shortest Job First

SKIA
: Secure Key Issuing Authority (TESS)

SKIP
: Simple Key-management for Internet Protocols Internet, cryptography, Sun

SLA
: Service Level Agreement

SLAM
: Simulation Language for Alternative Modeling

SLC
: Service Level Contract

SLC
: Simple Line Code [modulation]

SLDC
: Synchronous Data Link Control

SLDRAM
: SyncLink Dynamic Random Access Memory (DRAM, RAM)

TOC

SLE
: Screen List Editing

SLED
: Single Large Expensive Drive

SLFP
: Shared Frame Buffer Interconnect (ATI, Intel)

SLIC
: Serial Link and Interrupt Controller (TSMP, Wyse)

SLIC
: Subscriber Line Interface Circuit (PBX)

SLIP
: Serial Line Internet Protocol (Internet, RFC 1055), "SL/IP"

SLMR
: Silly Little Mail Reader

SLP
: Service Logic Program (IN)

SLP
: Symposium on Logic Programming conference

SLQ
: Super Letter Quality [fonts] (Star)

SLS
: Softlanding Linux System (Linux)

SLS
: Storage Library System

SLSI
: Super Large Scale Integration

SLT
: Solid-Logic Technology

SLU
: Secondary Logical Unit

SLU
: Serial Line Unit

SMA
: Standardization Management Activity

SMAE
: System Management Application Entity (OSI)

SMAF

Service Management Agent Function (IN)

SMAP

System Management Application Process (OSI)

SMART

Self-Monitoring, Analysis and Reporting Technology (HDD, Conner, IBM, Quantum, Seagate, WD), "S.M.A.R.T."

SMASE

System Management Application Service Element (OSI)

SMB

Server Message Block [protocol] (IBM, Intel, MS)

SMBA

Shared Memory Buffer Architecture (Intel)

SMBP

Sever Message Block Protocol

SMC

SCSI-3 Medium changer Commands (SAM, SCSI)

SMC

Standard Microsystems Corporation manufacturer

SMD

Storage Module Device

SMD

Surface Mounted Device

SMD

System Management Bus (Intel)

SMDAC

Single MAC Dual Attached Concentrator (FDDI, DAC, MAC)

SMDI

Storage Module Disk Interconnect

SMDL

Standard Music Description Language (ISO, IEC, CD 10743)

SMDR

Station Message Detail Recording

SMDR

Storage Management Data Requester (Novell, Netware, SMS)

SMDS

Switched Multimegabit Data Service (BELLCORE)

SMDSCNM

SMDS Customer Network Management (SMDS), "SMDS CNM"

SME

Society of Manufacturing Engineering org., USA

SME

Storage Management Engine (Novell, Netware, SMS)

SMF

Service Management Function (IN)

SMF

Single Mode Fiber (FDDI)

SMFA

Special / System Management Functional Area (OSI)

SMFF

Script Mathematical Formula Formatter

SMG

Special Mobile Group GSM, org.

SMI

Structure and identification of Management Information (OSI, RFC 1155/1902)

SMI

Sun Microsystems Inc. manufacturer

SMI

System Monitoring Interface (Informix, DB)

SMIME

Secure/Multipurpose Internet Mail Extensions MIME, MS, Lotus, Qualcomm, RSA, cryptography, "S/MIME"

SMIS

Service-Marketing-InformationsSystem (MBAG)

SMIT

System Management Interface Tool (IBM, AIX)

SMK

Software Migration Kit

SMK

Structured Meta-Knowledge

SML

Service Management Layer (TMN)

SML

Shared Memory Link (TCP/IP)

SML

Siemens Modular Link

SML

Standard Machine Language

SML

Standard Meta Language

SMLNJ

Standard Meta Language / New Jersey, "SML/NJ"

SMM

System Management Mode (CPU)

SMM

System Manager's Manual (BSD, Unix)

SMP

Software Motion Picture (DEC)

SMP

Symmetric MultiProcessor [system]

SMP

Symmetrisches MultiProzessor [system]

SMP

System Modification Program

SMPC

Shared Memory Parallel Computer (HPC)

SMPTE

Society of Motion Picture and Television Engineers org.

SMPU

Switch Module Processor Unit

SMR

Source Maintainability and Reliability

SMR

Specialized Mobile Radio [systems]

SMRT

Signal Message Rate Timing

SMS

Service Management System

SMS

Short Message Service (GSM)

SMS

Storage Management Services (Novell, Netware)

SMS
: System Management Server (MS)

SMS
: System-Managed Storage

SMSAC
: Society of Management Science and Applied Cybernetics org., Indien

SMSC
: Short Message Service Center (SMS)

SMSCEMI
: Short Message Service Center External Machine Interface [protocol] (SMS)

SMSP
: Storage Management Services Protocol (Novell, SMS)

SMT
: Segment Table Map

SMT
: Shared Memory Transport (X-Windows)

SMT
: Station ManagemenT (FDDI)

SMT
: Surface-Mount Technology

SMTA
: Subordinate Message Transfer Agent (MTA)

SMTP
: Simple Mail Transfer Protocol (RFC 821, TCP/IP)

SMUX
: SNMP MUltipleXing protocol (SNMP, MUX, RFC 1227)

SN
: Sequence Number

SN
: Serial Number

SN
: Subscriber Number (MS-ISDN, GSM)

SNA
: Systems Network Architecture (IBM)

SNAC
: SNA Network Access Controller (SNA, SDLC, IDS)

SNAC

SubNetwork ACcess [functions]

SNACP

[PPP] Systems Network Architecture Control Protocol (PPP, SNA, RFC 2043)

SNACP

SubNetwork ACcess Procedure (ISO, IS 8648, SNAC), "SNAcP"

SNADS

Systems Network Architecture Distribution Service (IBM, CCS)

SNAFU

Situation Normal All Fouled Up slang

SNAIP

[advanced] Systems Network Architecture/Internet Protocol (SNA, IP, RFC 1538), "SNA/IP"

SNAP

SubNetwork Access Protocol LAN, ethernet

SNAP

SubNetwork Attachment Point (IEEE 802.1a)

SNAP

System and Network Administration Program

SNCP

Single Node Control Point (IBM, SNA)

SNDC

SubNetwork Dependent Convergence [functions] (OSI)

SNDCP

SubNetwork Dependent Convergence Procedure (ISO, IS 8648, OSI, SNDC)

SNEPS

Semantic NEtwork Processing System (GNU, LISP), "SNePS"

SNF

Server Natural Format (Fonts, X)

SNF

Shared Network Facilities

SNG

Satellite News Gathering

SNI

Siemens Nixdorf Informationssysteme [AG] manufacturer

SNI

SNA Network Interconnection (IBM, VTAM, SNA)

SNI

Subscriber Network Interface (SMDS)

SNIC
: SubNetwork Independent Convergence [functions] (OSI)

SNICP
: SubNetwork Independent Convergence Procedure (ISO, IS 8648, OSI, SNIC)

SNMP
: Simple Network Management Protocol (RFC 1157/1902, TCP/IP, IETF)

SNNS
: Stuttgart Neural Network Simulator (IPVR)

SNOBOL
: StriNg Orientated symBOlic Language

SNP
: SubNetwork Protocol

SNPA
: Sub-Network Point of Attachment

SNPP
: Simple Network Paging Protocol (RFC 1861, SMS)

SNR
: Serial NumbeR (IMEI, GSM)

SNR
: Signal-to-Noise Ratio

SNRM
: Set Normal Response Mode (SDLC, HDLC, ADDCP, LAPB)

SNRME
: Set Normal Response Mode Extension (SNRME)

SNS
: Secure Network Server

SNTP
: Simple Network Time Protocol (RFC 2030)

SOA
: Start Of Authority record (DNS)

SOA
: State Of the Art slang

SOAP
: Symbolic Optimizer and Assembly Program

SOB
: Start Of Block

SOC

System-On-a-Chip

SOC

Systems and Option Catalog

SOCKS

SOCKet Secure

SODA

System Optimization and Design Algorithm

SODIMM

Small Outline Dual Inline Memory Module (DRAM, DIMM)

SODIS

SOftware Dokumentations- und InformationsSystem

SOE

Standard Operating Environment

SOE

Standards of Excellence

SOEP

Secondary Operand Execution Pipeline (Motorola, CPU), "sOEP"

SOFABED

[davenport] Standard Open Formal Architecture for Browsable Electronic Documents

SOH

Section OverHead

SOH

Start Of Header

SOHO

Small Office / Home Office [market]

SOIF

Summary Object Interchange Format (WWW)

SOJ

Small-Outline J-lead [chip] (IC, DRAM)

SOL

Simulation-Oriented Language

SOM

Self-Organizing Machine

SOM

Structured Object Method

SOM

System Object Model (IBM, ORB, CORBA)

SONDS
: Small Office Network Data System

SONET
: Synchronous Optical NETwork (FDDI, ATM)

SOP
: Standard Operating Procedure

SOS
: Share Operating System (OS)

SOS
: Standards and Open Systems

SOS
: Support On Site

SOS
: Symbolic Operating System (OS)

SOSP
: System Operational and Support Plan

SOTA
: State of the Art slang

SOX
: Sound EXchange [software]

SP
: Service Pack MS, Windows NT

SP
: Signal Processor

SP
: SPare (IMEI, GSM)

SP
: Speech Processing

SP
: Stack Pointer [register] CPU, Intel, assembler

SP
: Structured Programming

SP
: System Product

SPA
: Software Publishers Association org., USA

SPAG

[european] Standards Promotion and Application Group Org., Hersteller, Europe

SPAM
: Spiced Pork and hAM (Usenet, EMP)

SPANS
: Simple Protocol for ATM Network Signalling (ForeRunner, ATM)

SPAP
: ??? Protocol

SPARC
: Scalable Processor ARChitecture (Sun)

SPARC
: Standard Planning And Requirement Committee ANSI, org.

SPAT
: Speech Pronounciation Analysis Training (Uni Mainz), "S.P.A.T."

SPC
: SCSI-3 Primary Commands (SAM, SCSI)

SPC
: Stored Program Command

SPC
: Stored Program Control

SPCF
: Service Point Command Facility (IBM)

SPCS
: Stored Program Control Systems

SPD
: Serial Presence Detect (EEPROM, SDRAM)

SPD
: Software Product Description

SPD
: Software Products Division

SPDEEPROM
: Serial Presence Detect - Electronical Erasable Programmable Read Only MemorySerial Presence Detect (EPROM), "SPD-EEPROM"

SPDIF
: Sony/Philips - Digital Interface Format digital audio, Sony, Philips, "S/P-DIF"

SPDL
: Standard Page Description Language (ISO, IEC, IS 10180)

SPDN

Shared Private Data Network

SPDU

Session Protocol Data Unit (OSI, PDU, OSI/RM)

SPE

Symbolic Programming Environment

SPE

Synchronous Payload Envelope

SPEC

System Performance Evaluation Corporation org., RISC

SPF

Structured Programming Facility

SPI

SCSI-3 Parallel Interface (SAM, SCSI)

SPI

Security Parameter Index

SPI

Serial Peripheral Interface

SPI

Service Provider Interface (WOSA)

SPI

Software Products International manufacturer

SPICE

Scalable Parallel Intelligent Communications Engine

SPICE

Simulation Program with Integrated Circuit Emphasis

SPICS

Spare Parts Inventory Control System (MBAG)

SPID

Service Profile IDentifier (ISDN)

SPID

Service Protocol IDentifier

SPIN

Sponsored Programs Information Network

SPKM

Simple Public-Key [GSS-API] Mechanism (GSS, RFC 2025)

SPL

Set Priority Level (Unix)

SPL

Simple Programming Language

SPL

System Programming Language

SPM

Session Protocol Machine (OSI, ISO 8327)

SPM

Set Program Mask

SPM

Software Performance Monitor

SPM

Source Program Maintenance

SPM

System Performance Monitor

SPM2

System Performance Monitor /2 (IBM, OS/2), "SPM/2"

SPN

Substitution Permutation Network cryptography

SPOOL

Simultaneous Peripheral Operations OnLine

SPP

Scalable Parallel Processing (Intel)

SPP

Sequenced Packet Protocol

SPP

Standard Parallel Port

SPR

Software Problem Report

SPS

SpeicherProgrammierbare Steuerungstechnik

SPS

String Processing System

SPS

Symbolic Programming System

SPSL

Special-Purpose Simulation Language

SPSS
: Statistical Package of the Social Sciences

SPT
: Sectors Per Track

SPTS
: Single Program Transport Stream

SPU
: System Processing Unit

SPUCDL
: Serial Peripheral Unit Controller/Data Link, "SPUC/DL"

SPUD
: Storage Pedestal Upgrade Disk / Drive

SPUR
: Supercomputing Program for Undergraduate Research

SPX
: Sequenced Packet eXchange (Novell, Netware)

SQ
: Shielded Quart [cable]

SQ
: Signal Quality (MODEM)

SQA
: Software Quality Assurance

SQD
: Signal Quality Detector

SQFP
: Shrink Quad Flat Package (CPU)

SQL
: Structured Query Language (ISO 9075, DB, 4GL)

SQLCLI
: SQL Call Level Interface (SAG, SQL), "SQL/CLI"

SQLDA
: Structured Query Language Descriptor Area (SQL)

SQLDMO
: SQL Distributed Management Objects (MS, SQL Server, OLE, DB)

SQLDS
: Structured Query Language/Data System (IBM, VMS), "SQL/DS"

SQM

Software Quality Management

SR

Source Routing [bridging]

SR

Status Register IC, assembler

SRAM

Static Random Access Memory (RAM, IC)

SRC

Standard Context Routing (MODACOM)

SRC

System Resource Controller (AIX, IBM)

SRD

Screen Reader System

SRD

Secondary Received Data

SRD

Send and Request Data (Feldbus)

SRDF

Symmetrix Remote Data Facility

SRE

Self Routing switch Element (ATM)

SRF

Science Research Foundation org., UK

SRF

Service Resource Function (IN)

SRF

Specifically Routed Frame

SRI

Stanford Research Institute org., USA

SRM

System Resources Manager

SRN

Source/Recipient Node (IBM)

SRP

Software Reuse Program

SRP

Source Routing Protocol (IBM)

SRPI
: Server Requester Programming Interface (IBM, API)

SRPM
: Scalable Reverse Path Multicast

SRS
: Sound Retrieval System digital audio

SRT
: Secure Request Technology banking, Verschluesselung, Java

SRT
: Signal Requests Terminal

SRT
: Source Route Transparent [bridges] ethernet, Token Ring, IEEE 802.1D

SRT
: Speech Reception Threshold

SRTS
: Synchronous Residual Time Stamp

SRVIFS
: SeRVer Installable File System (IBM)

SS
: Single Sided [disk] (FDD)

SS
: Spread Spectrum (Wireless LAN)

SS
: Stack Segment [register] CPU, Intel, assembler

SSA
: Serial Storage Architecture (IBM)

SSADM
: Structured System Analysis and Design Method (DB)

SSAP
: Session Service Access Point (SAP, LLC, OSI/RM)

SSAP
: Source link Service Access Point (SAP, LLC)

SSAS
: Station Signaling and Announcement System

SSB
: Single SideBand

SSBA

Suite Synthetique des Benchmarks de l'AFFU (MP)

SSBAM
: Single SideBand Amplitude Modulation

SSC
: SCSI Stream Commands (SAM, SCSI)

SSC
: Specialized Systems Consultants

SSCF
: Service Specific Coordination Function (ATM)

SSCOP
: Service Specific Connection Orientated Protocol (ATM)

SSCP
: Service Switching & Control Point (IN)

SSCP
: System Services Control Point (NAU, SNA)

SSCS
: Service Specific Convergence Sublayer (ATM)

SSD
: Solid State Disk (HDD, RAM)

SSDA
: Synchronous Serial Data Adapter

SSDC
: Stack Segment Descriptor Cache [register] (SS, Intel, CPU)

SSDD
: Single-Sided/Double-Density [disk] (FDD), "SS/DD"

SSDD
: Solid State Disk Drive

SSDU
: Session Service Data Unit

SSE
: Simple Screen Editor

SSE
: Software Support Engineer

SSF
: Service Switching Function (IN)

SSFDC
: Solid State Floppy Disk Card (PCMCIA)

TOC

SSH
: Secure SHell (Unix, Shell)

SSI
: [Schwedisches Strahlenschutzinstitut] Schweden, org.

SSI
: Server Side Include [script] (HTTPD, CGI)

SSI
: Small Scale Integration

SSI
: Software Systems Interface

SSIMM
: Single [RAS] Single Inline Memory Module (IC, RAS), "S-SIMM"

SSL
: Secure Socket Layer (Netscape, RSA, WWW)

SSM
: Set System Mask

SSM
: Simplified Storage Management (HSM)

SSP
: Service Switching Point (IN)

SSP
: Silicon Switch Processor (Cisco)

SSP
: Standard Printer Port

SSP
: Structured Support Program

SSP
: Switch to Switch Protocol (DLSW, RFC 1795)

SSP
: System Stack Pointer

SSP
: System Support Program

SSR
: Solid-State Relay

SSSD
: Single-Sided/Single-Density [disk] (FDD), "SS/SD"

SSSNA

Server to Server Systems Network Architecture (Banyan, VINES), "SS/SNA"

SST

Simple SIPP Transition (Internet, SIPP)

SST

South Sumatra Time [+0700] (TZ)

SSTP

Screened Shielded Twisted Pair [cable] (STP, TP), "S/STP"

ST

Seagate Technology (HDD)

ST

Segment Table / Type

ST2

[internet] STream protocol 2 (Internet, ATM, RFC 1819)

STA

Spanning Tree Algorithm

STACS

Symposium on Theoretical Aspects of Computer Science conference

STAIRS

STorage And Information Retrieval System

STAM

Shared-Time Allocation Manager

STAR

Shareware Trade Association and Resources org.

STARS

Software Technology for Adaptable Reliable Systems

STB

Software Technical Bulletin

STC

Science and Technology Center (NSF, USA)

STC

Secure Transaction Channel banking, V-One, Verschluesselung

STC

SeT Carry [flag] assembler

STC

Standard Transmission Code

STD

Secondary Transmitted Data

STD
: SeT Direction [flag] assembler

STD
: State Transition Diagram

STD
: Subscriber Trunk Dialing

STDA
: StreetTalk Directory Assistance (Banyan, VINES)

STDIN
: STandarD INput

STDM
: Synchronous Time Division Multiplexer

STDOUT
: STandarD OUTput

STE
: Section Terminating Equipment (SONET)

STE
: Secure Terminal Equipment

STE
: Spanning Tree Explorer

STE
: Standard Terminal Equipment

STEP
: STandard for the External representation / Exchange of Product data definition (ISO, DP 10303, CAD)

STFT
: Short Time Fourier Transformation

STI
: SeT Interrupt [flag] assembler

STI
: Standard Tape Interface

STII
: [internet] STream protocol II (RFC 1819)

STING
: Software Technology Interest Group CERN, org.

STIX
: SmallTalk Interface to X (GNU)

STL
: Standard Template Library

STM
: Synchronous Transfer Mode (ATM)

STM
: System Master Tape

STM1
: Synchronous Transport Mode 1 (ATM, STM)

STNLCD
: SuperTwisted Nematic Lyquid Crystal Display, "STN-LCD"

STOC
: Symposium on Theory Of Computing conference

STONE
: STructured and OpeN Environment FZI Karlsruhe, Germany

STORM
: Statistically-Oriented Matrix Program

STP
: Selective Tape Print

STP
: Service Transaction Program (IBM)

STP
: Shielded Twisted Pair [cable] (TP)

STP
: Signaling Transfer Point (IN)

STP
: Software Through Pictures, "StP"

STP
: Spanning Tree Protocol (IEEE 802.1)

STP
: System Training Program

STR
: Synchronous Transmit Receive

STS
: Synchronous Time Stamps

STS
: Synchronous Transport Signal

STS3C
: Synchronous Transport System - level 3 Concatenated, "STS-3c"

STT
: Secure Transaction Technology ??? MS, banking

STT
: Surface Tunnel Tansistor (IC, DRAM, NEC)

STTL
: Standard Transistor-Transistor Logic (TTL)

STU
: Secure Telephone Unit

STV
: Sprint Telecommunications Venture org.

STX
: Start of TeXt

SU
: Screening Units

SU
: Selectable Unit

SU
: Signalling Unit

SU
: Storage Unit

SU
: Switch User (Unix)

SUDS
: Software Update and Distribution System

SUG
: Sun User's Group org., Sun, User group

SUGD
: Sun User's Group Deutschland Sun, org., User group

SUNET
: Swedish University NETwork

SUNOS
: SUN Operating System (Sun, OS, SPARC), "SunOS"

SUNVIEW
: SUN's Visual Integrated Environment for Workstations (Sun, GUI)

SURANET

Southeastern Universities Research Association NETwork network, USA, "SURAnet"

SURF
: System Utilization Reporting Facility

SUSE
: Software Und SystemEntwicklung [distribution] (Linux), "S.u.S.E."

SUSP
: System Use Sharing Protocol

SUT
: System Under Test

SUTP
: Screened Unshielded Twisted Pair [cable] (UTP, TP), "S/UTP"

SUTT
: Single User Test Tools

SV
: StandortVerteiler cable, EN 50 173

SVA
: Shared Virtual Area

SVABI
: System V Application Binary Interface (Unix, AT&T, SCD)

SVC
: Switched Virtual Circuit / Channel / Connection (ATM, PVC)

SVC
: Switched Vitual Call / Circuit (IBM, X.25)

SVCI
: Switched Virtual Circuit Identifier (SVC, ATM)

SVD
: Schweizerische Vereinigung fuer Datenverarbeitung org., Schweiz

SVD
: Simultaneous Voice/Data

SVD
: Supplementary Volume Descriptor (CD, IS 9660)

SVE
: Simple Virtual Environment

SVFS
: System V File System (Unix)

SVGA
: Super Video Graphics Array (VGA)

SVID
: System V Interface Definition (Unix, AT&T, SCD)

SVIP
: Secure Voice Improvement Program

SVMT
: System Virtual Memory Table (BS2000)

SVPC
: Single Variable Per Constraint

SVPMI
: Super VGA Protected Mode Interface (VESA)

SVR3
: System V Release 3 (Unix, OS, AT&T)

SVR4
: System V Release 4 (Unix, OS, AT&T)

SWAN
: Secure WAN ??? (RSA), "S/WAN"

SWAN
: Sun Wide Area Network (Sun, WAN)

SWEDAC
: [Staatliches Amt fuer Technische Akkreditierung] Schweden, org.

SWI
: SoftWare Interrupts (RISC, OS)

SWICO
: Schweizerischer Wirtschaftsverband der Informations-, Communikations- und Organisationstechnik org., Schweiz

SWIFT
: Society for Worldwide Interbank Financial Telecommunication Org., banking

SWIM
: Super Woz' Integrated Machine

SWISH
: Simple Web Indexing System for Humans (WAIS, WWW)

SXGA
: Super ??? eXtended Graphics Adapter

SYSAD
: SYStem ADministrator, "SysAd"

SYSOP
: SYStem OPerator (BBS), "SysOp"

TOC

Go to the previous, next section.

Go to the previous, next section.

-- T ---

T1
: [digital] Transmission link 1 ??? (DS1)

T3
: [digital] Transmission link 3 ??? (DS3)

TA
: TerminalAdapter (ISDN)

TAAI
: Taiwanese Association for Artificial Intelligence org., Taiwan, AI

TAC
: Technical Assistance Center (Cisco)

TAC
: Terminal Access Circuit

TAC
: Terminal Access Controller (ARPANET, MILNET)

TAC
: Type Approval Code (IMEI, GSM)

TACACS
: Terminal Access Controller Access Control System (RFC 1492)

TACS
: Total Access Communication Service (Mobile Systems)

TADI
: Time Assigned Data Interpolation

TADIL
: TActical Digital Information Link mil.

TADS
: Test and Debug System

TAE
: Telekommunikations-AnschlussEinheit (Telekom)

TAF
: Terminal Access Facility

TAFIM
: Technical Architecture Framework for Information Management JIEO, mil.

TAIS

Technisch-Administratives InformationsSystem

TALAE

TALigent Application Program (Taligent), "TalAE"

TALDE

TALigent Development Environment (Taligent), "TalDE"

TALISMAN

Tracing Authors' rights by Labelling Image Services and Monitoring Access Network [project] Europe

TALOS

TALigent Object Services (Taligent), "TalOS"

TAN

Table / Total Area Network

TAN

TransAction Number banking

TANJ

There Ain't No Justice slang, Usenet, IRC

TANSTAAFL

There Ain't No Such Thing As A Free Lunch slang, Usenet

TAO

Track At Once (CD-R)

TAP

Telocator Alphanumeric input Protocol (SNPP, PCIA, SMS, USA)

TAP

Terminal Access Point cable

TAP

Test Access Port (IC, IEEE 1149.1)

TAP

The Ada Project

TAP

Transport und Archivierung Produktdefinierender daten org., DIN, STEP, CIM

TAPI

Telephony Application Program Interface (Intel, MS, WOSA, CTI)

TAR

Tape ARchiver (Unix)

TAS

Tag Abuse Syndrome SGML, HTML, slang

TASI
: Time-Assigned Speech Interpolation

TASM
: Turbo ASseMbler Borland, assembler

TAT
: Theoretical Arrival Time (GCRA)

TB
: Tabular Bayes' [algorithm]

TB
: TeraByte

TB
: Transparent Bridging

TBC
: Time Base Corrector (Video)

TBCP
: Tagged Binary Control Protocol (Adobe, PS)

TBR
: Technical Basis for Regulations (ISDN)

TC
: Task Committee (IFIP)

TC
: Technical Committee (ISO)

TC
: Terminal Computer

TC
: Terminal Controller

TC
: Transaction Capabilities

TC
: Transfer Control

TC
: Transmission Control

TC
: Transmission Convergence

TCA
: Taipei Computer Association org., Taiwan

TCAM
: TeleCommunications Access Method (IBM, DFUE)

TCAP
: Transaction Capabilities Applications Part (MSC, GSM, IN)

TCB
: Task Control Block (BS2000)

TCC
: Telecommunications Center

TCF
: Transparent Computing Facility

TCH
: Traffic CHannel (GSM)

TCI
: Test Cell Input (UNI, ATM)

TCK
: Test ClocK (TAP, IC)

TCL
: Tool Command Language

TCLTK
: Tool Command Language/ToolKit (TCL, X-Windows), "TCL/TK"

TCM
: Thermal Conduction Module

TCM
: Time Compression Multiplexer

TCM
: Trellis Coded Modulation

TCNS
: Thomas Conrad Network System (LAN)

TCO
: [Schwedische Angestelltengewerkschaft] Schweden, org.

TCO
: Test Cell Output (UNI, ATM)

TCO
: Total Cost of Ownership

TCP
: Tape Carrier Package (CPU)

TCP

Test Coordination Procedure

TCP

Transmission Control Protocol (ARPANET)

TCPACO

TCP Alternate Checksum Option (RFC 1146, TCP), "TCP-ACO"

TCPBEUI

Transmission Control Protocol BIOS Extended User Interface (NETBIOS, TCP, UI)

TCPC

The Clean Personal Computer group Org., manufacturer, Grafikkarten

TCPIP

Transmission Control Protocol/Internet Protocol (RFC 793, IP), "TCP/IP"

TCPLDP

TCP extensions for Long Delay Paths (RFC 1072, TCP)

TCS

Transmission Convergence Sublayer (ATM)

TCSEC

Trusted Computer System Evaluation Criteria (Orange Book, NCSC)

TCSH

Trustedi C SHell (Unix, Shell)

TCU

Tape Control Unit

TCU

Timing Control Unit

TCU

Transmission Control Unit

TD

Transmit Data (MODEM)

TDA

TestDatenAuswerter (IC)

TDB

Task DataBase

TDB

Track Descriptor Block (UDF, CD-R)

TDC

Tape Data Controller

TDC

Terrestrial Data Circuit

TDCC
: Transportation Data Coordinating Comittee org., USA

TDD
: Telecommunications Device for Deaf

TDDSG
: TeldeDienstDatenSchutzGesetz (IUKDG)

TDG
: TeleDienstGesetz (IUKDG)

TDI
: Test Data Input (TAP, IC)

TDI
: Trusted Database Interpretation (DB)

TDJ
: Transfer Delay Jitter

TDM
: Telekom Designed Networks (Telekom)

TDM
: Time Division Multiplexing

TDMA
: Time Division Multiple Access (Mobile Systems)

TDMS
: Terminal Data Management System

TDN
: Telekom Designed Network

TDNN
: Time Delay Neural Net

TDO
: Test Data Output (TAP, IC)

TDP
: Telocator Data Protocol (PCIA, SMS, USA)

TDP
: Triton Data Path (Intel, Triton, IC)

TDR
: Time DOMAIN Reflectometer cable

TDS
: Tabular Data Strean [protocol] (Sybase)

TDU

Telesoftware Data Unit (BTX)

TE

Terminal Equipment

TEA

Telekommunikations-Anschluss-Einheit (Telekom)

TEAMA

Taiwanese Electric Appliance Manufacturer's Association org., Taiwan

TECO

Tape / Text Editor and COrrector (MIT)

TEDAX

TExt DAta eXchange [protocol] (MacOS)

TEDIS

Trade Electronic Data Interchange Systems EDI, Europe

TEI

Text Encoding Initiative [application] (SGML)

TELAS

TELephony Application System (CTI, SNI)

TELIS

TEileLogistik-InformationsSystem (MBAG)

TELNET

TELephone NETwork (Unix, Internet, RFC 854)

TELNETD

TELephone NETwork DAEMON (Unix, TELNET, DAEMON)

TEN

TransEuropean Networks network

TERENA

Trans-European Research and Education Networking Association Org., Niederlande, Europe

TERMCAP

TERMinal CAPability (Unix)

TERMINFO

TERMinal INFOrmation (Unix)

TESS

The Exponential Security System (RFC 1824)

TF

Trace Flag assembler

TFI

Terminal Facility Identifier (T-Online)

TFLOPS
: Tera FLoating-point Operations Per Second (CPU)

TFM
: Trusted Facility Manual

TFP
: Tops Filing Protocol

TFS
: Translucent File System

TFT
: Task File Table (BS2000)

TFT
: Thin Film Transistor (LCD)

TFTP
: Trivial File Transfer Protocol (UDP, RFC 1350/1782/1783/1784/1785)

TFTR
: ??? [protocol]

TGC
: Terminal Group Controller

THAMA
: Trident Hardware-Assisted MPEG-2/AC-3 (Trident, MPEG, DVD, AC-3)

THENET
: Texas Higher Education NETwork network, USA, "THEnet"

THT
: Token Holding Timer (FDDI, Token Ring)

TI
: Technical Interchange conference, IBM

TI
: Texas Instruments manufacturer

TIA
: Telecommunications Industries Associations org.

TIA
: Thanks In Advance slang, Usenet, IRC

TIA
: The Internet Adapter [software]

TID
: Technical Information Document (Novell)

TID

TOC

Touch Interactive Display

TIDL

Tool Integration Desciption Language (JCF)

TIE

Terminal Interface Equipment

TIES

Time Independent Escape Sequence (MODEM)

TIFF

Tag / Tagged Image File Format (Aldus)

TIGA

Texas Instruments Graphics Adapter (Texas Instruments)

TIL

Tech Info Library (Apple, WWW)

TIM

Token Interface Module (Token Ring)

TINA

Telecommunication Information Network Agent (IN)

TINAC

TINA Consortium org., IN, TINA, "TINA-C"

TIP

Terminal Interface Processor (ARPANET)

TIP

Transputer Image Processing

TIPC

Texas Instruments Personal Computer (TI)

TIPS

Truevision Image Paint Software (TI)

TIS

Tools Interface Standard (SCO, Unix)

TISN

Tokyo International Science Network network

TITN

??? Hersteller, France

TK

TeleKommunikation

TKO

TeleKommunikationsOrdnung

TOC

TKV
: TeleKommunikationsVerordnung

TLA
: Three Letter Acronym slang

TLAP
: Token ring Link Access Protocol (LAP)

TLB
: Translation Lookaside Buffer (CPU)

TLD
: Top Level DOMAIN (Internet)

TLI
: Transport Layer Interface

TLI
: Transport Level Interface (AT&T)

TLN
: Trunk Line Network

TLP
: Transmission Level Point

TLS
: Thread Local Storage

TLSP
: Transport Layer Security Protocol (ISO)

TLV
: Type - Length - Value

TLVTTL
: Terminated Low Voltage Transistor Transistor Logic

TM
: TeleMail BBS, Berlin, Germany, DFUE

TM
: Terminal Manager (Bull, DSA)

TM
: Tools for MIME (EMACS, GNU, MIME)

TM
: TradeMark

TM
: Traffic Management

TM

Transaction Monitor (TP)

TM

Turing Machine

TMC

Thinking Machines Corporation manufacturer

TMCC

Time-Multiplexed Communication Channel

TMDB

Tivoli Management Data Base (Tivoli, DB)

TME

Telocator Message Entry [protocol] (SNPP)

TME

Tivoli Management Environment (Tivoli)

TMF

Tivoli Management Framework (Tivoli)

TMF

Transaction Monitoring Facility (DB, Tandem)

TMG

TestMusterGenerator (IC)

TMG

The Master Genealogist

TMN

Telecommunication Management Network (IN)

TMP

Test Management Protocol

TMPDU

Test Management Protocol Data Unit (ISO 9646-1, PDU), "TM-PDU"

TMR

Transient Memory Record

TMR

Triple Modular Redundancy

TMS

Telecommunications / Telephone Management System

TMS

Test Mode Select (TAP, IC)

TMS

Time Multiplexed Switch

TMS
: Truth Maintenance System (AI)

TMSC
: Tape Mass Storage Control (DEC)

TMSCP
: Tape Mass Storage Control Protocol

TMSI
: Temporary Mobile Subscriber Identity (MM, GSM)

TMU
: Time Measurement Unit

TMUX
: Transport MUltiplXing protocol (RFC 1692), "TMux"

TN
: Terminal Node

TNC
: Terminal Node Controller

TNC
: Threaded Neill Concelman [connector]

TNEF
: Transportation Neutral Encapsulation Format (MAPI, MIME, MS)

TNFS
: Trusted Network File System (NFS)

TNL
: Technical NewsLetter (IBM)

TNLCD
: Twisted Nematic Liquid Crystal Display, "TN-LCD"

TNPC
: Taiwanese New Pc Consortium org., Taiwan

TNS
: Transit Network Selection

TNTC
: Too Numerous To Count slang, Usenet, IRC

TNVIP
: TelNet Visual Information Projection [protocol] (TELNET, VIP, RFC 1921)

TOC
: Table Of Contents (CD)

TODS

Transactions on Database Systems (ACM, DB)

TOEM

Technical Original Equipment Manufacturer

TOKREUI

TOKen-Ring Extended User Interface (IBM, Token Ring)

TOMS

Transactions on Mathematical Software (ACM)

TOOIS

Transactions on Office Information Systems (ACM)

TOOL

[conference on] Technology of Object-Orientated Languages and Systems OOP, conference

TOP

Technical and Office Protocols

TOP

The OS-9 Project (OS-9)

TOPICS

Total On-Line Program and Information Control System

TOPLAS

Transactions on Programming Languages and Systems (ACM)

TOPS

Timesharing OPerating System (DEC, OS)

TOS

The Operating System (Atari, OS)

TOS

Tramiel Operating System (Atari, OS)

TOS

Type Of Service (IP)

TP

Transaction Processing

TP

Turbo Pascal (Borland)

TP

Twisted Pair [cable] (LAN)

TP0

Transport Protocol class 0 (OSI)

TP4

Transport Protocol class 4 (OSI)

TPA
: Third Party Applications

TPA
: Transient Program Area (DOS)

TPC
: Transaction Processing Council Org., manufacturer, DB

TPCC
: Third Party Call Control

TPD
: Technical Product Documentation

TPDDI
: Twisted Pair Distributed Data Interface (FDDI, STP, Chipcom)

TPDU
: Transport Protocol Data Unit (OSI, OSI/RM, PDU)

TPE
: Twisted Pair Ethernet ethernet

TPF
: Transaction Processing Facility (IBM, MVS/XA)

TPI
: Tracks Per Inch (HDD)

TPM
: Transactions-Per-Minute

TPMS
: Transaction Processing Management System

TPPMD
: Twisted Pair Physical layer Medium Dependent (FDDI), "TP-PMD"

TPQ
: True Phone Quality (CAT)

TPRM
: ??? (Sun)

TPS
: TeleProcessing Systeme [gmbh] provider

TPS
: Transaction Processing System

TPS
: Transactions Per Second (DB, DBMS)

TPT

Twisted Pair Transceiver

TPU

Turbo Pascal Unit (TP, Borland)

TPW

Turbo Pascal for Windows (Borland)

TR

Task Register CPU, Intel, assembler

TRAP

Tandem Recursive Algorithm Process

TRIB

Transfer Rate of Information Bits

TRIF

Tiled Raster Interchange Format

TRIPS

Trade Related aspects of Intellectual Property rightS (GATT)

TRM

Technical Reference Model

TRMM

Token Ring Management Module (Token Ring)

TROFF

Typesetter New Run-OFF (Unix)

TROLI

Token Ring Optimized Link Interface (Token Ring)

TRON

The Realtime Operating system Nucleus

TRP

Token Ring serial Port

TRR

Token Ring Repeater

TRT

Token Rotation Timer (FDDI, Token Ring)

TS

Time Slot

TS

Time Stamp

TS

Traffic Shaping

TS
: Transport Stream

TS
: TriState (IC)

TSA
: Target Service Agent (Novell, Netware, SMS)

TSA
: Telecommunication Society of Australia Org., Australia

TSANET
: Technical Support Alliance NETwork Org., manufacturer, Lotus, Oracle, HP, ..., "TSANet"

TSAP
: Transport Service Access Point (OSI, OSI/RM, SAP)

TSAPI
: Telephony Server Application Programmer Interface (AT&T, Novell, API)

TSC
: Triton System Controller (Intel, Triton, IC)

TSD
: ??? color system

TSDU
: Transport Service Data Unit (OSI, OSI/RM)

TSE
: Technical Support Engineer (Sun)

TSE
: TestSteuerEinheit (IC)

TSEE
: Technical System Engineering Environment (Westmount, CASE)

TSI
: Time Slot Interchanger

TSIG
: Trusted Systems Interoperability Group org.

TSM
: Text Service Manager (Apple)

TSM
: Topology Specific Module (ODI)

TSMP
: True Symmetric MultiProcessor

TSN

Task Sequence Number (BS2000)

TSO

Telecommunications Service Order

TSO

Time Sharing Option

TSOCMS

Time Sharing Option/Conversational Monitor System (IBM, VME), "TSO/CMS"

TSOE

Time Sharing Option/ ??? (IBM), "TSO/E"

TSOP

Thin Small Outline Package (DRAM, IC)

TSP

Texture and Shading Processor (IC, Graphik)

TSP

Time Synchronization Protocol

TSP

Travelling Salesman Problem

TSP1

Test Synchronization Protocol 1+ (TTCN, ETSI), "TSP1+"

TSPI

TAPI Service Provider Interface (TAPI)

TSR

Terminate and Stay Resident

TSS

Task State Segment (Intel)

TSS

Time-Sharing System

TSSDC

Task State Segment Descriptor Cache (CPU)

TSTN

Triple SuperTwisted Nematic (LCD)

TSTS

Transaction and Switching and Transport Services (BELLCORE)

TSW

TeleSoftWare (T-Online)

TTCN

the Tree and Tabular Combined Notation (OSI, IUT)

TTF
: TrueType Font

TTFN
: Ta-Ta For Now slang, Usenet, IRC

TTL
: Time To Live (IP)

TTL
: Transistor Transistor Logic

TTP
: Trusted Third Parties cryptography

TTRP
: Time Token Rotation Protocol (FDDI)

TTRT
: Target Token Rotation Time (FDDI)

TTS
: Transaction Tracking System (DB, Netware)

TTS
: Trouble Ticketing System

TTY
: TeleTYpe

TUB
: Technische Universtaet Berlin org.

TUBA
: TCP and UDP with Bigger Addresses (TCP, UDP, RFC 1347)

TUC
: Total User Cell count (UNI)

TUCD
: Total User Cell Difference (UNI)

TUG
: TeX User's Group org., User group, TeX

TUI
: Text-Based User Interface (UI)

TUM
: Technische Universitaet Muenchen org.

TUP
: Telephone User Path (ISDN)

TUT

Transistor Under Test

TVOL

TV OnLine cable service (WorldGate, Internet)

TVOS

Terminal Velocity Operating System (3drealms)

TVS

Transparent Voice Signalling (VOFR)

TWAIN

Technology Without An Important Name

TWB

Terrestrial Wide-Band [networking]

TWIP

[one] TWentIeth of a Point

TZ

Time Zone (Internet)

Go to the previous, next section.

Go to the previous, next section.

TOC

-- U ---

UA
: Unnumbered Acknowledgement

UA
: User Agent (MHS, OSI)

UAA
: UnternehmensAnwendungsArchitektur (IM)

UAB
: Unix Appletalk Bridge (Apple, AppleTalk, Unix)

UAC
: Universal Access Control (IBM)

UADPS
: Uniform Automatic Data Processing System

UAE
: Unknown / Unrecoverable Application Error (Windows)

UAF
: User Authorization File

UAPDU
: User Agent Protocol Data Unit (PDU)

UART
: Universal Asynchronous Receive and Transmit

UAS
: UnAvailable Seconds (DS1/E1, DS3/E3)

UBA
: UniBus Adapter (DEC)

UBE
: Unsolicited Bulk Email spam, Internet

UBNI
: Ungermann-Bass Network Interface

UBR
: Unspecified Bit Rate (ATM, CBR, VBR, ABR, QOS)

UCE
: Unsolicited Commercial E-mail Usenet, spam

UCI
: User-Computer Interface

UCLA
: University of California at Los Angeles org., USA

UCP
: Universal Computer Protocol SMS, Europe

UCR
: Under Color Removal (DTP)

UCS
: Universal [multiple-octet] coded Character Set (ISO, IEC, DIS 10646)

UCS
: Universal Classification System

UCSB
: University of California at Santa Barbara org. USA

UCSD
: University California San Diego org., USA

UDB
: Universal Database Server (IBM, DB)

UDC
: Universal Decimal Classification

UDF
: Universal Disc Format (CD, OSTA)

UDF
: User Defined Function

UDK
: UmweltDatenKatalog (NUIS-SH)

UDLI
: ??? [hardware desription language] (HDL), "UDL/I"

UDMA
: Ultra Direct Memory Access (DMA, ATA)

UDP
: Usenet Death Penalty Usenet, spam

UDP
: User Datagram Protocol (Internet, RFC 768)

UDPIP
: User Datagram Protocol / Internet Protocol, "UDP/IP"

UDT

User-defined DataType (DB)

UDVM

Universal Data Voice Multiplexer

UEIDE

Ultra Enhanced Integrated Drive Electronics (IDE, HDD)

UEV

User End of Volume

UFS

Universal File System

UFS

Unix File System (Unix)

UFST

Universal Font Scaling Technology (Agfa)

UHC

United Hitech Corporation manufacturer, Taiwan

UHF

Ultra High Frequency

UHL

User Head Label

UI

Unix International manufacturer, Unix

UI

User Interface

UIC

User Identification Code

UID

Unit IDentifier cryptography, EES

UID

User IDentification

UIDL

Unique ID Listing (POP3, RFC 1939)

UIL

User Interface Language

UIMS

User Interface Management System

UIS

UmweltInformationsSystem

TOC

UISRM
: User Interface System Reference Model

UKERNA
: United Kingdom Education and Research Networking Association org., UK

ULA
: Uncommitted Logic Array

ULANA
: Unified Local Area Network Architecture

ULCC
: University of London Computer Center org., UK

ULP
: Upper Layer Protocols (FC)

ULSI
: Ultra Large Scale Integration

UMA
: Unified Memory Architecture

UMA
: Universal Measurement Architecture (Unix, X/Open)

UMA
: Upper Memory Area (Intel)

UMADS
: Universal Measurement Architecture Data Storage (UMA)

UMB
: Upper Memory Block (Intel, UMA)

UMC
: United Microelectronics Corporation manufacturer

UME
: UNI Management Entity (UNI, ILMI)

UML
: Unified Method Language (CASE)

UML
: Unified Modelling Language (OOP)

UMTS
: Universal Mobile Telecommunications System (IN, Mobile Systems)

UNA
: Universal Network Architecture

UNARP

UNsolicited Address Resolution Protocol (ARP, RFC 1868)

UNC

Universal Naming Convention (IBM, MS, Novell, LAN)

UNCID

UNiform rules of Conduct for Interchange of Trade data by teletransmission (EDIFACT)

UNCLE

Unix Net for Computer security in Law Enforcement org., USA, Unix, "U.N.C.L.E."

UNEDIFACT

United Nations EDIFACT (ISO 9735, EDIFACT), "UN/EDIFACT"

UNGTDI

United Nations Guidelines for Trade Data Interchange UN/EDIFACT, predecessor, "UN/GTDI"

UNI

Universal Network Interface (Cogent)

UNI

User Network Interface

UNIVAC

UNIVersal Automatic Computer

UNMA

Unified Network Management Architecture

UNSM

United Nations Standard Message, (UN/EDIFACT)

UNTDED

United Nations Trade Data Elements Directory (EDIFACT)

UNTDID

United Nations Trade Data Interchange Directory (EDIFACT)

UP

Uni Processor [system]

UPA

??? (Sun, SMP)

UPAM

User Primary Access Method (BS2000)

UPC

Universal Product Code (EAN)

UPC

Usage Parameter Control (UNI, ATM)

UPL

User Program Language

UPM
: Umdrehungen Pro Minute (HDD)

UPM
: User Profile Management (IBM)

UPN
: Umgekehrte Polnische Notation

UPP
: Universal Procedure Pointer (AE, Apple)

UPPS
: Universal Portable Protocol Stack / Support (Schneider & Koch)

UPS
: Uninterruptible Power Supply

UPS
: Unix Print Services (Unix)

UPT
: Universal Personal Telecommunications (IN)

UPT
: Universelle Personengebundene Telekommunikation (IN)

URA
: Uniform Resource Agent (WWW)

URC
: Uniform Resource Citation (WWW)

URC
: Uniform Resources Characteristics (URI, WWW)

URI
: Uniform Resource Identifier (WWW, RFC 1630)

URL
: Uniform Resource Locator (WWW, RFC 1738)

URN
: Uniform Resource Name (WWW, RFC 1737)

URSN
: Unique Resource Serial Number (URI, WWW)

US
: Unit Separator (BTX, VPCE)

USA
: United Software Association org., USA

USACNII

US Advisory Council on the National Information Infrastructure org., USA

USART
: Universal Synchronous Asynchronous Receiver / Transmitter (IC)

USB
: Universal Serial Bus (Intel)

USCP
: Unicos Station Call Processor [protocol] (Cray, MPP)

USD
: User's Supplementary Documents (BSD, Unix)

USDC
: Universal Switched Data Capability

USDN
: United States Digital Network

USEC
: User-based SECurity [model] (SNMP)

USENET
: USErs' NETwork (Internet)

USG
: Unix Support Group org., Unix

USITA
: United States Independent Telephone Association org., USA

USL
: Unix Systems Laboratories (AT&T, Unix)

USMTF
: United States Message Text Format (USA)

USP
: User Stack Pointer

USR
: U.S. Robotics manufacturer

USR
: User Service Routines

USRT
: Universal Synchronous Receiver/Transmitter (IC)

USTA
: United States Telephone Association org., USA

USV
: Unterbrechungsfreie Strom-Versorgung

USWC
: Uncached Speculative Write Combining (CPU)

UT
: Universal Time [+0000] (TZ, GMT)

UT
: Upper Tester

UTC
: Universal Time Coordinated (DCE)

UTC
: Universal Time Coordinates [+0000] (TZ, GMT)

UTI
: Universal Text Interchange

UTL
: User Trailer Label

UTLB
: Unified Translation Look-aside Buffer (CPU)

UTM
: Universal Transaction Monitor

UTM
: Universeller TransaktionsMonitor (BS2000, Sinix, TP)

UTOPIA
: Universal Test & Operations Physical layer Interface for ATM (PL, ATM)

UTP
: Unshielded Twisted Pair [cable] (TP)

UTS
: Universal Timesharing System

UTTC
: Universal Tape-To-Tape Converter

UUCICO
: Unix to Unix Copy Incoming Copy Outgoing (Unix)

UUCP
: Unix to Unix Copy Protocol (Unix)

UUG
: Unix User Group org., Unix, User group

UUID
: Universal Unique IDentifier

UUNET

Unix to Unix NETwork org., ISP

UUT

Unit-Under-Test

UVL

User Volume Label

UW

Ultra Wide [SCSI] (SCSI)

Go to the previous, next section.

Go to the previous, next section.

TOC

-- V ---

VA
: Virtual Acoustic

VAC
: Visual Age C++ (IBM)

VACS
: Visual Access Control System (SNI, CCD)

VAD
: Value Added Driver

VAD
: Voice Activity Detection (GSM)

VADD
: Value Added Disk Driver

VAFC
: VESA Advanced Feature Connector (VESA)

VAG
: VRML Architecture Group org., VRML

VAGI
: VESA Advanced Graphic Interface (VESA)

VAM
: Verband der Anbieter von Mobilfunkdiensten org.

VAMAS
: Versailles project on Advanced Materials And Standards (ISO)

VAN
: Value Added Network

VANS
: Value Added Network Services

VAP
: Value Added Process Netware, NLM, predecessor

VAPI
: Virtual [ISDN] Application Programming Interface (ISDN, CAPI, API)

VAPI
: Voyetra Applications Program Interface (API)

VAR
: Value-Added Reseller

VAS
: Value Added Services

VASCAR
: Visual Average Speed Computer And Recorder

VAST
: Virtual Archive Storage Technology

VAT
: [UDF] Virtual Allocation Table (UDF, CD-R)

VAX
: Virtual Address eXtension (DEC)

VB
: Visual BASIC (MS, BASIC)

VBA
: Visual BASIC for Applications (MS, BASIC)

VBE
: VGA standard BIOS Extentions (VESA, VGA, BIOS)

VBN
: Vermittelndes Breitband-Netz

VBNS
: Very highspeed Backbone Network Service (NSF, ATM), "vBNS"

VBR
: Variable Bit Rate (ATM, CBR, ABR, UBR, QOS)

VBRNRT
: Variable Bit Rate - Non Real Time (VBR, ATM), "VBR-NRT"

VBRRT
: Variable Bit Rate - Real Time (VBR, ATM), "VBR-RT"

VBX
: Visual Basic eXtentions (MS)

VC
: Virtual Channel / Circuit (ATM)

VC
: Virtual Console

VCA
: Voltage Controlled Amplifier

VCC

Virtual Channel Connection (ATM)

VCF
: Voltage Controlled Filters

VCI
: Video Cursor Interface (VESA)

VCI
: Virtual Channel Identifier (ATM)

VCI
: Virtual Channel Identifier (ATM)

VCI
: Virtual Connection Identifier (ATM)

VCL
: Virtual Channel Link (UNI, ATM)

VCL
: Visual Component Library (Borland, Delphi)

VCO
: Voltage Controlled Oscillator

VCOS
: Visible Caching Operating System (DSP)

VCP
: Vector Control Processor

VCPI
: Virtual Control Program Interface (DOS, Intel)

VCS
: Virtual Circuit System

VDAFS
: ??? [Format zum Austausch geometrischer Informationen] (DIN, CIM)

VDASEDAS
: ??? / Standardisiertes Einheitliches DatenAustauschSystem (EDI), "VDA/SEDAS"

VDD
: Virtual Display Driver

VDE
: Verband Deutscher Elektrotechniker [e.v.] org.

VDI
: Verein Deutscher Ingenieure org.

VDI
: Video Device Interface (Intel)

VDI
: Virtual Device Interface

VDIF
: VESA Display Information File (VESA)

VDM
: Virtual DOS Machine (OS/2, Windows NT, DOS)

VDS
: Virtual DMA Services (DMA)

VDSL
: Very high data / bit rate Digital Subscriber Line (DSL)

VDT
: Video Display Terminal

VDU
: Visual Display Unit

VDV
: Vorbestellte DauerwaehlVerbindung (Telekom)

VE
: Virtual Environment (VR)

VEC
: Virtual Embedded Circuitry (IC)

VEG
: Very Evil Grin slang, Usenet, IRC

VERA
: Just playing around, huh? For this one, look at the top of the page.

VERNET
: Virginia Educational Research NETwork network, USA, "VERnet"

VERONICA
: Very Easy Rodent-Orientated Netwide Index of Computerized Archives (Internet)

VESA
: Video Electronics Standards Association org.

VEST
: VAX Environment Software Translator

VFAT
: Virtual File Allocation Table (MS, Windows 95)

VFC
: V. Fast Class (MODEM), "V.FC"

VFEA

VMEbus Futurebus+ Extended Architecture

VFS
: Virtual File System (Unix, Linux, OSF/1)

VFW
: Video fuer Windows (MS), "VfW"

VG
: Volume Group (LVM)

VGA
: Video Graphics Array (IBM)

VGDA
: Volume Group Descriptor Area (AIX, LVM, IBM)

VGF
: 100VoiceGrade anylan Forum manufacturer

VGI
: Virtual Graphics Interface

VHDCI
: Very High Density Cable Interconnect (SCSI)

VHDL
: VHSIC Hardware Description Language (ASIC)

VHF
: Very High Frequency

VHRCD
: Very High Resolution Color Display

VHSIC
: Very High Speed Integrated Circuit

VI
: VIsual editor (Unix)

VIA
: VAX Information Architecture (VAX, DEC)

VIAD
: VESA Image Area Definition (VESA)

VIC
: V.35 Interface Cable cable

VIC
: Vendor Independent ??? (VIM)

VIDL
: VIrus Description Language, "ViDL"

VIF
: Virtual Interrupt Flag (Intel, CPU)

VIFA
: Victoria Free-Net Association org., USA

VILE
: VI Like EMACS (Unix, VI, EMACS)

VIM
: Vendor Independent Messaging (Lotus, Borland, IBM, Novell)

VIMF
: Vierer [kabel] In MetallFolie (VDE, SQ), "ViMF"

VINCE
: Vendor Independent Network Control Entity

VINES
: VIrtual NEtwork Software (Banyan, NOS)

VIO
: Virtual Input/Output

VIP
: Visual Information Projection [terminal] (Bull)

VIP
: VLB - ISA - PCI [board] (VLB, ISA, PCI)

VIPNET
: Voice Interactive Paging Network network, MAN, "VIP-Net"

VIQR
: VIetnamese Quoted-Readable [specification] (VISCII, RFC 1456)

VIR
: Visual Information Retrieval (DB, Informix)

VIROS
: VIRtual memory Operating System (DEC, OS)

VIS
: Verlaessliche InformationsSysteme (GI)

VIS
: Virtual Instruction Set (Sun, CPU)

VIS
: Visual Interactive Simulation

VISCII
: VIetnamese Standard Code for Information Interchange

VITAL

VHDL Initiative Toward ASIC Libraries (ASIC, VHDL)

VITC

Vertical Interval Time Code (Video)

VK

VideoKonferenz

VLA

Volume Licence Agreement (Novell)

VLAN

Virtual Local Area Network (LAN, IEEE 802.1q)

VLB

VESA Local Bus (VESA)

VLD

Variable Length Decoder (MPEG)

VLDB

Very Large DataBase (DB)

VLF

Very-Low-Frequency Band

VLF

Virtual Lookaside Facility (IBM)

VLIW

Very Long Instruction Word (CPU, IC)

VLM

Virtual Loadable Modul (Novell, Netware)

VLM

Volume Logical Module (IBM, OS/2)

VLR

Visitor Location Register (LR, GSM)

VLSI

Very Large Scale Integration

VM

Virtual Machine (IBM)

VM

Virtual Memory (OSF)

VMA

Virtual Memory Address

VMB

Virtual Machine Boot (IBM, OS/2)

VMC
: VESA Media Channel (VESA)

VMCF
: Virtual Machine Communications Facility (IBM, VM)

VMCMS
: Virtual Machine / Conversational Monitoring System (IBM, VM), "VM/CMS"

VMD
: Virtual Manufacturing Device (MMS)

VME
: Virtual Machine Environment (IBM, VM)

VMM
: Virtual Machine / Memory Manager (IBM, VM)

VMOS
: Vertical Metal Oxide Semiconductor (IC), "V-MOS"

VMP
: Virtual MultiPorting (POWER, CPU)

VMS
: Virtual Memory [operating] System (DEC, OS)

VMS
: Voice Mail System

VMS
: Voice Management System

VMSP
: Virtual Machine / System Product (IBM, VM), "VM/SP"

VMT
: Virtual Method Table

VMTP
: Versatile Message Transaction Protocol (RFC 1045)

VMTSS
: Virtual Machine Time-Sharing System (IBM)

VNA
: Virtual Network Architectur (Ungermann-Bass)

VNA
: Virtual Network Architecture (ATM)

VNCA
: VTAM Node Control Application (IBM, VTAM)

VNF

Virtual Network Feature

VNL

Via Net Loss plan

VNLF

Via Net Loss Factor

VNSM

VINES Network and Systems Management (Banyan, VINES)

VOD

Video On Demand

VOFR

Voice Over Frame Relay

VOI

Verband Optischer Informationssysteme org.

VOIP

Voice Over IP (IP, Internet, CTI)

VON

Voice On the Net [coalition] (USA, Internet)

VOODOO

Versions Of Outdated Documents Organized Orthogonally (Mac)

VP

Virtual Path (ATM)

VPA

VAX Performance Advisor (DEC, VMS, VAX)

VPC

Virtual Path Connection (ATM, VP)

VPCD

Videotex Presentation Control Element (BTX, VPDE)

VPCI

Virtual Path Connection Identifier (VP, ATM)

VPD

Vital Product Data

VPDE

Videotex Presentation Data Element (BTX)

VPDN

Virtual Private Data Network

VPI

Virtual Path Identifier (ATM, VP)

VPI
: Virtual Private Internet (TradeWave)

VPL
: Virtual Path Link (UNI, ATM)

VPL
: Visual Programming Languages

VPMI
: Virtual Protected Mode Interface

VPN
: Virtual Private Network

VPP
: Value Purchase Plan (Adobe)

VPT
: Virtual Path Terminator (UNI, ATM)

VPU
: Video Processing Unit

VPUG
: Ventura Publishing User Group org., DTP, User Group

VR
: Virtual Reality

VR
: Virtual Route (SNA, PSDN)

VR
: Virtuelle Realitaet

VRAI
: Virtual Reality Annual International [symposium] IEEE, conference

VRAM
: Volatile Random Access Memory (RAM, IC)

VRC
: Vertical Redundancy Check

VRD
: Virtual Retinal Display (VR)

VRM
: Virtual Resource Manager

VRM
: Voltage Regulation Module (IC)

VRML

Virtual Reality Modeling Language (VR, ISO/IEC 14772)

VROOMM

Virtual Realtime Object Oriented Memory Manager (OOP)

VRS

Voice Response System

VS

VINES Security (Banyan, VINES)

VSA

Virtual Server Architecture

VSAM

Virtual Storage Access Method

VSAT

Very Small Aperture Terminal

VSCE

Videotex Service Control Element (BTX, VPDE)

VSE

Virtual Storage Extended (IBM)

VSEESA

Virtual Storage Extended/Enterprise Systems Architecture, "VSE/ESA"

VSESP

Virtual Storage Extended/System Product (IBM), "VSE/SP"

VSI

Verband der SoftwareIndustrie deutschlands [e.v.] org.

VSM

??? (Norm f. Tastatur)

VSM

Virtual Storage Management

VSN

Volume Serial Number (BS2000)

VSP

Virtual Single Processor

VSR

Voice Storage and Retrieval

VSS

Voice Storage System

VST

VermittlungsSTelle (Telekom), "VSt"

VST
: Virtual Studio Technology (Steinberg, Audio)

VSU
: Video Service Unit

VT
: Virtual Terminal (DEC)

VT100
: Virtual Terminal 100 (DEC), "VT-100"

VTAM
: Virtual Telecommunications Access Method (IBM, SNA)

VTAME
: Virtual Telecommunicaitons Access Method Entry (VTAM)

VTD
: Virtual Tape Device

VTI
: Virtual Terminal Interface

VTM
: Verband der Telekommunikationsnetz- und Mehrwertdiensteanbieter org.

VTOC
: Volume Table of Contents

VTP
: Virtual Terminal Protocol

VTS
: Video Teleconferencing System

VTS
: Virtual Terminal Service (OSI)

VU
: Virtual User

VUD
: Verband der Unterhaltungssoftwareindustrie Deutschlands org.

VUE
: Visual User Environment (HP)

VUI
: Virtual User Interface (AMS, UI)

VUMA
: VESA Unified Memory Architecture (VESA)

VUMASBE

VESA Unified Memory Architecture - System BIOS Extension (VESA), "VUMA-SBE"

VUP

VAX Units of Performance, "VUPs"

VV

Virtuelle Verbindung

VVIDD

VESA Video Interface for Digital Displays (VESA, LCD)

Go to the previous, next section.

Go to the previous, next section.

-- W ---

WABI
: Windows Application Binary Interface (MS, Windows)

WAI
: Web Application Interface (WWW, Netscape)

WAIS
: Wide Area Information Service (Internet)

WAITS
: Westcoast Alternative to ITS

WAM
: Warren Abstract Machine (PROLOG)

WAMIS
: Wireless, Adaptive and Mobile Information Systems (ESTO)

WAN
: Wide Area Network

WAND
: Wide Area Network Distribution (WAN)

WAP
: Wissenschaftliche ArbeitsPlatzrechner

WAT
: West Africa Time [-0100] (TZ)

WATS
: Wide Area Telephone Service

WB
: WorkBench (Amiga, Commodore)

WBC
: Wide Band Channel (FDDI)

WBI
: Web Browser Intelligence (WWW, IBM)

WBS
: WissensBasierte Systeme (KI)

WC
: Word Count (Unix)

WCB
: Write Combining Buffer (CPU)

WCCE
: World Conference on Computers in Education IFIP, conference

WCGA
: World Computer Graphics Association org.

WCS
: Writable Control Store (VAX)

WD
: Western Digital [corporation] manufacturer

WD
: Working Draft

WDK
: Word Developers Kit (MS)

WDM
: Wavelength Division Multiplexing [protocol]

WDM
: Win32 Driver Model (MS)

WDP
: WatchDog Process

WDS
: WatchDog Server

WDT
: Watch Dog Timer

WDT
: Western [european] Daylight Time [+0100] (TZ, WET)

WEEB
: Western Europe EDIFACT Board org., EDIFACT, "WE/EB"

WEFT
: Web Embedding Font Tool (MS, WWW)

WELL
: Whole Earth 'Lectronic Net network

WESTNET
: WESTern regional NETwork network, USA, "Westnet"

WET
: Western European Time [+0000] (TZ, WDT)

WFC

Wait for Caller (BBS)

WFM
: Wired For Management (Intel), "WfM"

WFW
: Windows For Workgroups (MS), "WfW"

WG
: Working Group (SC, ISO, IEC)

WHCA
: White House Communications Agency (DISA)

WHOLIS
: World Health Organization Library Information System org., UNO

WHOSIS
: World Health Organization Statistical Information System org., UNO

WHQL
: Windows Hardware Quality Lab (MS, Windows, PC97)

WIDD
: Web InformationsDienst Deutschland (WWW, Neuss)

WIF
: Web Interface Facility (WWW, MVS, OS/390)

WIMP
: Window, Icon, Menu, Pointing device

WIN
: WIssenschaftsNetz network, DFN

WINHEC
: WINdows Hardware Engineering Conference MS, Windows, conference, "WinHEC"

WINLAB
: Wireless Information Network LABoratory org., STC, USA

WINS
: Windows Internet Name Service (MS, Windows NT)

WISE
: World-wide Information System for r&d Efforts (WWW, IGD)

WITT
: Workstation Interactive Test Tool (IBM)

WIZOP
: WIzard sysOP, "WizOp"

WKS
: Well Known Services (DNS, Internet)

WLL
: Wireless Local Loop

WLO
: Windows Library Objects

WLS
: White Line Skip (Fax)

WMS
: Warehouse Management System (DB)

WMS
: Workflow Management Systeme

WNIM
: Wide area Network Interface Module

WORDIA
: WORD Internet Assistent (MS)

WORM
: Write Once Read Many (CD)

WOSA
: Windows Open System Architecture (MS)

WOSAXCEM
: WOSA eXtensions for Control, Engineering and Manufacturing (WOSA, MS)

WOSAXFS
: WOSA eXtensions for Financial Services (WOSA, MS), "WOSA X FS"

WOSAXRT
: WOSA eXtensions for Real-Time market data (WOSA, MS)

WOSC
: World Organization of Systems and Cybernetics Org., France

WOW
: Windows On Windows

WP
: Word Perfect

WPS
: Word Processing Software

WPS
: WorkPlace Shell (OS/2, IBM, Shell)

WRAM
: Window Random Access Memory (RAM, IC, Samsung, Matrox)

WRB

Web Request Broker (Oracle, WWW)

WRT

Whitewater Ressource Toolkit (Windows, TPW)

WRT

With Respect To slang, Usenet, IRC

WSF

Work Station Feature (IBM)

WSF

Work Station Function (IN)

WSP

Workstation Security Package

WTS

Web Transaction Security (WWW)

WVNET

West Virginia Network for Educational Telecomputing network, USA

WVSO

WiederVerwendbare SoftwareObjekte

WWDC

World Wide Developer Conference (Apple)

WWIMS

Worldwide Warning Indicator Monitoring System mil.

WWMCCS

Worldwide Military Command and Control System mil., Vorlaeufer, GCCS

WWOLS

World Wide On-Line System mil.

WWW

World Wide Waiting slang

WWW

World Wide Web (Internet)

WYSBYGI

What You See Before You Get It (DTP)

WYSIWIS

What You See Is What I See

WYSIWYG

What You See Is What You Get (DTP)

Go to the previous, next section.

Go to the previous, next section.

-- X ---

X11
: X window System version 11 (X-Windows)

XA
: eXtended Architecture (IBM)

XACT
: Xante's Accurate Calibration Technology (Xante), "X*ACT"

XAPIA
: X.400 API Association org., X.400, API

XBP
: XBase Parts (Xbase/2, OS/2, DB)

XCCI
: X Common Client Interface, "X CCI"

XCEM
: eXtensions for Control, Engineering and Manufacturing (API, MS, Windows)

XCHS
: eXtended Cylinder Head Sectors (EIDE)

XCMOS
: eXtended CMOS (IC, CMOS)

XCMS
: X Color Management System, "Xcms"

XCOFF
: eXtended Common Object File Format (Unix)

XCUTS
: X Commands and Utilities Test Suite (X/Open)

XDMCP
: X Display Manager Control Protocol

XDR
: eXternal Data Representation (ONC, RPCL, Sun, RFC 1832)

XES
: Xerox Engineering Systems (Xerox)

XFCB
: Extended File Control Block

XFE
: X Front End

XFS
: eXtended ??? File System (JFS, SGI, Unix)

XGA
: eXtended Graphics Adapter (IBM, PS/2)

XID
: [SNA] eXchange IDentifier (SNA)

XIE
: X Image Extension

XIO
: eXtended Input/Output ??? (Novell, Netware)

XMAPI
: eXtended Messaging Application Program Interface (API)

XMI
: eXtended Memory Interconnect (DEC)

XML
: eXtensible Markup Language

XMM
: eXtended Memory Manager (DOS, LIM)

XMP
: X/open Management Protocol (X/Open)

XMS
: eXtended Memory Specification (DOS, LIM)

XMS
: eXtended Multiprocessor operating System (OS)

XN
: eXecution Node

XNS
: Xerox Network Services / Systems / Standard (Xerrox)

XNSCP
: [PPP] XNS IDP Control Protocol (RFC 1764, Xerox, PPP, XNS, IDP)

XNSIDP
: Xerox Network Services - Internet Datagram Protocol (Xerox, Internet), "XNS-IDP"

XNSITP
: Xerox Network Services/Internet ??? Protocol (Xerox), "XNS/ITP"

XOFF

eXchange OFF (MODEM)

XON

eXchange ON (MODEM)

XOT

X.25 Over TCP (Cisco, X.25, TCP, RFC 1613)

XPC

X Performance Characterization [group] (GPC)

XPG

X/open Portability Guide (X/Open)

XPGR3

X/open Portability Guide Release 3 (X/Open)

XPS

eXPert System (AI)

XPS

eXtended Parallel Server (DB, Informix)

XPSB

X - Protocol Stream Benchmark (XPC)

XRIP

eXtended ??? Routing Information Protocol (Novell, Netware)

XRT

eXtensions for Real-Time market data (API, MS, Windows)

XSMD

eXtended Storage Module Driver interface

XT

eXtended Technology (IBM, PC)

XTI

X/open Transport Interface (X/Open)

XTP

eXpress Transport Protocol

XUI

X-windows User Interface (UI)

XUIDL

X-Unique-ID-Listing. (UIDL, POP3), "X-UIDL"

XUMA

eXpertensystem UMweltgefaehrlichkeit von Altlasten (XPS)

XVT

eXecutive Vector Table (BS2000)

Go to the previous, next section.

TOC

Go to the previous, next section.

TOC

-- Y ---

YAAS
: Yet Another Acronym Server (WWW)

YABA
: Yet Another Bloody Acronym slang

YACC
: Yet Another Compiler Compiler (Unix)

YADE
: Yet Another DSSSL Engine (DSSSL)

YAFIYGI
: You asked for it, you got it Usenet, IRC, slang

YAHOO
: Yet Another Hierarchically Officious Oracle (WWW)

YASOS
: Yet Another Scheme Object System

YAST
: Yet Another Setup Tool (Linux, SUSE), "YaST"

YAUN
: Yet Another Unix Nerd

YAY
: Yet Another YACC (YACC, Bull)

YCPS
: Yale Center for Parallel Supercomputing org., USA, HPC

YHTWFWYWTS
: You Have To Wait For What You Want To See slang, WYSIWYG

YMMV
: Your Mileage May Vary slang

YSM
: Yourdon Structured Method

YST
: Yukon Standard Time [-0900] (TZ)

Go to the previous, next section.

Go to the previous, next section.

-- Z ---

ZAK
: Zero Administration Kit ZAW, MS, Windows NT

ZAPP
: Zero Assignment Parallel Processor

ZAW
: Zero Administration initiative for Windows MS, Windows NT

ZBR
: Zone Bit Recording

ZC
: Zone Co-ordinator (FidoNet)

ZCAV
: Zone Constant Angular Velocity

ZCV
: Zortech Class Viewer (Zortech)

ZDS
: Zenith Data Systems manufacturer

ZDVA
: Zentrale DatenVerarbeitungsAnlage [des bundes]

ZE
: Zentraleinheit (CPU)

ZEAS
: ZEntrales AbsatzanalyseSystem (MBAG)

ZEBES
: Zentrales teile-BEstandsSteuerungssystem (MBAG)

ZEDAT
: ZentralEinrichtung DATenverarbeitung (FUB, Org)

ZEDIS
: ZEntrales teile-DispositionsSystem (MBAG)

ZEPRAS
: ZEntrales PReisabwicklungs- und AnalyseSystem (MBAG)

ZESOD
: ZEntrale teileSOrtimentDatei (MBAG)

ZEVIS
: ZEntrales VerkehrsInformationsSystem (Kraftfahrzeugbundesamt)

ZIAM
: Zentrum fuer Industrielle Anwendungen Massiver parallelitaet [gmbh] org.

ZIB
: [konrad-zuse] Zentrum fuer Informationstechnik Berlin org.

ZIF
: Zero Insertation Force [socket] (IC)

ZIL
: Zork Implementation Language (Infocom)

ZIP
: Zigzag Inline Package (VRAM)

ZIP
: Zone Information Protocol (AppleTalk)

ZISC
: Zero Instruction Set Computer (CPU)

ZIT
: interdisziplinaeres forschungsZentrum fuer Informatik und Technik org., Uni Paderborn

ZKA
: Zentraler Kreditausschuss banking, Org.

ZMH
: Zone Mail Hour (FidoNet)

ZMT
: Zentrum fuer Multimediale Telekommunikation (BERKOM)

ZOC
: Zap-O-Comm (OS/2, DFUE)

ZPR
: Zentrum fuer Paralleles Rechnen Org., Uni Koeln, Germany

ZPR
: Zero Power Resistance

ZRE
: Zero Rate Error

ZRZ
: Zentrales RechenZentrum org., TUB

ZSI
: Zentralstelle fuer Sicherheit in der Informationstechnik org., BSI, Vorlaeufer

ZSL

Zero Slot LAN (LAN)

ZWEI

ZWEI Was EINE Initially (EINE, LISP)

ZZF

Zentralamt zur Zulassung von Fernmeldeeinrichtungen org., Telekom, Vorlaeufer, BZT

Go to the previous, next section.

Go to the previous, next section.

TOC

About...

V.E.R.A. is a free list of acronyms all of which are used in the field of computing.

V.E.R.A. is primarily meant to be used as an online reference, although some efforts have been taken to make its TeX output looking acceptable. However I doubt that somebody would like to print it. The original release compiled to 250 pages.

This edition is a special contribution to the GNU project similar to the version 4.4 of V.E.R.A. It contains approximately 7433 acronyms.

The idea to make V.E.R.A. available in Texinfo fomat came after reading two other Info files both of which are available on the Internet, the *Standards* and the *Languages* file, both dealing with antique DEC mainframe environments and both distributed stand-alone; that is, both are not serving as a software manual like most of Texinfo files. I thought it would be very handy to have a list of acronyms available while using Emacs.

It may be considered only a very little contribution. However I make it to express my appreciation for the GNU project and its philosophy. This project does not only make software available free to everyone, it also seems that it sets the only standard `everyone' supports nowadays.

I started V.E.R.A. around easter 1993. The ASCII version is being posted to the German=speaking newsgroups *z-netz.alt.listen* and *de.etc.lists* every three months. It is not being posted to an international newsgroup, because the introduction as well as many internal references are only available in German.

Please note that the original version of V.E.R.A. is not related to the to the Free Software Foundation or the GNU project at all.

The list is far from being complete as dozens of new acronyms evolve every month. Also I am not an every day Unix user, so I am pretty sure that a lot of acronyms from the Unix world are missing. The same is true for the Apple world. Please don't be angry if your field of computing seems little or not represented at all. Instead please consider contributeing the missing acronyms to let other people know what you know.

The shift from the original format of V.E.R.A. to Texinfo format took a whole week. Please forgive any errors which still remain. For hints how to submit corrections or to report bugs, please see section Release notes and other useful information.

For a more in-depth discussion of some of the more common acronyms contained in this document you may want to consider visiting the *Free Online Dictionary of Computing (FOLDOC)*, at least if you have access to the World Wide Web. It can be found at http://wombat.doc.ic.ac.uk/foldoc (as of June 1997).

Go to the previous, next section.

Go to the previous, next section.

TOC

List format

Some notes on the systematic approach of the acronym list:

Because often acronyms are being referenced not very unique in style, the acronyms in V.E.R.A. do not contain any special characters; only characters and numbers are used. Please remember this, when you are searching for acronyms.

German umlauts are expressed with AE, ae, OE, oe, UE, ue and ss.

Examples:

ASN.1 = ASN1
OS/2 = OS2
DFUE, Foerderung, Grossforschungseinrichtung, ...

What is thought to be the common style can be found at the end of the line(s) in quotation marks.

Examples:

ASN1

: Abstract Syntax Notation One (OSI), "ASN.1"

ATT

: American Telephone and Telegraph, "AT&T"

Style of the acronym expansions

The acronym expansions basically follow their acronyms in style. Exceptions are made if expansions are containing acronyms theirselves. This often leads to an kind of odd or even wrong orthogaphical style.

Examples:

AUX

: Apple UniX, "A/UX"

XT

: eXtended Technology

ACL

: Advanced CMOS Logic

Alternative expansions

If alternative expansions for the same acronym are known these are separated by a slash. However that is not true for acronyms of basically different meanings.

Examples:

AE

Apple Events

AE

Application Entity / Environment / Engineering (APE)

Additional explanations

Terms in square brackets show additional explanations, of which do not directly belong to the acronym's expansion, but rather to the acronym's meaning.

Examples:

ARM

Annotated [c++] Reference Manual

RLL

Run Length Limited [encoding]

Reference tags for acronyms

The terms in parenthesis which often appear behind the expansions are reference links to certain topics or firms and should help you to get an idea of in which context an acronym is being used.

Examples:

WB

WorkBench (Amiga)

WYSIWYG

What You See Is What You Get (DTP)

Please note that the original version of V.E.R.A. is meant to be formatted in a hypertext environment. Thus some references may seem redundant in an environment which does not support such links.

Examples:

API

Application Program Interface (API)

Concatenated acronyms

Often-used concatenated acronyms appear without the space or any special characters between them.

Examples:

AMTPE

Apple Media Tool Programming Environment (Apple), "AMT PE"

RISCOS

RISC Operating System (Acorn), "RISC OS"

Acronyms pointing to versions

When there is a series of related acronyms differing by a number at the end, V.E.R.A. usually has just one entry, which omits the number. For example, there are many versions of MNP (Microcom Networking Protocol), distinguished by numbers; but we do not list MNP4, MNP5 or MNP10, only MNP.

So if you look for an acronym that ends in a number, and you don't find it, try leaving out the number(s).

File extensions

Please note that file extensions are not covered by V.E.R.A. at the moment, although some exceptions were made. You will not find things like EPS (Encapsulated PostScript) or GIF (Graphics Interchange Format).

There are many such lists on the Internet if you should be looking for file extensions.

Gaps

You will probably notice that some acronym expansions or references are tagged with three question marks (`???'). This indicates that an expansion or reference is still missing or uncertain.

Go to the previous, next section.

Go to the previous, next section.

TOC

Acronym

An acronym is a word derived from the initial letters or groups of letters of several other words.

Popular examples are:
Laser, from "**l**ight **a**mplification by **s**timulated **e**mission of **r**adition", *Radar*, from "**ra**dio **d**etecting **a**nd **r**anging" or *snafu*, which should politely be rendered as "**s**ituation **n**ormal, **a**ll **f**ouled **u**p".

The word acronym itself derives from the greek words *akros* "the highest, the most outer" and *onyma* "name".

Please note that acronyms are always pronounced as the spelling indicates.

Go to the previous, next section.

Go to the previous, next section.

Release notes and other useful information

Version 4.4

Version 4.4 (Texinfo release 1.3) contains about 250 acronyms more than the initial release. Now V.E.R.A. knows about more than 7400 acronyms.

Version 4.2 had accidently been released containing ISO 8859-1 characters. I received no complaints, but I believe not everyone can display these characters as meant to be, so I switched back to 7 bit with this version.

Recently V.E.R.A. ran into copyright/trademark problems. It's unbelievable how small the name space in the computer field went over the years. To give you an example there are acronyms in this list which have five or more different meanings. However Systems Science Inc. agreed to solve the problem by including the following sentence into the distribution: "This dictionary has nothing to do with Systems Science Inc. or its products." Please don't wonder if you surprisingly run into this sentence.

Version 4.3

I found no time to release V.E.R.A. 4.3 as Texinfo version in March 1998 as I was busy with other things. As compiling the Texinfo version is always additional work to do, I thought of to spare the March version and release the next Texinfo version on the next release date (June, 1st).

Version 4.2

Version 4.2 contains about 180 acronyms more than the last release. I also received a mail from the folks who maintain the GNU catalogue. This may mean that V.E.R.A. will be included on the GNU CD-ROM distribution, but I'm not sure.

Version 4.1

Version 4.1 contains about 500 acronyms more than the initial release. Now V.E.R.A. knows about more than 7000 acronyms.

Besides I wrote a script which automagically translates the german references sometimes to be found behind the acronym expansions in the original version of V.E.R.A. into the appropriate english terms. It also adds `Germany' to german locations. However I still may have missed some german terms or locations. It is kind of awful to proof-read 7000 acronym expansions. If you still spot german terms in the references, please let me know about.

Installation as Info file

If you like to install V.E.R.A. in your Info environment, there are three steps to do so:

1. Make an Info file from the Texinfo source, type:

makeinfo vera.texi

(The file extension may differ in your environment.)

2. Copy the files generated by the previous step to the directory where the Info files reside on your site.
3. Go to your Info directory and edit the `dir` file. Insert a line like the following where appropriate.

   * V.E.R.A.: (vera). Virtual Entity of Relevant Aconyms

Bugs

This edition of V.E.R.A. has been translated from German and has not been intensively proofread. Please report bugs concerning the use of broken or unsound English used in this document to the current maintainer of this document.

If you spot any faults in the acronym chapters, these will be considered a bug too. Please report those bugs to the current maintainer of this document.

Call for submissions

This acronym list is not and will never be complete. However if you feel that something particular is missing, please submit it to the current maintainer. Your submission will be included in the next edition of V.E.R.A.

Current maintainer

The current maintainer of this document is
Oliver Heidelbach <oheiabbd@zedat.fu-berlin.de>

Go to the previous, next section.

Go to the previous, next section.

TOC

Credits

The following people contributed to this document:
(In order to block spam-robots all @ have been replaced with #.)

<peterk#cbmger.de.so.commodore.com>
Wolfgang Houben <castor#newswire.de>
Hans Fischer <profi#romeo.berlinet.de>
Harald Welte <h.welte#silver.nbg.sub.org>
<a.lessmann#link-k.zer>
Rainer Frohnhoff <r.frohnhoff#aworld.zer>
Marcus Cai Degler <m.c.degler#ithh.sh.sub.de>
Sven van der Meer <vdmeer#cs.tu-berlin.de>
<elmar#peggy.e-technik.uni-dortmund.de>
Patrick Hess <poseidon#newswire.de>
Hans-Georg Forster <h.g.forster#parabol.cl.sub.de>
<thahn#berg93.in-berlin.de>
<jochen#wip.mhs.compuserve.com>
Stefan Rinne <s_rinne#trashcan.mcnet.de>
Wolfgang Kopp <kopp#naranek.camelot.de>
<ismias#wi.uni-muenster.de>
<smendoza#yeti.dit.upm.es>
<tms#dame.shnet.org>
Alexander Senne <akki#terra-i.rhein-ruhr.de>
Frank Schlueter <frs#frs.in-berlin.de>
Frank-Th. Bonnemeyer <ftb%mbpdo%mbpdo.uucp#Germany.EU.net>
Olaf Naumann <amandus#esrf.fr>
Keith Edgerley <edgerley#ebu.ch>
Kristian Koehntopp <kris#schulung.netuse.de>
Andreas Kempf <Andreas_Kempf#sidoun.go.germany-online.de>
Thomas Rexroth <100414.42#compuserve.com>
Christoph Lechner <Lechner#edvz.uni-linz.ac.at>
Timothy Slater <t.slater#link-m.de>
J.P. Lodge <jpl1#doc.ic.ac.uk>
Andre Torrez <andre#frontside.com>
Jeff Jewell <jcjewell#concentric.net>
Peter H. Wendt <phw#compunet.de>
Rene Caspersen <rene#sondagsavisen.dk>
Carsten Schymik <cschy#zedat.fu-berlin.de>
Noah Friedman <friedman#splode.com>
Robert Bihlmeyer <robbe#orcus.priv.at>
Matt Hillman <dhillman#netgate.net>

Claude Gingras <CGingras#telebec.qc>
David Frewen <a-davfre#microsoft.com>
Thorsten 'thh' Hempel <thh#valis.netestate.de>
Bruno Haible <haible#ilog.fr>
Andreas Waibel <Andreas.Waibel#studbox.uni-stuttgart.de>
Martin Zierke <Zierke#ccschaper.de>
Leonard <acmeman#myriad.net>
Otto Stolz <Otto.Stolz#uni-konstanz.de>
Wolfgang Borgert
Stefan Hackebeil <stefan#hackebeil.l.uunet.de>
Wolf Ivo Lademann <wil#kiel.netsurf.de>
Werner Henze <beinhart#cs.tu-berlin.de>
Kai Reese <reese#tecs.de>
John L. Sokol <sokol#livecam.com>
Guy Dumais <dumais#rgl.polymtl.ca>
Gerhard Moeller <Gerhard.Moeller#offis.uni-oldenburg.de>
Mike Moxcey <mmoxcey#email.aphis.usda.gov>
Marco Koering <Marco.Koering#swisscom.com>
Olaf Columbus <olaf.columbus#ffm2.sni.de>
Craig Menefee <cmenefee#cwia.com>
Kevin L. Burns <kburns#netscape.com>

About 200 acronyms dealing with the ATM subject were taken
from the online collection of the ATM forum.
By courtesy of ATM forum.

Another batch of acronyms was taken from the freely available
acronym collection "Internet Perls 22" (*iperls22.zip*).
By courtesy of William Hogg <wmhogg#execpc.com>.

A lot of military related acronyms as well as several others
were taken from the online collection of the DISA Center for
Standards.
By courtesy of DISA Center for Standards.

Thanks to all of them and everybody who should be missing.

Also thanks to the following people:

Richard M. Stallman
For his encouraging words to get this edition done and some comments on how a Texinfo file should be formatted. Also for the efforts taken to deal with the Systems Science Inc. copyright/trademark incident in April 1998.

Horst von Brand <vonbrand#sleipnir.valparaiso.cl>
For pointing to the inconsistency between the file name in `@setfilename` (upper case) and the

suggested info entry (lower case).

Go to the previous, next section.

Go to the previous, next section.

TOC

Disclaimer

This is a special GNU edition of V.E.R.A., a list dealing with computational acronyms. It is currently maintained by Oliver Heidelbach <oheiabbd@zedat.fu-berlin.de>.

Although V.E.R.A. has been corrected and worked over for quite a long time, no guarantee can be given for the correctness of any part of its contents. Please don't get it wrong and help to correct any faults you may find.

Please send corrections to the current maintainer.

The author suggests not to redistribute corrected versions of this document, but instead to inform the actual maintainer about corrections you might have. This would help not to confuse users because of different versions.

Within the above restrictions the distribution of this document is explicitly encouraged and I hope you'll find it of some value.

Please note that another version of V.E.R.A. exists, which is not related to the GNU project.

This dictionary has nothing to do with Systems Science Inc. or its products.

Oliver Heidelbach (original author)
<oheiabbd@zedat.fu-berlin.de>, <o.heidelbach@telemail.berlinet.de>

Go to the previous, next section.

Go to the previous section.

Index

0

a

b

c

d

e

f

g

h

o

p

q

r

s

t

u

- U, Acronyms beginning with
- Umlauts, German
- Used format

v

- V, Acronyms beginning with
- V.E.R.A., What is it?
- Versions, Acronyms pointing to

w

- W, Acronyms beginning with
- What is V.E.R.A.?

x

- X, Acronyms beginning with

y

- Y, Acronyms beginning with

z

- Z, Acronyms beginning with

Go to the previous section.

VIP

Go to the next section.

VIP

A Vi Package for GNU Emacs (Version 3.5, September 15, 1987)

Masahiko Sato

Distribution

Go to the next section.

Go to the previous, next section.

Introduction

VIP is a Vi emulating package written in Emacs Lisp. VIP implements most Vi commands including Ex commands. It is therefore hoped that this package will enable you to do Vi style editing under the powerful GNU Emacs environment. This manual describes the usage of VIP assuming that you are fairly accustomed to Vi but not so much with Emacs. Also we will concentrate mainly on differences from Vi, especially features unique to VIP.

It is recommended that you read chapters on survey and on customization before you start using VIP. Other chapters may be used as future references.

Comments and bug reports are welcome. Please send messages to `ms@Sail.Stanford.Edu` if you are outside of Japan and to `masahiko@unsun.riec.tohoku.junet` if you are in Japan.

Go to the previous, next section.

Go to the previous, next section.

A Survey of VIP

In this chapter we describe basics of VIP with emphasis on the features not found in Vi and on how to use VIP under GNU Emacs.

Basic Concepts

We begin by explaining some basic concepts of Emacs. These concepts are explained in more detail in the GNU Emacs Manual.

Conceptually, a buffer is just a string of ASCII characters and two special characters PNT (point) and MRK (mark) such that the character PNT occurs exactly once and MRK occurs at most once. The text of a buffer is obtained by deleting the occurrences of PNT and MRK. If, in a buffer, there is a character following PNT then we say that point is looking at the character; otherwise we say that point is at the end of buffer. PNT and MRK are used to indicate positions in a buffer and they are not part of the text of the buffer. If a buffer contains a MRK then the text between MRK and PNT is called the region of the buffer.

Emacs provides (multiple) windows on the screen, and you can see the content of a buffer through the window associated with the buffer. The cursor of the screen is always positioned on the character after PNT.

A keymap is a table that records the bindings between characters and command functions. There is the global keymap common to all the buffers. Each buffer has its local keymap that determines the mode of the buffer. Local keymap overrides global keymap, so that if a function is bound to some key in the local keymap then that function will be executed when you type the key. If no function is bound to a key in the local map, however, the function bound to the key in the global map becomes in effect.

Loading VIP

The recommended way to load VIP automatically is to include the line:

```
(load "vip")
```

in your `.emacs' file. The `.emacs' file is placed in your home directory and it will be executed every time you invoke Emacs. If you wish to be in vi mode whenever Emacs starts up, you can include the following line in your `.emacs' file instead of the above line:

```
(setq term-setup-hook 'vip-mode)
```

(See section Vi Mode, for the explanation of vi mode.)

Even if your `.emacs' file does not contain any of the above lines, you can load VIP and enter vi mode by typing the following from within Emacs.

M-x vip-mode

Modes in VIP

Loading VIP has the effect of globally binding C-z (Control-z) to the function `vip-change-mode-to-vi`. The default binding of C-z in GNU Emacs is `suspend-emacs`, but, you can also call `suspend-emacs` by typing C-x C-z. Other than this, all the key bindings of Emacs remain the same after loading VIP.

Now, if you hit C-z, the function `vip-change-mode-to-vi` will be called and you will be in vi mode. (Some major modes may locally bind C-z to some special functions. In such cases, you can call `vip-change-mode-to-vi` by `execute-extended-command` which is invoked by M-x. Here M-x means Meta-x, and if your terminal does not have a META key you can enter it by typing ESC x. The same effect can also be achieve by typing M-x vip-mode.)

You can observe the change of mode by looking at the mode line. For instance, if the mode line is:

```
-----Emacs: *scratch*                (Lisp Interaction)----All-----------
```

then it will change to:

```
-----Vi:    *scratch*                (Lisp Interaction)----All-----------
```

Thus the word `Emacs' in the mode line will change to `Vi'.

You can go back to the original emacs mode by typing C-z in vi mode. Thus C-z toggles between these two modes.

Note that modes in VIP exist orthogonally to modes in Emacs. This means that you can be in vi mode and at the same time, say, shell mode.

Vi mode corresponds to Vi's command mode. From vi mode you can enter insert mode (which corresponds to Vi's insert mode) by usual Vi command keys like i, a, o ... etc.

In insert mode, the mode line will look like this:

```
-----Insert *scratch*                (Lisp Interaction)----All-----------
```

You can exit from insert mode by hitting ESC key as you do in Vi.

That VIP has three modes may seem very complicated, but in fact it is not so. VIP is implemented so that you can do most editing remaining only in the two modes for Vi (that is vi mode and insert mode).

Emacs Mode

You will be in this mode just after you loaded VIP. You can do all normal Emacs editing in this mode. Note that the key C-z is globally bound to `vip-change-mode-to-vi`. So, if you type C-z in this mode then you will be in vi mode.

Vi Mode

This mode corresponds to Vi's command mode. Most Vi commands work as they do in Vi. You can go back to emacs mode by typing C-z. You can enter insert mode, just as in Vi, by typing i, a etc.

Insert Mode

The key bindings in this mode is the same as in the emacs mode except for the following 4 keys. So, you can move around in the buffer and change its content while you are in insert mode.

ESC

This key will take you back to vi mode.

C-h

Delete previous character.

C-w

Delete previous word.

C-z

Typing this key has the same effect as typing ESC in emacs mode. Thus typing C-z x in insert mode will have the same effect as typing ESC x in emacs mode.

Differences from Vi

The major differences from Vi are explained below.

Undoing

You can repeat undoing by the . key. So, u will undo a single change, while u . . ., for instance, will undo 4 previous changes. Undo is undoable as in Vi. So the content of the buffer will be the same before and after u u.

Changing

Some commands which change a small number of characters are executed slightly differently. Thus, if point is at the beginning of a word `foo' and you wished to change it to `bar' by typing c w, then VIP will prompt you for a new word in the minibuffer by the prompt `foo => '. You can then enter `bar' followed by RET or ESC to complete the command. Before you enter RET or ESC you can abort the command by typing C-g. In general, you can abort a partially formed command by typing C-g.

Searching

As in Vi, searching is done by / and ?. The string will be searched literally by default. To invoke a regular expression search, first execute the search command / (or ?) with empty search string. (I.e, type / followed by RET.) A search for empty string will toggle the search mode between vanilla search and regular expression search. You cannot give an offset to the search string. (It is a limitation.) By default, search will

wrap around the buffer as in Vi. You can change this by rebinding the variable `vip-search-wrap-around`. See section Customization, for how to do this.

z Command

For those of you who cannot remember which of z followed by RET, . and - do what. You can also use z followed by H, M and L to place the current line in the Home (Middle, and Last) line of the window.

Counts

Some Vi commands which do not accept a count now accept one

p
P

Given counts, text will be yanked (in Vi's sense) that many times. Thus 3 p is the same as p p p.

o
O

Given counts, that many copies of text will be inserted. Thus o a b c ESC will insert 3 lines of `abc' below the current line.

/
?

Given a count n, n-th occurrence will be searched.

Marking

Typing an m followed by a lower case character ch marks the point to the register named ch as in Vi. In addition to these, we have following key bindings for marking.

m <

Set mark at the beginning of buffer.

m >

Set mark at the end of buffer.

m .

Set mark at point (and push old mark on mark ring).

m ,

Jump to mark (and pop mark off the mark ring).

Region Commands

Vi operators like d, c etc. are usually used in combination with motion commands. It is now possible to use current region as the argument to these operators. (A region is a part of buffer delimited by point and mark.) The key r is used for this purpose. Thus d r will delete the current region. If R is used instead of r the region will first be enlarged so that it will become the smallest region containing the original region and consisting of whole lines. Thus m . d R will have the same effect as d d.

Some New Commands

Note that the keys below (except for R) are not used in Vi.

C-a

Move point to the beginning of line.

C-n

If you have two or more windows in the screen, this key will move point to the next window.

C-o

Insert a newline and leave point before it, and then enter insert mode.

C-r

Backward incremental search.

C-s

Forward incremental search.

C-c

C-x

ESC

These keys will exit from vi mode and return to emacs mode temporarily. If you hit one of these keys, Emacs will be in emacs mode and will believe that you hit that key in emacs mode. For example, if you hit C-x followed by 2, then the current window will be split into 2 and you will be in vi mode again.

\

Escape to emacs mode. Hitting \ will take you to emacs mode, and you can execute a single Emacs command. After executing the Emacs command you will be in vi mode again. You can give a count before typing \. Thus 5 \ *, as well as \ C-u 5 *, will insert `*****' before point. Similarly 1 0 \ C-p will move the point 10 lines above the current line.

K

Kill current buffer if it is not modified. Useful when you selected a buffer which you did not want.

Q

R

Q is for query replace and R is for replace. By default, string to be replaced are treated literally. If you wish to do a regular expression replace, first do replace with empty string as the string to be replaced. In this way, you can toggle between vanilla and regular expression replacement.

v

V

These keys are used to Visit files. v will switch to a buffer visiting file whose name can be entered in the minibuffer. V is similar, but will use window different from the current window.

#

If followed by a certain character ch, it becomes an operator whose argument is the region determined by the motion command that follows. Currently, ch can be one of c, C, g, q and s.

c

Change upper case characters in the region to lower case (`downcase-region`).

C

Change lower case characters in the region to upper case. For instance, # C 3 w will capitalize 3 words from the current point (`upcase-region`).

g

Execute last keyboard macro for each line in the region (`vip-global-execute`).

q

Insert specified string at the beginning of each line in the region (`vip-quote-region`).

s

Check spelling of words in the region (`spell-region`).

*

Call last keyboard macro.

New Key Bindings

In VIP the meanings of some keys are entirely different from Vi. These key bindings are done deliberately in the hope that editing under Emacs will become easier. It is however possible to rebind these keys to functions which behave similarly as in Vi. See section Customizing Key Bindings, for details.

C-g

g

In Vi, C-g is used to get information about the file associated to the current buffer. Here, g will do that, and C-g is used to abort a command (this is for compatibility with emacs mode.)

SPC

RET

Now these keys will scroll up and down the text of current window. Convenient for viewing the text.

s

S

They are used to switch to a specified buffer. Useful for switching to already existing buffer since buffer name completion is provided. Also a default buffer will be given as part of the prompt, to which you can switch by just typing RET key. s is used to select buffer in the current window, while S selects buffer in another window.

C

X

These keys will exit from vi mode and return to emacs mode temporarily. If you type C (X), Emacs will be in emacs mode and will believe that you have typed C-c (C-x, resp.) in emacs mode. Moreover, if the following character you type is an upper case letter, then Emacs will believe that you have typed the corresponding control character. You will be in vi mode again after the command is executed. For example, typing X S in vi mode is the same as typing C-x C-s in emacs mode. You get the same effect by typing C-x C-s in vi mode, but the idea here is that you can execute useful

Emacs commands without typing control characters. For example, if you hit X (or C-x) followed 2, then the current window will be split into 2 and you will be in vi mode again.

In addition to these, `ctl-x-map` is slightly modified:

X 3

C-x 3

This is equivalent to C-x 1 C-x 2 (1 + 2 = 3).

Window Commands

In this and following subsections, we give a summary of key bindings for basic functions related to windows, buffers and files.

C-n

Switch to next window.

X 1

C-x 1

Delete other windows.

X 2

C-x 2

Split current window into two windows.

X 3

C-x 3

Show current buffer in two windows.

Buffer Commands

s

Switch to the specified buffer in the current window (`vip-switch-to-buffer`).

S

Switch to the specified buffer in another window (`vip-switch-to-buffer-other-window`).

K

Kill the current buffer if it is not modified.

X S

C-x C-s

Save the current buffer in the file associated to the buffer.

File Commands

v

Visit specified file in the current window.

V

Visit specified file in another window.

X W

C-x C-w

Write current buffer into the specified file.

X I

C-x C-i

Insert specified file at point.

Miscellaneous Commands

X (

C-x (

Start remembering keyboard macro.

X)

C-x)

Finish remembering keyboard macro.

*

Call last remembered keyboard macro.

X Z

C-x C-z

Suspend Emacs.

Z Z

Exit Emacs.

Q

Query replace.

R

Replace.

Go to the previous, next section.

Go to the previous, next section.

Vi Commands

This chapter describes Vi commands other than Ex commands implemented in VIP. Except for the last section which discusses insert mode, all the commands described in this chapter are to be used in vi mode.

Numeric Arguments

Most Vi commands accept a numeric argument which can be supplied as a prefix to the commands. A numeric argument is also called a count. In many cases, if a count is given, the command is executed that many times. For instance, 5 d d deletes 5 lines while simple d d deletes a line. In this manual the metavariable n will denote a count.

Important Keys

The keys C-g and C-l are unique in that their associated functions are the same in any of emacs, vi and insert mode.

C-g

Quit. Cancel running or partially typed command (`keyboard-quit`).

C-l

Clear the screen and reprint everything (`recenter`).

In Emacs many commands are bound to the key strokes that start with C-x, C-c and ESC. These commands can be accessed from vi mode as easily as from emacs mode.

C-x

C-c

ESC

Typing one of these keys have the same effect as typing it in emacs mode. Appropriate command will be executed according as the keys you type after it. You will be in vi mode again after the execution of the command. For instance, if you type ESC < (in vi mode) then the cursor will move to the beginning of the buffer and you will still be in vi mode.

C

X

Typing one of these keys have the effect of typing the corresponding control character in emacs mode. Moreover, if you type an upper case character following it, that character will also be translated to the corresponding control character. Thus typing X W in vi mode is the same as typing C-x C-w in emacs mode. You will be in vi mode again after the execution of a command.

\

Escape to emacs mode. Hitting the \ key will take you to emacs mode, and you can execute a single Emacs command. After executing the Emacs command you will be in vi mode again. You can give a count before typing \. Thus 5 \ +, as well as \ C-u 5 +, will insert `+++++' before point.

Buffers and Windows

In Emacs the text you edit is stored in a buffer. See GNU Emacs Manual, for details. There is always one selected buffer which is called the current buffer.

You can see the contents of buffers through windows created by Emacs. When you have multiple windows on the screen only one of them is selected. Each buffer has a unique name, and each window has a mode line which shows the name of the buffer associated with the window and other information about the status of the buffer. You can change the format of the mode line, but normally if you see `**' at the beginning of a mode line it means that the buffer is modified. If you write out the content of the buffer to a file, then the buffer will become not modified. Also if you see `%%' at the beginning of the mode line, it means that the file associated with the buffer is write protected.

We have the following commands related to windows and buffers.

C-n

Move cursor to the next-window (`vip-next-window`).

X 1

Delete other windows and make the selected window fill the screen (`delete-other-windows`).

X 2

Split current window into two windows (`split-window-vertically`).

X 3

Show current buffer in two windows.

s buffer RET

Select or create a buffer named buffer (`vip-switch-to-buffer`).

S buffer RET

Similar but select a buffer named buffer in another window (`vip-switch-to-buffer-other-window`).

K

Kill the current buffer if it is not modified or if it is not associated with a file (`vip-kill-buffer`).

X B

List the existing buffers (`list-buffers`).

As buffer name completion is provided, you have only to type in initial substring of the buffer name which is sufficient to identify it among names of existing buffers. After that, if you hit TAB the rest of the buffer name will be supplied by the system, and you can confirm it by RET. The default buffer name to switch to will also be prompted, and you can select it by giving a simple RET. See GNU Emacs

Manual for details of completion.

Files

We have the following commands related to files. They are used to visit, save and insert files.

v file RET

Visit specified file in the current window (`vip-find-file`).

V file RET

Visit specified file in another window (`vip-find-file-other-window`).

X S

Save current buffer to the file associated with the buffer. If no file is associated with the buffer, the name of the file to write out the content of the buffer will be asked in the minibuffer.

X W file RET

Write current buffer into a specified file.

X I file RET

Insert a specified file at point.

g

Give information on the file associated with the current buffer. Tell you the name of the file associated with the buffer, the line number of the current point and total line numbers in the buffer. If no file is associated with the buffer, this fact will be indicated by the null file name `"".

In Emacs, you can edit a file by visiting it. If you wish to visit a file in the current window, you can just type v. Emacs maintains the default directory which is specific to each buffer. Suppose, for instance, that the default directory of the current buffer is `` `/usr/masahiko/lisp/' ``. Then you will get the following prompt in the minibuffer.

```
visit file: /usr/masahiko/lisp/
```

If you wish to visit, say, `` `vip.el' `` in this directory, then you can just type `vip.el' followed by RET. If the file `` `vip.el' `` already exists in the directory, Emacs will visit that file, and if not, the file will be created. Emacs will use the file name (`` `vip.el' ``, in this case) as the name of the buffer visiting the file. In order to make the buffer name unique, Emacs may append `<2>', `<3>' etc., to the buffer name. As the file name completion is provided here, you can sometime save typing. For instance, suppose there is only one file in the default directory whose name starts with `v', that is `vip.el'. Then if you just type v TAB then it will be completed to `vip.el'. Thus, in this case, you just have to type v v TAB RET to visit `` `/usr/masahiko/lisp/vip.el' ``. Continuing the example, let us now suppose that you wished to visit the file `` `/usr/masahiko/man/vip.texinfo' ``. Then to the same prompt which you get after you typed v, you can enter `/usr/masahiko/man/vip.texinfo' or `../man/vip.texinfo' followed by RET.

Use V instead of v, if you wish to visit a file in another window.

You can verify which file you are editing by typing g. (You can also type X B to get nformation on other buffers too.) If you type g you will get an information like below in the echo area:


```
"/usr/masahiko/man/vip.texinfo" line 921 of 1949
```

After you edited the buffer (`vip.texinfo', in our example) for a while, you may wish to save it in a file. If you wish to save it in the file associated with the buffer (`/usr/masahiko/man/vip.texinfo', in this case), you can just say X S. If you wish to save it in another file, you can type X W. You will then get a similar prompt as you get for v, to which you can enter the file name.

Viewing the Buffer

In this and next section we discuss commands for moving around in the buffer. These command do not change the content of the buffer. The following commands are useful for viewing the content of the current buffer.

SPC

C-f

Scroll text of current window upward almost full screen. You can go *forward* in the buffer by this command (`vip-scroll`).

RET

C-b

Scroll text of current window downward almost full screen. You can go *backward* in the buffer by this command (`vip-scroll-back`).

C-d

Scroll text of current window upward half screen. You can go *down* in the buffer by this command (`vip-scroll-down`).

C-u

Scroll text of current window downward half screen. You can go *up* in the buffer by this command (`vip-scroll-up`).

C-y

Scroll text of current window upward by one line (`vip-scroll-down-one`).

C-e

Scroll text of current window downward by one line (`vip-scroll-up-one`).

You can repeat these commands by giving a count. Thus, 2 SPC has the same effect as SPC SPC.

The following commands reposition point in the window.

z H

z RET

Put point on the top (*home*) line in the window. So the current line becomes the top line in the window. Given a count n, point will be placed in the n-th line from top (`vip-line-to-top`).

z M

z .

Put point on the *middle* line in the window. Given a count n, point will be placed in the n-th lir from the middle line (`vip-line-to-middle`).

z L

z -

Put point on the *bottom* line in the window. Given a count n, point will be placed in the n-th line from bottom (`vip-line-to-bottom`).

C-l

Center point in window and redisplay screen (`recenter`).

Mark Commands

The following commands are used to mark positions in the buffer.

m ch

Store current point in the register ch. ch must be a lower case character between a and z.

m <

Set mark at the beginning of current buffer.

m >

Set mark at the end of current buffer.

m .

Set mark at point.

m ,

Jump to mark (and pop mark off the mark ring).

Emacs uses the mark ring to store marked positions. The commands m <, m > and m . not only set mark but also add it as the latest element of the mark ring (replacing the oldest one). By repeating the command `m ,' you can visit older and older marked positions. You will eventually be in a loop as the mark ring is a ring.

Motion Commands

Commands for moving around in the current buffer are collected here. These commands are used as an `argument' for the delete, change and yank commands to be described in the next section.

h

Move point backward by one character. Signal error if point is at the beginning of buffer, but (unlike Vi) do not complain otherwise (`vip-backward-char`).

l

Move point backward by one character. Signal error if point is at the end of buffer, but (unlike Vi) do not complain otherwise (`vip-forward-char`).

j

Move point to the next line keeping the current column. If point is on the last line of the buffer, a new line will be created and point will move to that line (`vip-next-line`).

k

Move point to the previous line keeping the current column (`vip-next-line`).

+

Move point to the next line at the first non-white character. If point is on the last line of the buffer, a new line will be created and point will move to the beginning of that line (`vip-next-line-at-bol`).

-

Move point to the previous line at the first non-white character (`vip-previous-line-at-bol`).

If a count is given to these commands, the commands will be repeated that many times.

0

Move point to the beginning of line (`vip-beginning-of-line`).

^

Move point to the first non-white character on the line (`vip-bol-and-skip-white`).

$

Move point to the end of line (`vip-goto-eol`).

n |

Move point to the n-th column on the line (`vip-goto-col`).

Except for the | command, these commands neglect a count.

w

Move point forward to the beginning of the next word (`vip-forward-word`).

W

Move point forward to the beginning of the next word, where a word is considered as a sequence of non-white characters (`vip-forward-Word`).

b

Move point backward to the beginning of a word (`vip-backward-word`).

B

Move point backward to the beginning of a word, where a *word* is considered as a sequence of non-white characters (`vip-forward-Word`).

e

Move point forward to the end of a word (`vip-end-of-word`).

E

Move point forward to the end of a word, where a *word* is considered as a sequence of non-white characters (`vip-end-of-Word`).

Here the meaning of the word `word' for the w, b and e commands is determined by the syntax table effective in the current buffer. Each major mode has its syntax mode, and therefore the meaning of a

word also changes as the major mode changes. See GNU Emacs Manual for details of syntax table.

H

Move point to the beginning of the *home* (top) line of the window. Given a count n, go to the n-th line from top (`vip-window-top`).

M

Move point to the beginning of the *middle* line of the window. Given a count n, go to the n-th line from the middle line (`vip-window-middle`).

L

Move point to the beginning of the *lowest* (bottom) line of the window. Given count, go to the n-th line from bottom (`vip-window-bottom`).

These commands can be used to go to the desired line visible on the screen.

(

Move point backward to the beginning of the sentence (`vip-backward-sentence`).

)

Move point forward to the end of the sentence (`vip-forward-sentence`).

{

Move point backward to the beginning of the paragraph (`vip-backward-paragraph`).

}

Move point forward to the end of the paragraph (`vip-forward-paragraph`).

A count repeats the effect for these commands.

G

Given a count n, move point to the n-th line in the buffer on the first non-white character. Without a count, go to the end of the buffer (`vip-goto-line`).

` `

Exchange point and mark (`vip-goto-mark`).

` ch

Move point to the position stored in the register ch. ch must be a lower case letter.

' '

Exchange point and mark, and then move point to the first non-white character on the line (`vip-goto-mark-and-skip-white`).

' ch

Move point to the position stored in the register ch and skip to the first non-white character on the line. ch must be a lower case letter.

%

Move point to the matching parenthesis if point is looking at (,), {, }, [or] (`vip-paren-match`).

The command G mark point before move, so that you can return to the original point by ` `. The original point will also be stored in the mark ring.

The following commands are useful for moving points on the line. A count will repeat the effect.

f ch

Move point forward to the character ch on the line. Signal error if ch could not be found (`vip-find-char-forward`).

F ch

Move point backward to the character ch on the line. Signal error if ch could not be found (`vip-find-char-backward`).

t ch

Move point forward upto the character ch on the line. Signal error if ch could not be found (`vip-goto-char-forward`).

T ch

Move point backward upto the character ch on the line. Signal error if ch could not be found (`vip-goto-char-backward`).

;

Repeat previous f, t, F or T command (`vip-repeat-find`).

,

Repeat previous f, t, F or T command, in the opposite direction (`vip-repeat-find-opposite`).

Searching and Replacing

Following commands are available for searching and replacing.

/ string RET

Search the first occurrence of the string string forward starting from point. Given a count n, the n-th occurrence of string will be searched. If the variable `vip-re-search` has value `t` then regular expression search is done and the string matching the regular expression string is found. If you give an empty string as string then the search mode will change from vanilla search to regular expression search and vice versa (`vip-search-forward`).

? string RET

Same as /, except that search is done backward (`vip-search-backward`).

n

Search the previous search pattern in the same direction as before (`vip-search-next`).

N

Search the previous search pattern in the opposite direction (`vip-search-Next`).

C-s

Search forward incrementally. See GNU Emacs Manual for details (`isearch-forward`).

C-r

Search backward incrementally (`isearch-backward`).

R string RET newstring

There are two modes of replacement, vanilla and regular expression. If the mode is *vanilla* you will get a prompt `Replace string:', and if the mode is *regular expression* you will ge a prompt `Replace regexp:'. The mode is initially *vanilla*, but you can toggle these modes by giving a null string as string. If the mode is vanilla, this command replaces every occurrence of string with newstring. If the mode is regular expression, string is treated as a regular expression and every string matching the regular expression is replaced with newstring (`vip-replace-string`).

Q string RET newstring

Same as R except that you will be asked form confirmation before each replacement (`vip-query-replace`).

r ch

Replace the character point is looking at by the character ch. Give count, replace that many characters by ch (`vip-replace-char`).

The commands / and ? mark point before move, so that you can return to the original point by ``.

Modifying Commands

In this section, commands for modifying the content of a buffer are described. These commands affect the region determined by a motion command which is given to the commands as their argument.

We classify motion commands into point commands and line commands. The point commands are as follows:

```
h, l, 0, ^, $, w, W, b, B, e, E, (, ), /, ?, `, f, F, t, T, %, ;, ,
```

The line commands are as follows:

```
j, k, +, -, H, M, L, {, }, G, '
```

If a point command is given as an argument to a modifying command, the region determined by the point command will be affected by the modifying command. On the other hand, if a line command is given as an argument to a modifying command, the region determined by the line command will be enlarged so that it will become the smallest region properly containing the region and consisting of whole lines (we call this process expanding the region), and then the enlarged region will be affected by the modifying command.

Delete Commands

d motion-command

Delete the region determined by the motion command motion-command.

For example, d $ will delete the region between point and end of current line since $ is a point command that moves point to end of line. d G will delete the region between the beginning of current line and end of the buffer, since G is a line command. A count given to the command above will become the count for the associated motion command. Thus, 3 d w will delete three words.

It is also possible to save the deleted text into a register you specify. For example, you can say " t 3 d to delete three words and save it to register t. The name of a register is a lower case letter between a and z. If you give an upper case letter as an argument to a delete command, then the deleted text will be appended to the content of the register having the corresponding lower case letter as its name. So, " T d w will delete a word and append it to register t. Other modifying commands also accept a register name as their argument, and we will not repeat similar explanations.

We have more delete commands as below.

d d

Delete a line. Given a count n, delete n lines.

d r

Delete current region.

d R

Expand current region and delete it.

D

Delete to the end of a line (`vip-kill-line`).

x

Delete a character after point. Given n, delete n characters (`vip-delete-char`).

DEL

Delete a character before point. Given n, delete n characters (`vip-delete-backward-char`).

Yank Commands

Yank commands yank a text of buffer into a (usually anonymous) register. Here the word `yank' is used in Vi's sense. Thus yank commands do not alter the content of the buffer, and useful only in combination with commands that put back the yanked text into the buffer.

y motion-command

Yank the region determined by the motion command motion-command.

For example, y $ will yank the text between point and the end of line into an anonymous register, while "c y $ will yank the same text into register c.

Use the following command to yank consecutive lines of text.

y y

Y

Yank a line. Given n, yank n lines (`vip-yank-line`).

y r

Yank current region.

y R

Expand current region and yank it.

Put Back Commands

Deleted or yanked texts can be put back into the buffer by the command below.

p

Insert, after the character point is looking at, most recently deleted/yanked text from anonymous register. Given a register name argument, the content of the named register will be put back. Given a count, the command will be repeated that many times. This command also checks if the text to put back ends with a new line character, and if so the text will be put below the current line (`vip-put-back`).

P

Insert at point most recently deleted/yanked text from anonymous register. Given a register name argument, the content of the named register will be put back. Given a count, the command will be repeated that many times. This command also checks if the text to put back ends with a new line character, and if so the text will be put above the current line rather than at point (`vip-Put-back`).

Thus, " c p will put back the content of the register c into the buffer. It is also possible to specify number register which is a numeral between 1 and 9. If the number register n is specified, n-th previously deleted/yanked text will be put back. It is an error to specify a number register for the delete/yank commands.

Change Commands

Most commonly used change command takes the following form.

c motion-command

Replace the content of the region determined by the motion command motion-command by the text you type. If the motion command is a point command then you will type the text into minibuffer, and if the motion command is a line command then the region will be deleted first and you can insert the text in insert mode.

For example, if point is at the beginning of a word `foo' and you wish to change it to `bar', you can type c w. Then, as w is a point command, you will get the prompt `foo =>' in the minibuffer, for which you can type b a r RET to complete the change command.

c c

Change a line. Given a count, that many lines are changed.

c r

Change current region.

c R

Expand current region and change it.

Repeating and Undoing Modifications

VIP records the previous modifying command, so that it is easy to repeat it. It is also very easy to undo changes made by modifying commands.

u

Undo the last change. You can undo more by repeating undo by the repeat command `.'. For example, you can undo 5 previous changes by typing `u....'. If you type `uu', then the second `u' undoes the first undo command (`vip-undo`).

.

Repeat the last modifying command. Given count n it becomes the new count for the repeated command. Otherwise, the count for the last modifying command is used again (`vip-repeat`).

Other Vi Commands

Miscellaneous Vi commands are collected here.

Z Z

Exit Emacs. If modified buffers exist, you will be asked whether you wish to save them or not (`save-buffers-kill-emacs`).

! motion-command format-command

n ! ! format-command

The region determined by the motion command motion-command will be given to the shell command format-command and the region will be replaced by its output. If a count is given, it will be passed to motion-command. For example, `3!Gsort' will sort the region between point and the 3rd line. If ! is used instead of motion-command then n lines will be processed by format-command (`vip-command-argument`).

J

Join two lines. Given count, join that many lines. A space will be inserted at each junction (`vip-join-lines`).

< motion-command

n < <

Shift region determined by the motion command motion-command to left by shift-width (default is 8). If < is used instead of motion-command then shift n lines (`vip-command-argument`).

> motion-command

n > >

Shift region determined by the motion command motion-command to right by shift-width (default is 8). If < is used instead of motion-command then shift n lines (`vip-command-argument`).

= motion-command

Indent region determined by the motion command motion-command. If = is used instead of

motion-command then indent n lines (`vip-command-argument`).

*

Call last remembered keyboard macro.

#

A new vi operator. See section Some New Commands, for more details.

The following keys are reserved for future extensions, and currently assigned to a function that just beeps (`vip-nil`).

`&, @, U, [, ], _, q, ~`

VIP uses a special local keymap to interpret key strokes you enter in vi mode. The following keys are bound to nil in the keymap. Therefore, these keys are interpreted by the global keymap of Emacs. We give below a short description of the functions bound to these keys in the global keymap. See GNU Emacs Manual for details.

C-@

Set mark and push previous mark on mark ring (`set-mark-command`).

TAB

Indent line for current major mode (`indent-for-tab-command`).

LFD

Insert a newline, then indent according to mode (`newline-and-indent`).

C-k

Kill the rest of the current line; before a newline, kill the newline. With a numeric argument, kill that many lines from point. Negative arguments kill lines backward (`kill-line`).

C-l

Clear the screen and reprint everything (`recenter`).

n C-p

Move cursor vertically up n lines (`previous-line`).

C-q

Read next input character and insert it. Useful for inserting control characters (`quoted-insert`).

C-r

Search backward incrementally (`isearch-backward`).

C-s

Search forward incrementally (`isearch-forward`).

n C-t

Interchange characters around point, moving forward one character. With count n, take character before point and drag it forward past n other characters. If no argument and at end of line, the previous two characters are exchanged (`transpose-chars`).

n C-v

Scroll text upward n lines. If n is not given, scroll near full screen (`scroll-up`).

C-w

Kill between point and mark. The text is save in the kill ring. The command P or p can retrieve it from kill ring (`kill-region`).

Insert Mode

You can enter insert mode by one of the following commands. In addition to these, you will enter insert mode if you give a change command with a line command as the motion command. Insert commands are also modifying commands and you can repeat them by the repeat command . (`vip-repeat`).

i

Enter insert mode at point (`vip-insert`).

I

Enter insert mode at the first non white character on the line (`vip-Insert`).

a

Move point forward by one character and then enter insert mode (`vip-append`).

A

Enter insert mode at end of line (`vip-Append`).

o

Open a new line below the current line and enter insert mode (`vip-open-line`).

O

Open a new line above the current line and enter insert mode (`vip-Open-line`).

C-o

Insert a newline and leave point before it, and then enter insert mode (`vip-open-line-at-point`).

Insert mode is almost like emacs mode. Only the following 4 keys behave differently from emacs mode.

ESC

This key will take you back to vi mode (`vip-change-mode-to-vi`).

C-h

Delete previous character (`delete-backward-char`).

C-w

Delete previous word (`vip-delete-backward-word`).

C-z

This key simulates ESC key in emacs mode. For instance, typing C-z x in insert mode iw the same as typing ESC x in emacs mode (`vip-ESC`).

You can also bind C-h to `help-command` if you like. (See section Customizing Key Bindings, for details.) Binding C-h to `help-command` has the effect of making the meaning of C-h uniform among

emacs, vi and insert modes.

When you enter insert mode, VIP records point as the start point of insertion, and when you leave insert mode the region between point and start point is saved for later use by repeat command etc. Therefore, repeat command will not really repeat insertion if you move point by emacs commands while in insert mode.

Go to the previous, next section.

Go to the previous, next section.

Ex Commands

In vi mode, you can execute an Ex command ex-command by typing:

```
: ex-command RET
```

Every Ex command follows the following pattern:

```
address command ! parameters count flags
```

where all parts are optional. For the syntax of address, the reader is referred to the reference manual of Ex.

In the current version of VIP, searching by Ex commands is always magic. That is, search patterns are always treated as regular expressions. For example, a typical forward search would be invoked by :/pat/. If you wish to include `/' as part of pat you must preceded it by `\'. VIP strips off these \'s before / and the resulting pat becomes the actual search pattern. Emacs provides a different and richer class or regular expressions than Vi/Ex, and VIP uses Emacs' regular expressions. See GNU Emacs Manual for details of regular expressions.

Several Ex commands can be entered in a line by separating them by a pipe character `|'.

Ex Command Reference

In this section we briefly explain all the Ex commands supported by VIP. Most Ex commands expect address as their argument, and they use default addresses if they are not explicitly given. In the following, such default addresses will be shown in parentheses.

Most command names can and preferably be given in abbreviated forms. In the following, optional parts of command names will be enclosed in brackets. For example, `co[py]' will mean that copy command can be give as `co' or `cop' or `copy'.

If command is empty, point will move to the beginning of the line specified by the address. If address is also empty, point will move to the beginning of the current line.

Some commands accept flags which are one of p, l and #. If flags are given, the text affected by the commands will be displayed on a temporary window, and you will be asked to hit return to continue. In this way, you can see the text affected by the commands before the commands will be executed. If you hit C-g instead of RET then the commands will be aborted. Note that the meaning of flags is different in VIP from that in Vi/Ex.

(.,.) co[py] addr flags

(.,.) t addr flags

Place a copy of specified lines after addr. If addr is 0, it will be placed before the first line.

(.,.) d[elete] register count flags

Delete specified lines. Text will be saved in a named register if a lower case letter is given, and appended to a register if a capital letter is given.

e[dit] ! +addr file

e[x] ! +addr file

vi[sual] ! +addr file

Edit a new file file in the current window. The command will abort if current buffer is modified, which you can override by giving !. If +addr is given, addr becomes the current line.

file

Give information about the current file.

(1,$) g[lobal] ! /pat/ cmds

(1,$) v /pat/ cmds

Among specified lines first mark each line which matches the regular expression pat, and then execute cmds on each marked line. If ! is given, cmds will be executed on each line not matching pat. v is same as g!.

(.,.+1) j[oin] ! count flags

Join specified lines into a line. Without !, a space character will be inserted at each junction.

(.) k ch

(.) mar[k] ch

Mark specified line by a lower case character ch. Then the addressing form 'ch will refer to this line. No white space is required between k and ch. A white space is necessary between mark and ch, however.

map ch rhs

Define a macro for vi mode. After this command, the character ch will be expanded to rhs in vi mode.

(.,.) m[ove] addr

Move specified lines after addr.

(.) pu[t] register

Put back previously deleted or yanked text. If register is given, the text saved in the register will be put back; otherwise, last deleted or yanked text will be put back.

q[uit] !

Quit from Emacs. If modified buffers with associated files exist, you will be asked whether you wish to save each of them. At this point, you may choose not to quit, by hitting C-g. If ! is given, exit from Emacs without saving modified buffers.

(.) r[ead] file

Read in the content of the file file after the specified line.

(.) r[ead] ! command

Read in the output of the shell command command after the specified line.

se[t]

Set a variable's value. See section Customizing Constants, for the list of variables you can set.

sh[ell]

Run a subshell in a window.

(.,.) s[ubstitute] /pat/repl/ options count flags

(.,.) & options count flags

On each specified line, the first occurrence of string matching regular expression pat is replaced by replacement pattern repl. Option characters are g and c. If global option character g appears as part of options, all occurrences are substituted. If confirm option character c appears, you will be asked to give confirmation before each substitution. If /pat/repl/ is missing, the last substitution is repeated.

st[op]

Suspend Emacs.

ta[g] tag

Find first definition of tag. If no tag is given, previously given tag is used and next alternate definition is find. By default, the file `TAGS' in the current directory becomes the selected tags table. You can select another tags table by set command. See section Customizing Constants, for details.

und[o]

Undo the last change.

unm[ap] ch

The macro expansion associated with ch is removed.

ve[rsion]

Tell the version number of VIP.

(1,$) w[rite] ! file

Write out specified lines into file file. If no file is given, text will be written to the file associated to the current buffer. Unless ! is given, if file is different from the file associated to the current buffer and if the file file exists, the command will not be executed. Unlike Ex, file becomes the file associated to the current buffer.

(1,$) w[rite]>> file

Write out specified lines at the end of file file. file becomes the file associated to the current buffer.

(1,$) wq ! file

Same as write and then quit. If ! is given, same as write ! then quit.

(.,.) y[ank] register count

Save specified lines into register register. If no register is specified, text will be saved in an anonymous register.

addr ! command

Execute shell command command. The output will be shown in a new window. If addr is given, specified lines will be used as standard input to command.

($) =

Print the line number of the addressed line.

(.,.) > count flags

Shift specified lines to the right. The variable `vip-shift-width` (default value is 8) determines the amount of shift.

(.,.) < count flags

Shift specified lines to the left. The variable `vip-shift-width` (default value is 8) determines the amount of shift.

(.,.) ~ options count flags

Repeat the previous substitute command using previous search pattern as pat for matching.

The following Ex commands are available in Vi, but not implemented in VIP.

```
abbreviate, list, next, print, preserve, recover, rewind, source,
unabbreviate, xit, z
```

Go to the previous, next section.

Go to the previous, next section.

Customization

If you have a file called `.vip' in your home directory, then it will also be loaded when VIP is loaded. This file is thus useful for customizing VIP.

Customizing Constants

An easy way to customize VIP is to change the values of constants used in VIP. Here is the list of the constants used in VIP and their default values.

`vip-shift-width 8`

The number of columns shifted by > and < command.

`vip-re-replace nil`

If `t` then do regexp replace, if `nil` then do string replace.

`vip-search-wrap-around t`

If `t`, search wraps around the buffer.

`vip-re-search nil`

If `t` then search is reg-exp search, if `nil` then vanilla search.

`vip-case-fold-search nil`

If `t` search ignores cases.

`vip-re-query-replace nil`

If `t` then do reg-exp replace in query replace.

`vip-open-with-indent nil`

If `t` then indent to the previous current line when open a new line by o or O command.

`vip-tags-file-name "TAGS"`

The name of the file used as the tags table.

`vip-help-in-insert-mode nil`

If `t` then C-h is bound to `help-command` in insert mode, if `nil` then it sis bound to `delete-backward-char`.

You can reset these constants in VIP by the Ex command set. Or you can include a line like this in your `.vip' file:

```
(setq vip-case-fold-search t)
```

Customizing Key Bindings

VIP uses `vip-command-mode-map` as the local keymap for vi mode. For example, in vi mode, SPC is bound to the function `vip-scroll`. But, if you wish to make SPC and some other keys behave like Vi, you can include the following lines in your `` `.vip' `` file.

```
(define-key vip-command-mode-map "\C-g" 'vip-info-on-file)
(define-key vip-command-mode-map "\C-h" 'vip-backward-char)
(define-key vip-command-mode-map "\C-m" 'vip-next-line-at-bol)
(define-key vip-command-mode-map " " 'vip-forward-char)
(define-key vip-command-mode-map "g" 'vip-keyboard-quit)
(define-key vip-command-mode-map "s" 'vip-substitute)
(define-key vip-command-mode-map "C" 'vip-change-to-eol)
(define-key vip-command-mode-map "R" 'vip-change-to-eol)
(define-key vip-command-mode-map "S" 'vip-substitute-line)
(define-key vip-command-mode-map "X" 'vip-delete-backward-char)
```

Go to the previous, next section.

Go to the previous, next section.

Key Index

0

1

TOC

- 177 DEL (`vip-delete-backward-char`)

Go to the previous, next section.

Go to the previous section.

Concept Index

a

b

c

d

e

f

g

i

k

l

m

n

p

r

s

t

v

w

y

Go to the previous section.